O'REILLY®

ВЫСОКО-НАГРУЖЕННЫЕ ПРИЛОЖЕНИЯ

Программирование масштабирование поддержка



ПИТЕР®

Мартин Клеппман

Designing Data-Intensive Applications

The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

Martin Kleppmann

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Мартин Клеппман

ВЫСОКО-НАГРУЖЕННЫЕ ПРИЛОЖЕНИЯ

Программирование, масштабирование, поддержка

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2018

ББК 32.973.233-018.2
УДК 004.65
К48

Клеппман М.

К48 Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб.: Питер, 2018. — 640 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-0512-0

В этой книге вы найдете ключевые принципы, алгоритмы и компромиссы, без которых не обойтись при разработке высоконагруженных систем для работы с данными. Материал рассматривается на примере внутреннего устройства популярных программных пакетов и фреймворков. В книге три основные части, посвященные, прежде всего, теоретическим аспектам работы с распределенными системами и базами данных. От читателя требуются базовые знания SQL и принципов работы баз данных.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.233-018.2
УДК 004.65

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1449373320 англ.

ISBN 978-5-4461-0512-0

Authorized Russian translation of the English edition of Designing Data-Intensive Applications
© 2017 Martin Kleppmann.
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same
© Перевод на русский язык ООО Издательство «Питер», 2018
© Издание на русском языке, оформление ООО Издательство «Питер», 2018
© Серия «Бестселлеры O'Reilly», 2018

Краткое содержание

Предисловие.....14

Часть I. Основы информационных систем

Глава 1. Надежные, масштабируемые и удобные в сопровождении приложения..... 23

Глава 2. Модели данных и языки запросов..... 53

Глава 3. Подсистемы хранения и извлечение данных..... 97

Глава 4. Кодирование и эволюция..... 143

Часть II. Распределенные данные

Глава 5. Репликация 185

Глава 6. Секционирование 239

Глава 7. Транзакции..... 265

Глава 8. Проблемы распределенных систем 323

Глава 9. Согласованность и консенсус..... 375

Часть III. Производные данные

Глава 10. Пакетная обработка 449

Глава 11. Потоковая обработка..... 505

Глава 12. Будущее информационных систем..... 563

Оглавление

Предисловие.....14
Кому стоит прочесть эту книгу.....16
Что рассматривается в издании.....17
Структура книги.....18
Ссылки и дополнительная литература.....19
Благодарности.....19

Часть I. Основы информационных систем

Глава 1. Надежные, масштабируемые и удобные в сопровождении приложения.....23
1.1. Подходы к работе над информационными системами.....24
1.2. Надежность.....27
Аппаратные сбои.....28
Программные ошибки.....29
Человеческий фактор.....30
Насколько важна надежность.....31
1.3. Масштабируемость.....32
Описание нагрузки.....32
Описание производительности.....36
Как справиться с нагрузкой.....42
1.4. Удобство сопровождения.....43
Удобство эксплуатации.....44
Простота: регулируем сложность.....45
Возможность развития: облегчаем внесение изменений.....47
1.5. Резюме.....47
1.6. Библиография.....48

Глава 2. Модели данных и языки запросов 53
2.1. Реляционная модель в сравнении с документоориентированной моделью 54
Рождение NoSQL 55
Объектно-реляционное несоответствие 56
Связи «многие-к-одному» и «многие-ко-многим» 59
Повторяется ли история в случае документоориентированных баз данных 62
Реляционные и документоориентированные базы данных сегодня 65
2.2. Языки запросов для данных 70
Декларативные запросы в Интернете 71
Выполнение запросов с помощью MapReduce 73
2.3. Графоподобные модели данных 76
Графы свойств 78
Язык запросов Cypher 79
Графовые запросы в SQL 81
Хранилища тройных кортежей и SPARQL 83
Фундамент: Datalog 88
2.4. Резюме 90
2.5. Библиография 92

Глава 3. Подсистемы хранения и извлечение данных 97
3.1. Базовые структуры данных БД 98
Хеш-индексы 100
SS-таблицы и LSM-деревья 104
B-деревья 108
Сравнение B- и LSM-деревьев 113
Другие индексные структуры 115
3.2. Обработка транзакций или аналитика? 120
Складирование данных 122
«Звезды» и «снежинки»: схемы для аналитики 125
3.3. Столбцовое хранилище 127
Сжатие столбцов 129
Порядок сортировки в столбцовом хранилище 131
Запись в столбцовое хранилище 132
Агрегирование: кубы данных и материализованные представления 133
3.4. Резюме 135
3.5. Библиография 136

Глава 4. Кодирование и эволюция ... 143
4.1. Форматы кодирования данных ... 144
Форматы, ориентированные на конкретные языки ... 145
JSON, XML и двоичные типы данных ... 146
Thrift и Protocol Buffers ... 150
Avro ... 154
Достоинства схем ... 160
4.2. Режимы движения данных ... 161
Поток данных через БД ... 162
Поток данных через сервисы: REST и RPC ... 164
Поток данных передачи сообщений ... 170
4.3. Резюме ... 173
4.4. Библиография ... 175

Часть II. Распределенные данные

Глава 5. Репликация ... 185
5.1. Ведущие и ведомые узлы ... 186
Синхронная и асинхронная репликация ... 188
Создание новых ведомых узлов ... 190
Перебои в обслуживании узлов ... 190
Реализация журналов репликации ... 193
5.2. Проблемы задержки репликации ... 196
Читаем свои же записи ... 197
Монотонные чтения ... 200
Согласованное префиксное чтение ... 201
Решения проблемы задержки репликации ... 202
5.3. Репликация с несколькими ведущими узлами ... 203
Сценарии использования репликации с несколькими ведущими узлами ... 204
Обработка конфликтов записи ... 207
Топологии репликации с несколькими ведущими узлами ... 212
5.4. Репликация без ведущего узла ... 214
Запись в базу данных при отказе одного из узлов ... 215
Ограничения согласованности по кворуму ... 218
Нестрогие кворумы и направленная передача ... 221
Обнаружение конкурентных операций записи ... 222
5.5. Резюме ... 230
5.6. Библиография ... 232

Глава 6. Секционирование 239
6.1. Секционирование и репликация 240
6.2. Секционирование данных типа «ключ — значение» 241
Секционирование по диапазонам значений ключа 242
Секционирование по хешу ключа 243
Асимметричные нагрузки и разгрузка горячих точек 245
6.3. Секционирование и вторичные индексы 246
Секционирование вторичных индексов по документам 247
Секционирование вторичных индексов по термам 248
6.4. Перебалансировка секций 250
Методики перебалансировки 250
Эксплуатация: автоматическая или ручная перебалансировка 254
6.5. Маршрутизация запросов 255
6.6. Резюме 259
6.7. Библиография 260

Глава 7. Транзакции 265
7.1. Неустоявшаяся концепция транзакции 267
Смысл аббревиатуры ACID 267
Однообъектные и многообъектные операции 272
7.2. Слабые уровни изоляции 277
Чтение зафиксированных данных 279
Изоляция снимков состояния и воспроизводимое чтение 282
Предотвращение потери обновлений 288
Асимметрия записи и фантомы 292
7.3. Сериализуемость 298
Последовательное выполнение 299
Двухфазная блокировка (2PL) 304
Сериализуемая изоляция снимков состояния (SSI) 308
7.4. Резюме 314
7.5. Библиография 316

Глава 8. Проблемы распределенных систем 323
8.1. Сбои и частичные отказы 324
8.2. Ненадежные сети 328
Сетевые сбои на практике 329
Обнаружение сбоев 331
Время ожидания и неограниченные задержки 332
Асинхронные и синхронные сети 335

8.3. Ненадежные часы 338
Монотонные часы и часы истинного времени 339
Синхронизация часов и их точность 341
Ненадежность синхронизированных часов 343
Паузы при выполнении процессов 348
8.4. Знание, истина и ложь 353
Истина определяется большинством 353
Византийские сбои 357
Модели системы на практике 360
8.5. Резюме 364
8.6. Библиография 366

Глава 9. Согласованность и консенсус 375
9.1. Гарантии согласованности 376
9.2. Линеаризуемость 378
Что делает систему линеаризуемой 379
Опора на линеаризуемость 384
Реализация линеаризуемых систем 387
Цена линеаризуемости 390
9.3. Гарантии упорядоченности 394
Порядок и причинность 395
Упорядоченность по порядковым номерам 399
Рассылка общей последовательности 404
9.4. Распределенные транзакции и консенсус 409
Атомарная и двухфазная фиксация (2PC) 411
Распределенные транзакции на практике 417
Отказоустойчивый консенсус 422
Сервисы членства и координации 428
9.5. Резюме 432
9.6. Библиография 435

Часть III. Производные данные

III.1. Системы записи и производные данные 446
III.2. Обзор глав 447

Глава 10. Пакетная обработка 449
10.1. Пакетная обработка средствами Unix 451
Простой анализ журнала 452
Философия Unix 454

10.2. MapReduce и распределенные файловые системы 458
Выполнение задач в MapReduce 460
Объединение и группировка на этапе сжатия 464
Объединения на этапе сопоставления 470
Выходные данные пакетных потоков 473
Сравнение Hadoop и распределенных баз данных 478
10.3. За пределами MapReduce 482
Материализация промежуточного состояния 483
Графы и итеративная обработка 488
API и языки высокого уровня 491
10.4. Резюме 494
10.5. Библиография 496

Глава 11. Потоковая обработка 505
11.1. Передача потоков событий 506
Системы обмена сообщениями 507
Секционирование журналов 513
11.2. Базы данных и потоки 519
Синхронизация систем 519
Перехват изменений данных 521
Источники событий 525
Состояние, потоки и неизменяемость 528
11.3. Обработка потоков 533
Применение обработки потоков 534
Рассуждения о времени 538
Объединения потоков 543
Отказоустойчивость 547
11.4. Резюме 551
11.5. Библиография 553

Глава 12. Будущее информационных систем 563
12.1. Интеграция данных 564
Объединение специализированных инструментов путем сбора информации 565
Пакетная и потоковая обработка 569
12.2. Отделение от баз данных 574
Объединение технологий хранения данных 575
Проектирование приложений на основе потока данных 580
Наблюдение за производными состояниями 586

12.3. Стремление к корректности 594
Сквозные аргументы в базе данных 595
Принудительные ограничения 600
Своевременность и целостность 604
Доверяй, но проверяй 609
12.4. Делать что должно 614
Предсказательная аналитика 615
Конфиденциальность и отслеживание 619
12.5. Резюме 627
12.6. Библиография 628

Обработка данных — поп-культура. ...Поп-культура презирает историю. Поп-культура всецело завязана на идентичности и ощущении себя частью чего-то. Для нее совершенно не важны сотрудничество, прошлое и будущее — она живет в настоящем. Я полагаю, что это же справедливо в отношении всех тех, кто пишет код за деньги. Они понятия не имеют, на чем основывается их культура.

Алан Кей[1]. *Из интервью Dr. Dobb's Journal* [2]

Технология в нашем обществе — могучая сила. Данные, программное обеспечение и связь можно использовать для неблаговидных целей: укрепления несправедливых властных структур, нарушения прав человека и защиты интересов власть имущих. Но они также потенциально полезны: с их помощью можно дать высказаться различным слоям населения, создавать перспективы для всех людей и предотвращать катастрофы. Эта книга посвящается всем, кто работает во благо людей.

---

1 http://www.drdobbs.com/architecture-and-design/interview-with-alan-kay/240003442.

2 См.: https://ru.wikipedia.org/wiki/Dr._Dobb's_Journal. — *Примеч. пер.*

Предисловие

Если в последнее время вам приходилось работать в сфере программной инженерии, особенно с серверными системами, то вас, вероятно, просто заваливали множеством модных словечек из области хранения и обработки данных. NoSQL! Большие данные! Масштабирование! Шардинг! Конечная согласованность! ACID! Теорема CAP! Облачные сервисы! MapReduce! В режиме реального времени!

За последнее десятилетие мы увидели немало интересных нововведений и усовершенствований в сферах баз данных (БД), распределенных систем, а также в способах создания работающих с ними приложений. Вот некоторые факторы, приведшие к этим усовершенствованиям.

- Такие интернет-компании, как Google, Yahoo!, Amazon, Facebook, LinkedIn, Microsoft и Twitter, обрабатывают колоссальные объемы данных и трафика, что вынуждает их создавать новые инструменты, подходящие для эффективной работы в подобных масштабах.
- Коммерческим компаниям приходится адаптироваться, проверять гипотезы с минимальными затратами и быстро реагировать на изменения рыночной обстановки путем сокращения циклов разработки и обеспечения гибкости моделей данных.
- Свободное программное обеспечение стало чрезвычайно популярным и во многих случаях является более предпочтительным по сравнению с коммерческим ПО и ПО для внутреннего использования.
- Тактовые частоты процессоров не слишком возросли, но многоядерные процессоры стали стандартом, плюс увеличились скорости передачи данных по сети. Это означает дальнейший рост параллелизма.
- Даже небольшая команда разработчиков может создавать системы, распределенные по множеству машин и географических регионов, благодаря такой IaaS (infrastructure as a service — «инфраструктура как сервис»), как Amazon Web Services.
- Многие сервисы стали высокодоступными; длительные простои из-за перебоев в обслуживании или текущих работ считаются все менее приемлемыми.

Высоконагруженные данными приложения (data-intensive applications, DIA) открывают новые горизонты возможностей благодаря использованию этих технологических усовершенствований. Мы говорим, что приложение является *высоконагруженным данными* (data-intensive), если те представляют основную проблему, с которой оно сталкивается, — качество данных, степень их сложности или скорость изменений, — в отличие от *высоконагруженного вычислениями* (compute-intensive), где узким местом являются циклы CPU.

Инструменты и технологии, обеспечивающие хранение и обработку данных с помощью DIA, быстро адаптировались к этим изменениям. В центр внимания попали новые типы систем баз данных (NoSQL), но очереди сообщений, кэши, поисковые индексы, фреймворки для пакетной и потоковой обработки и тому подобные технологии тоже очень важны. Многие приложения используют какое-либо их сочетание.

Наводняющие эту сферу модные словечки — признак горячего интереса к новым возможностям, что очень хорошо. Однако, как разработчики и архитекторы программного обеспечения, мы должны технически верно и точно понимать различные технологии, а также их достоинства и недостатки, если, конечно, собираемся создавать качественные приложения. Чтобы достичь подобного понимания, нам нужно заглянуть глубже, за модные словечки.

К счастью, за быстрой сменой технологий стоят принципы, остающиеся актуальными вне зависимости от используемой версии конкретной утилиты. Если вы понимаете эти принципы, то сумеете понять, где можно применить конкретную утилиту, как задействовать ее по максимуму и притом обойти подводные камни. Для этого и предназначена данная книга.

Цель издания — помочь вам проложить маршрут по многоликому и быстро меняющемуся ландшафту технологий для обработки и хранения данных. Эта книга — не учебное пособие по одному конкретному инструменту и не учебник, набитый сухой теорией. Напротив, мы изучим примеры успешно работающих информационных систем: технологий, которые составляют фундамент множества популярных приложений и в промышленной эксплуатации ежедневно должны отвечать требованиям по масштабированию, производительности и надежности.

Мы рассмотрим внутреннее устройство этих систем, разберемся в ключевых алгоритмах, обсудим их принципы и неизбежные компромиссы. В процессе попытаемся отыскать удобные *подходы* к информационным системам — не только к тому, *как* они работают, но и *почему* они работают именно так, а также какие вопросы необходимо задать себе о них.

После прочтения этой книги вы сможете решить, для каких задач подходят различные технологии, и понять принципы сочетания инструментов с целью закладки фундамента архитектуры хорошего приложения. Вряд ли вы будете готовы к созданию с нуля своей собственной подсистемы хранения базы данных, но это, к счастью,

требуется редко. Однако вы сумеете развить интуицию в отношении происходящего «за кулисами» ваших систем, что позволит судить об их поведении, принимать хорошие проектные решения и обнаруживать любые возникающие проблемы.

Кому стоит прочесть эту книгу

Если вы разрабатываете приложения, включающие какую-либо серверную/прикладную часть для хранения или обработки данных и использующие Интернет (например, веб-приложения, мобильные приложения либо подключенные через Сеть датчики), то эта книга — для вас.

Издание предназначено для разработчиков программного обеспечения, архитекторов ПО и технических директоров, которые в свободное время любят писать код. Оно особенно актуально, если вам нужно принять решение об архитектуре систем, над которыми вы работаете, — например, выбрать инструменты для решения конкретной проблемы и придумать, как их лучше использовать. Но даже если у вас нет права выбирать инструменты, данная книга поможет понять их достоинства и недостатки.

Желательно иметь опыт создания веб-приложений и сетевых сервисов. Кроме того, вы должны хорошо понимать принципы реляционных БД и знать язык SQL. Умение работать с какими-либо нереляционными БД и другими инструментами для управления данными приветствуется, но оно не обязательно. Не помешает иметь общее представление о распространенных сетевых протоколах, таких как TCP и HTTP. Выбранные вами язык программирования и фреймворк не имеют значения.

Если вы можете согласиться хоть с чем-то из перечисленного ниже, то эта книга будет полезна для вас.

- Вы хотели бы научиться создавать масштабируемые информационные системы, например, для поддержки веб- или мобильных приложений с миллионами пользователей.
- Вам необходимо обеспечить высокую доступность приложений (минимизировать время простоя) и их устойчивость к ошибкам в эксплуатации.
- Вы ищете способы упростить сопровождение систем в долгосрочной перспективе, даже в случае их роста и смены требований и технологий.
- Вам интересны механизмы процессов, и вы хотели бы разобраться, что происходит внутри крупных сайтов и онлайн-сервисов. В этой книге изучается внутреннее устройство множества различных БД и систем обработки данных, а исследовать воплощенные в их архитектуре блестящие идеи — чрезвычайно увлекательное дело.

Иногда при разговоре о масштабируемых информационных системах люди высказываются примерно следующим образом: «Ты не Google и не Amazon. Перестань волноваться о масштабировании и просто воспользуйся реляционной базой данных». В приведенном утверждении есть доля истины: создание приложения в расчете на масштабы, которые вам не понадобятся, — пустая трата сил, способная привести к неадаптивному дизайну. По существу, это один из видов преждевременной оптимизации. Однако важно также выбрать правильный инструмент для стоящей перед вами задачи, а у каждой технологии — свои плюсы и минусы. Как мы увидим далее, реляционные БД — важное, но отнюдь не последнее слово техники при работе с данными.

Что рассматривается в издании

Здесь не приводятся подробные инструкции по установке или использованию конкретных пакетов программ либо различных API, поскольку по данным вопросам существует множество документации. Вместо этого мы обсуждаем различные принципы и компромиссы, базовые для информационных систем, а также исследуем проектные решения разных программных продуктов.

Все ссылки на онлайн-ресурсы в этой книге проверены на момент публикации, но, к сожалению, они склонны часто портиться по естественным причинам. Если вы наткнетесь на «битую» ссылку, то можете найти ресурс с помощью поисковых систем. Что касается научных статей, можно найти находящиеся в свободном доступе PDF-файлы, вводя их названия в Google Scholar. Кроме того, все ссылки можно найти по адресу https://github.com/ept/ddia-references, где мы следим за их актуальностью.

Нас интересуют в основном *архитектура* информационных систем и способы их интеграции в высоконагруженные данными приложения. В настоящей книге недостаточно места, чтобы охватить развертывание, эксплуатацию, безопасность, управление и другие сферы — это сложные и важные темы, и мы не отдали бы им должное, выделив под них поверхностные заметки на полях. Они заслуживают отдельных книг.

Многие из представленных в этой книге технологий относятся к сфере, описываемой модным словосочетанием *«большие данные»*. Однако одноименный термин настолько плохо определен и им так сильно злоупотребляют, что в серьезном техническом обсуждении он пользы не принесет. В книге применяются менее расплывчатые термины, например одноузловые системы в отличие от распределенных или онлайн/диалоговые системы обработки данных вместо офлайн/пакетных.

В книге мы склоняемся к использованию свободного программного обеспечения (с открытым исходным кодом — free and open source software, FOSS)[1], поскольку чтение, изменение и выполнение исходного кода — прекрасный способ лучше понять, как что-либо работает. Кроме того, открытость платформы снижает риск замыкания на одном поставщике. Однако везде, где это имело смысл, мы обсуждаем также проприетарное ПО (ПО с закрытым исходным кодом, ПО как сервис и ПО для внутреннего пользования, которое лишь описывается в документации, но не выпускается для всеобщего применения).

Структура книги

Книга разделена на три части.

1. В части I мы обсудим базовые идеи, лежащие в основе проектирования DIA. В главе 1 мы начнем с обсуждения того, чего же на самом деле хотим добиться: надежности, масштабируемости и удобства сопровождения; какие подходы возможны и как достичь этих целей. В главе 2 сравним несколько моделей данных и языков запросов и рассмотрим, какие из них подходят для различных ситуаций. В главе 3 мы обсудим подсистемы хранения: как БД выстраивают данные на диске таким образом, чтобы их можно было эффективно находить. Глава 4 обращается к вопросу форматов кодирования данных (сериализации) и эволюции схем с течением времени.
2. В части II мы перейдем от данных, хранимых на одной машине, к данным, распределенным по нескольким компьютерам. Такое положение дел зачастую необходимо для хорошей масштабируемости, но требует решения множества непростых специфических задач. Сначала мы обсудим репликацию (глава 5), секционирование/шардинг (глава 6) и транзакции (глава 7). Затем углубимся в возникающие при работе с распределенными системами проблемы (глава 8) и разберем, что такое согласованность и консенсус в распределенной системе (глава 9).
3. В части III мы обсудим системы, производящие одни наборы данных из других наборов. Производные данные часто встречаются в неоднородных системах: когда невозможно найти одну БД, работающую удовлетворительно во всех случаях, приложениям приходится интегрировать несколько различных баз, кэшей, индексов и т. д. В главе 10 мы начнем с пакетного подхода к обработке производных данных и придем на его основе к потоковой обработке в главе 11. Наконец, в главе 12 мы объединим всю информацию и обсудим подходы к созданию надежных, масштабируемых и удобных в сопровождении приложений.

---

[1] См. более детальное объяснение этого термина здесь: https://www.gnu.org/philosophy/floss-and-foss.ru.html — *Примеч. пер.*

Ссылки и дополнительная литература

Большая часть обсуждаемых в книге вопросов уже была изложена где-то в той или иной форме — в презентациях на конференциях, научных статьях, размещенных в блогах сообщениях, исходном коде, системах отслеживания ошибок, почтовых рассылках и технических форумах. Эта книга подытоживает важнейшие идеи из множества различных источников и содержит ссылки на исходную литературу. Перечень в конце каждой из глав — замечательный источник информации на случай, если вы захотите изучить какой-либо из вопросов подробнее, причем большая часть их есть в свободном доступе в Интернете.

Благодарности

В настоящей книге скомбинировано и систематизировано множество чужих идей и знаний, при этом объединяется опыт, полученный при научных исследованиях, с опытом промышленной эксплуатации. В сфере обработки данных привлекают «новые и блестящие» вещи, но я полагаю, что опыт предшественников может быть очень полезным. В книге содержится более 800 ссылок на статьи, размещенные в блогах сообщения, обсуждения, документацию и т. п., и все перечисленное представляется мне бесценным образовательным ресурсом. Я чрезвычайно благодарен авторам материалов, поделившимся этими знаниями.

Я также многое узнал из личных разговоров с множеством людей, потративших время на обсуждение некоторых тем или терпеливо объяснявших мне какие-либо вещи. В частности, я хотел бы поблагодарить Джо Адлера (Joe Adler), Росса Андерсона (Ross Anderson), Питера Бэйлиса (Peter Bailis), Мартона Баласси (Márton Balassi), Аластера Бересфорда (Alastair Beresford), Марка Кэллахана (Mark Callaghan), Мэта Клэйтона (Mat Clayton), Патрика Коллисона (Patrick Collison), Шона Криббса (Sean Cribbs), Ширшанку Дэса (Shirshanka Das), Никласа Экстрема (Niklas Ekström), Стивена Ивена (Stephan Ewen), Алана Фекете (Alan Fekete), Гьюлу Фора (Gyula Fóra), Камиллу Фурнье (Camille Fournier), Андреса Френда (Andres Freund), Джона Гарбута (John Garbutt), Сета Джильберта (Seth Gilbert), Тома Хаггета (Tom Haggett), Пэт Хелланд (Pat Helland), Джо Хеллерстайна (Joe Hellerstein), Джейкоба Хомана (Jakob Homan), Гейди Говард (Heidi Howard), Джона Хагга (John Hugg), Джулиана Хайда (Julian Hyde), Конрада Ирвина (Conrad Irwin), Эвана Джонса (Evan Jones), Флавио Жункейру (Flavio Junqueira), Джессику Керр (Jessica Kerr), Кайла Кингсбэри (Kyle Kingsbury), Джея Крепса (Jay Kreps), Карла Лерке (Carl Lerche), Николаса Льошона (Nicolas Liochon), Стива Локрана (Steve Loughran), Ли Мэллабона (Lee Mallabone), Нэйтана Марца (Nathan Marz), Кэйти Маккэфри (Caitie McCaffrey), Джози МакЛеллан (Josie McLellan), Кристофера Мейклджона (Christopher Meiklejohn), Йэна Мейерса (Ian Meyers), Неху Нархеде (Neha Narkhede), Неху Нарула (Neha Narula), Кэти О'Нейл (Cathy O'Neil), Онору О'Нейль (Onora O'Neill), Людовика Орбана (Ludovic Orban),

Зорана Перкова (Zoran Perkov), Джулию Поульс (Julia Powles), Криса Риккомини (Chris Riccomini), Генри Робинсона (Henry Robinson), Дэвида Розенталя (David Rosenthal), Дженнифер Рульманн (Jennifer Rullmann), Мэтью Сакмана (Matthew Sackman), Мартина Шоля (Martin Scholl), Амита Села (Amit Sela), Гвен Шапира (Gwen Shapira), Грега Спурье (Greg Spurrier), Сэма Стоукса (Sam Stokes), Бена Стопфорда (Ben Stopford), Тома Стюарта (Tom Stuart), Диану Василе (Diana Vasile), Рахула Вохра (Rahul Vohra), Пита Уордена (Pete Warden) и Брета Вулриджа (Brett Wooldridge).

Еще несколько человек оказали неоценимую помощь при написании данной книги, прочитав черновики и сделав замечания. За этот вклад я особенно благодарен Раулю Агепати (Raul Agepati), Тайлеру Акидо (Tyler Akidau), Мэтиасу Андерссону (Mattias Andersson), Саше Баранову (Sasha Baranov), Вине Басаварадж (Veena Basavaraj), Дэвиду Бееру (David Beyer), Джиму Брикману (Jim Brikman), Полу Кайри (Paul Carey), Раулю Кастро Фернандесу (Raul Castro Fernandez), Джозефу Чау (Joseph Chow), Дереку Элкинсу (Derek Elkins), Сэму Эллиоту (Sam Elliott), Александру Галлего (Alexander Gallego), Марку Груверу (Mark Grover), Стю Хэлловэю (Stu Halloway), Гейди Говард (Heidi Howard), Никола Клепманну (Nicola Kleppmann), Стефану Круппе (Stefan Kruppa), Бьерну Мадсену (Bjorn Madsen), Сэндеру Мэку (Sander Mak), Стефану Подковински (Stefan Podkowinski), Филу Поттеру (Phil Potter), Хамиду Рамазани (Hamid Ramazani), Сэму Стоуксу (Sam Stokes) и Бену Саммерсу (Ben Summers).

Я благодарен моим редакторам, Мари Богюро (Marie Beaugureau), Майку Локидесу (Mike Loukides), Энн Спенсер (Ann Spencer) и всей команде издательства O'Reilly за помощь, без которой эта книга не увидела бы свет, и за то, что они терпеливо относились к моему медленному темпу письма и необычным вопросам. Я благодарен Рэйчел Хэд (Rachel Head) за помощь в подборе нужных слов. За предоставление мне свободного времени и возможности писать, несмотря на другую работу, я благодарен Аластеру Бересфорду (Alastair Beresford), Сьюзен Гудхью (Susan Goodhue), Нехе Нархеде (Neha Narkhede) и Кевину Скотту (Kevin Scott).

Особая благодарность Шабиру Дайвэну (Shabbir Diwan) и Эдди Фридману (Edie Freedman), тщательнейшим образом создавшим иллюстрации карт, сопутствующих главам книги. Чудесно, что они согласились с нестандартной идеей создания карт и сделали их столь красивыми и интересными.

Наконец, я говорю слова любви моей семье и друзьям. Без них я наверняка не смог бы завершить процесс написания, занявший почти четыре года. Вы лучше всех!

Часть I
Основы информационных систем

Первые четыре главы этой книги посвящены основным идеям, относящимся ко всем информационным системам, работающим на одной машине или распределенным по кластеру.

- В главе 1 вы познакомитесь с терминологией и общим подходом данной книги. Здесь обсуждается, что мы понимаем под словами *«надежность»* (reliability), *«масштабируемость»* (scalability) и *«удобство сопровождения»* (maintainability), а также как достичь этих целей.
- В главе 2 сравниваются несколько разных моделей данных и языков запросов — наиболее заметных качественных различий баз данных с точки зрения разработчика. Кроме того, мы рассмотрим, насколько различные модели подходят для разных ситуаций.
- В главе 3 мы обратимся к внутреннему устройству подсистем хранения и размещению данных на диске теми или иными СУБД. Различные подсистемы хранения оптимизированы под разные нагрузки, и выбор правильной подсистемы может оказать колоссальное влияние на производительность.
- В главе 4 сравниваются разнообразные форматы кодирования данных (сериализации), особенно их работа в средах с меняющимися требованиями приложений, к которым необходимо со временем адаптировать схемы.

Позже, в части II, мы обратимся к конкретным проблемам распределенных информационных систем.

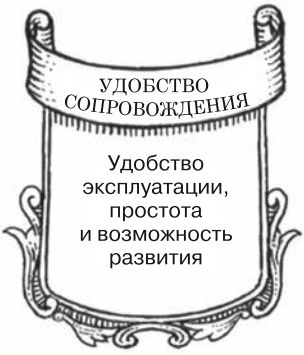
Разработка высоконагруженных данными приложений

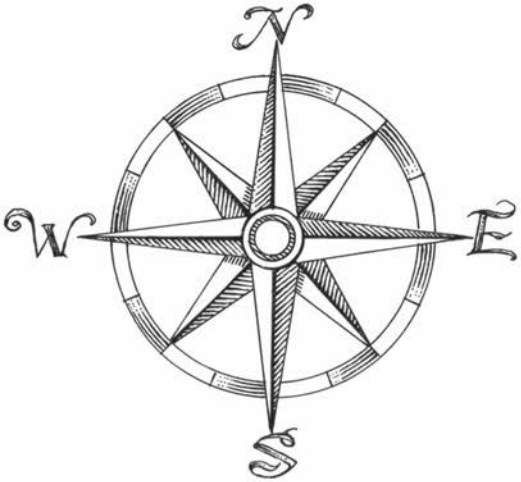
Программирование, масштабирование, поддержка











1 Надежные, масштабируемые и удобные в сопровождении приложения

> Интернет настолько хорош, что большинство людей считает его природным ресурсом, таким как Тихий океан, а не чем-то сотворенным руками человека. Когда в последний раз в технологии подобного масштаба не было ни одной ошибки?
>
> *Алан Кей[1]. Из интервью Dr. Dobb's Journal (2012)*

Многие приложения сегодня относятся к категории *высоконагруженных данными* (data-intensive), в отличие от *высоконагруженных вычислениями* (compute-intensive). Чистая производительность CPU — просто ограничивающий фактор для этих приложений, а основная проблема заключается в объеме данных, их сложности и скорости изменения.

Высоконагруженное данными приложение (DIA) обычно создается из стандартных блоков, обеспечивающих часто требующуюся функциональность. Например, многим приложениям нужно:

- хранить данные, чтобы эти или другие приложения могли найти их в дальнейшем *(базы данных)*;
- запоминать результат ресурсоемкой операции для ускорения чтения *(кэши)*;
- предоставлять пользователям возможность искать данные по ключевому слову или фильтровать их различными способами *(поисковые индексы)*;

[1] http://www.drdobbs.com/architecture-and-design/interview-with-alan-kay/240003442.

- отправлять сообщения другим процессам для асинхронной обработки *(потоковая обработка)*;
- время от времени «перемалывать» большие объемы накопленных данных *(пакетная обработка)*.

Все описанное выглядит до боли очевидным лишь потому, что *информационные системы* — очень удачная абстракция: мы используем их все время, даже не задумываясь. При создании приложения большинство разработчиков и не помышляют о создании с нуля новой подсистемы хранения, поскольку базы данных — инструмент, отлично подходящий для этой задачи.

Но в жизни не все так просто. Существует множество систем баз данных с разнообразными характеристиками, поскольку у разных приложений — различные требования. Существует много подходов к кэшированию, несколько способов построения поисковых индексов и т. д. При создании приложения необходимо определиться с тем, с помощью каких инструментов и подходов лучше всего решать имеющуюся задачу. Кроме того, иногда бывает непросто подобрать нужную комбинацию инструментов, когда нужно сделать что-то, для чего одного инструмента в отдельности недостаточно.

Эта книга проведет обширный экскурс в страну как теоретических принципов, так и практических аспектов информационных систем, а также возможностей их использования для создания высоконагруженных данными приложений. Мы выясним, что объединяет различные инструменты и отличает их, а также то, как они получат свои характеристики.

В этой главе мы начнем с изучения основ того, чего хотим добиться: надежности, масштабируемости и удобства сопровождения информационных систем. Мы проясним, что означают указанные понятия, обрисуем отдельные подходы к работе с информационными системами и пройдемся по необходимым для следующих глав основам. В следующих главах мы продолжим наш послойный анализ, рассмотрим различные проектные решения, которые целесообразно принять во внимание при работе над DIA.

1.1. Подходы к работе над информационными системами

Обычно мы относим базы данных, очереди, кэши и т. п. к совершенно различным категориям инструментов. Хотя базы данных и очереди сообщений выглядят похожими — и те и другие хранят данные в течение некоторого времени, — паттерны доступа для них совершенно различаются, что означает различные характеристики производительности, а следовательно, очень разные реализации.

Так зачем же валить их все в одну кучу под таким собирательным термином, как *информационные системы*?

В последние годы появилось множество новых инструментов для хранения и обработки данных. Они оптимизированы для множества различных сценариев использования и более не укладываются в обычные категории [1]. Например, существуют хранилища данных, применяемые как очереди сообщений (Redis) и очереди сообщений с соответствующим базам данных уровнем надежности (Apache Kafka). Границы между категориями постепенно размываются.

Кроме того, все больше приложений предъявляют такие жесткие или широкие требования, что отдельная утилита уже не способна обеспечить все их потребности в обработке и хранении данных. Поэтому работа разбивается на отдельные задачи, которые *можно* эффективно выполнить с помощью отдельного инструмента, и эти различные инструменты объединяются кодом приложения.

Например, при наличии управляемого приложением слоя кэширования (путем использования Memcached или аналогичного инструмента) либо сервера полнотекстового поиска (такого как Elasticsearch или Solr), отдельно от БД, синхронизация этих кэшей и индексов с основной базой данных становится обязанностью кода приложения. На рис. 1.1 в общих чертах показано, как все указанное могло бы выглядеть (более подробно мы рассмотрим этот вопрос в следующих главах).

Если для предоставления сервиса объединяется несколько инструментов, то интерфейс сервиса или программный интерфейс приложения (API) обычно скрывает подробности реализации от клиентских приложений. По существу, мы создали новую специализированную информационную систему из более мелких, универсальных компонентов. Получившаяся объединенная информационная система может гарантировать определенные вещи: например, что кэш будет корректно сделан недействительным или обновлен при записи, вследствие чего внешние клиенты увидят непротиворечивые результаты. Вы теперь не только разработчик приложения, но и архитектор информационной системы.

При проектировании информационной системы или сервиса возникает множество непростых вопросов. Как обеспечить правильность и полноту данных, в том числе при внутренних ошибках? Как обеспечить одинаково хорошую производительность для всех клиентов даже в случае ухудшения рабочих характеристик некоторых частей системы? Как обеспечить масштабирование для учета возросшей нагрузки? Каким должен быть хороший API для этого сервиса?

Существует множество факторов, влияющих на конструкцию информационной системы, включая навыки и опыт вовлеченных в проектирование специалистов, унаследованные системные зависимости, сроки поставки, степень приемлемости разных видов риска для вашей компании, законодательные ограничения и т. д. Эти факторы очень сильно зависят от конкретной ситуации.

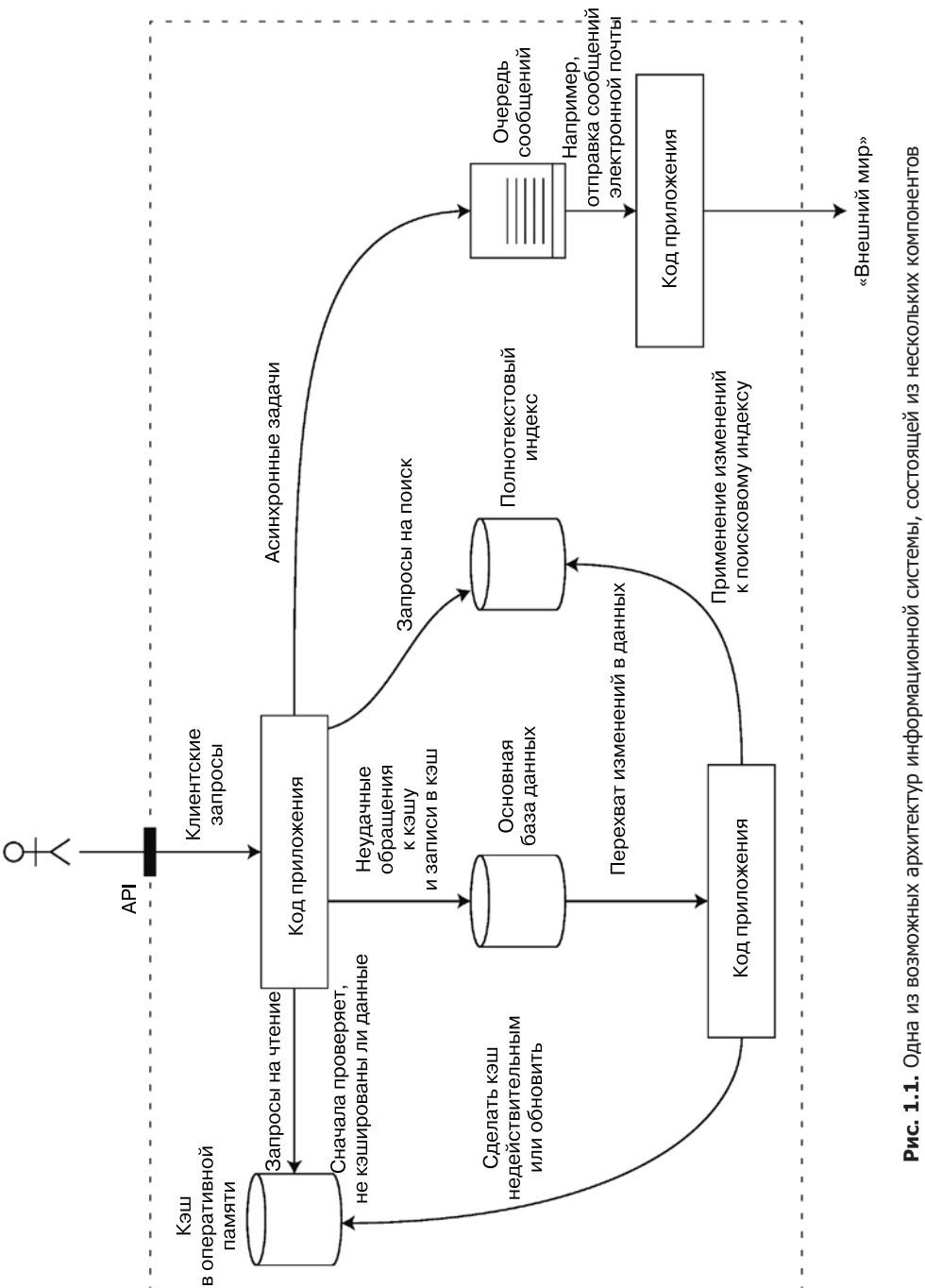


Рис. 1.1. Одна из возможных архитектур информационной системы, состоящей из нескольких компонентов

В данной книге мы сосредоточимся на трех вопросах, имеющих наибольшее значение в большинстве программных систем.

- *Надежность*. Система должна продолжать работать *корректно* (осуществлять нужные функции на требуемом уровне производительности) даже при *неблагоприятных обстоятельствах* (в случае аппаратных или программных сбоев либо ошибок пользователя). См. раздел 1.2.
- *Масштабируемость*. Должны быть предусмотрены разумные способы решения возникающих при *росте* (в смысле объемов данных, трафика или сложности) системы проблем. См. раздел 1.3.
- *Удобство сопровождения*. Необходимо обеспечить возможность эффективной работы с системой множеству различных людей (разработчикам и обслуживающему персоналу, занимающимся как поддержкой текущего функционирования, так и адаптацией системы к новым сценариям применения). См. раздел 1.4.

Приведенные термины часто задействуют без четкого понимания их смысла. Ради более осмысленного проектирования мы потратим остаток главы на изучение концепций надежности, масштабируемости и удобства сопровождения. Далее, в следующих главах, мы рассмотрим различные методики, архитектуры и алгоритмы, используемые для достижения этих целей.

1.2. Надежность

Каждый интуитивно представляет, что значит быть надежным или ненадежным. От программного обеспечения обычно ожидается следующее:

- приложение выполняет ожидаемую пользователем функцию;
- оно способно выдержать ошибочные действия пользователя или применение программного обеспечения неожиданным образом;
- его производительность достаточно высока для текущего сценария использования, при предполагаемой нагрузке и объеме данных;
- система предотвращает любой несанкционированный доступ и неправильную эксплуатацию.

Если считать, что все указанное означает «работать нормально», то термин «надежность» будет иметь значение, грубо говоря, «продолжать работать нормально даже в случае проблем».

Возможные проблемы называются *сбоями*, а системы, созданные в расчете на них, называются *устойчивыми к сбоям*. Этот термин способен ввести в некоторое заблуждение: он наводит на мысль, что можно сделать систему устойчивой ко всем возможным видам сбоев. Однако на практике это неосуществимо. Если наша планета (и все находящиеся на ней серверы) будет поглощена черной дырой, то для обеспечения устойчивости к такому «сбою» потребовалось бы размещение данных

в космосе — удачи вам в одобрении подобной статьи бюджета. Так что имеет смысл говорить об устойчивости лишь к *определенным типам* сбоев.

Обратите внимание, что сбой (fault) и отказ (failure) — разные вещи [2]. Сбой обычно определяется как отклонение одного из компонентов системы от рабочих характеристик, в то время как *отказ* — ситуация, когда вся система в целом прекращает предоставление требуемого сервиса пользователю. Снизить вероятность сбоев до нуля невозможно, следовательно, обычно лучше проектировать механизмы устойчивости к сбоям, которые бы предотвращали переход сбоев в отказы. В этой книге мы рассмотрим несколько методов создания надежных систем из ненадежных составных частей.

Парадоксально, но в подобных устойчивых к сбоям системах имеет смысл *повысить* частоту сбоев с помощью их умышленной генерации — например, путем прерывания работы отдельных, выбранных случайным образом процессов без предупреждения. Многие критические ошибки фактически происходят из-за недостаточной обработки ошибок [3]; умышленное порождение сбоев гарантирует постоянное тестирование механизмов обеспечения устойчивости к ним, что повышает уверенность в должной обработке сбоев при их «естественном» появлении. Пример этого подхода — сервис *Chaos Monkey* компании Netflix [4].

Хотя обычно считается, что устойчивость системы к сбоям важнее их предотвращения, существуют случаи, когда предупреждение лучше лечения (например, когда «лечения» не существует). Это справедливо в случае вопросов безопасности: если атакующий скомпрометировал систему и получил доступ к конфиденциальным данным, то ничего поделать уже нельзя. Однако в этой книге по большей части речь идет о сбоях, допускающих «лечение», как описывается ниже.

Аппаратные сбои

Когда речь идет о причинах отказов систем, первым делом в голову приходят аппаратные сбои. Фатальные сбои винчестеров, появление дефектов ОЗУ, отключение электропитания, отключение кем-то не того сетевого кабеля. Любой, кто имел дело с большими центрами обработки и хранения данных, знает, что подобное происходит *постоянно* при наличии большого количества машин.

Считается, что среднее время наработки на отказ (mean time to failure, MTTF) винчестеров составляет от 10 до 50 лет [5, 6]. Таким образом, в кластере хранения с 10 тысячами винчестеров следует ожидать в среднем одного отказа жесткого диска в день.

Первая естественная реакция на эту информацию — повысить избыточность отдельных компонентов аппаратного обеспечения с целью снизить частоту отказов системы. Можно создать RAID-массивы из дисков, обеспечить дублирование электропитания серверов и наличие в них CPU с возможностью горячей замены,

а также запастись батареями и дизельными генераторами в качестве резервных источников электропитания ЦОДов. При отказе одного компонента его место на время замены занимает резервный компонент. Такой подход не предотвращает полностью отказы, возникающие из-за проблем с оборудованием, но вполне приемлем и часто способен поддерживать бесперебойную работу машин в течение многих лет.

До недавних пор избыточность компонентов аппаратного обеспечения была достаточной для большинства приложений, делая критический отказ отдельной машины явлением вполне редким. При наличии возможности достаточно быстрого восстановления из резервной копии на новой машине время простоя в случае отказа не катастрофично для большинства приложений. Следовательно, многомашинная избыточность требовалась только небольшой части приложений, для которых была критически важна высокая доступность.

Однако по мере роста объемов данных и вычислительных запросов приложений все больше программ начали использовать большее количество машин, что привело к пропорциональному росту частоты отказов оборудования. Более того, на многих облачных платформах, таких как Amazon Web Services (AWS), экземпляры виртуальных машин достаточно часто становятся недоступными без предупреждения [7], поскольку платформы отдают предпочтение гибкости и способности быстро адаптироваться[1] перед надежностью одной машины.

Поэтому происходит сдвиг в сторону систем, способных перенести потерю целых машин, благодаря применению методов устойчивости к сбоям вместо избыточности аппаратного обеспечения или дополнительно к ней. У подобных систем есть и эксплуатационные преимущества: система с одним сервером требует планового простоя при необходимости перезагрузки машины (например, для установки исправлений безопасности), в то время как устойчивая к аппаратным сбоям система допускает установку исправлений по узлу за раз, без вынужденного бездействия всей системы (*плавающее обновление*; см. главу 4).

Программные ошибки

Обычно считают так: аппаратные сбои носят случайный характер и независимы друг от друга: отказ диска одного компьютера не означает, что диск другого скоро тоже начнет сбоить. Конечно, возможны слабые корреляции (например, вследствие общей причины наподобие температуры в серверной стойке), но в остальных случаях одновременный отказ большого количества аппаратных компонентов маловероятен.

Другой класс сбоев — систематическая ошибка в системе [8]. Подобные сбои сложнее предотвратить, и в силу их корреляции между узлами они обычно вызывают

[1] Определение этому термину дается в подразделе «Как справиться с нагрузкой» раздела 1.3.

гораздо больше системных отказов, чем некоррелируемые аппаратные сбои [5]. Рассмотрим примеры.

- Программная ошибка, приводящая к фатальному сбою экземпляра сервера приложения при конкретных «плохих» входных данных. Например, возьмем секунду координации 30 июня 2012 года, вызвавшую одновременное зависание множества приложений из-за ошибки в ядре операционной системы Linux [9].
- Выходит из-под контроля процесс, полностью исчерпавший какой-либо общий ресурс: время CPU, оперативную память, пространство на диске или полосу пропускания сети.
- Сервис, от которого зависит работа системы, замедляется, перестает отвечать на запросы или начинает возвращать испорченные ответы.
- Каскадные сбои, при которых крошечный сбой в одном компоненте вызывает сбой в другом компоненте, а тот, в свою очередь, вызывает дальнейшие сбои [10].

Ошибки, вызывающие подобные программные сбои, часто долго остаются неактивными, вплоть до момента срабатывания под влиянием необычных обстоятельств. При этом оказывается, что приложение делает какое-либо допущение относительно своего окружения — и хотя обычно такое допущение справедливо, в конце концов оно становится неверным по какой-либо причине [11].

Быстрого решения проблемы систематических ошибок в программном обеспечении не существует. Может оказаться полезным множество мелочей, таких как: тщательное обдумывание допущений и взаимодействий внутри системы; всестороннее тестирование; изоляция процессов; предоставление процессам возможности перезапуска после фатального сбоя; оценка, мониторинг и анализ поведения системы при промышленной эксплуатации. Если система должна обеспечивать выполнение какого-либо условия (например, в очереди сообщений количество входящих сообщений должно быть равно количеству исходящих), то можно организовать постоянную самопроверку во время работы и выдачу предупреждения в случае обнаружения расхождения [12].

Человеческий фактор

Проектируют и создают программные системы люди; ими же являются и операторы, обеспечивающие их функционирование. Даже при самых благих намерениях люди ненадежны. Например, одно исследование крупных интернет-сервисов показало, что основной причиной перебоев в работе были допущенные операторами ошибки в конфигурации, в то время как сбои аппаратного обеспечения (серверов или сети) играли какую-либо роль лишь в 10–25 % случаев [13].

Как же обеспечить надежность нашей системы, несмотря на ненадежность людей? Оптимальные системы сочетают в себе несколько подходов.

- Проектирование систем таким образом, который минимизировал бы возможности появления ошибок. Например, грамотно спроектированные абстрак-

ции, API и интерфейсы администраторов упрощают «правильные» действия и усложняют «неправильные». Однако если интерфейсы будут слишком жестко ограничены, то люди начнут искать пути обхода; это приведет к нивелированию получаемой от таких интерфейсов выгоды, так что самое сложное здесь — сохранить равновесие.

- Расцепить наиболее подверженные человеческим ошибкам места системы с теми местами, где ошибки могут привести к отказам. В частности, предоставить не для промышленной эксплуатации полнофункциональную среду-*«песочницу»*, в которой можно было бы безопасно изучать работу и экспериментировать с системой с помощью настоящих данных, не влияя на реальных пользователей.
- Выполнять тщательное тестирование на всех уровнях, начиная с модульных тестов и заканчивая комплексным тестированием всей системы и ручными тестами [3]. Широко используется автоматизированное тестирование, вполне приемлемое и особенно ценное для пограничных случаев, редко возникающих при нормальной эксплуатации.
- Обеспечить возможность быстрого и удобного восстановления после появления ошибок для минимизации последствий в случае отказа. Например, предоставить возможность быстрого отката изменений конфигурации, постепенное внедрение нового кода (чтобы все неожиданные ошибки оказывали влияние на небольшое подмножество пользователей) и возможность использования утилит для пересчета данных (на случай, если окажется, что предыдущие вычисления были неправильными).
- Настроить подробный и ясный мониторинг, в том числе метрик производительности и частот ошибок. В других областях техники это носит название *телеметрии*. (После отрыва ракеты от земли телеметрия превращается в важнейшее средство отслеживания происходящего и выяснения причин отказов [14].) Мониторинг обеспечивает диагностические сигналы на ранних стадиях и позволяет проверять, не были ли нарушены правила или ограничения. При возникновении проблемы метрики оказываются бесценным средством диагностики проблем.
- Внедрение рекомендуемых управленческих практик и обучение — сложный и важный аспект, выходящий за рамки данной книги.

Насколько важна надежность?

Надежность нужна не только в управляющем программном обеспечении атомных электростанций и воздушного сообщения — от обычных приложений тоже ожидается надежная работа. Ошибки в коммерческих приложениях приводят к потерям производительности (и юридическим рискам, если цифры в отчетах оказались неточны), а простои сайтов интернет-магазинов могут приводить к колоссальным убыткам в виде недополученных доходов и ущерба для репутации.

Создатели даже «некритичных» приложений несут ответственность перед пользователями. Возьмем, например, родителей, хранящих все фотографии и видео своих детей в вашем приложении для фото [15]. Как они будут себя чувствовать,

если база данных неожиданно окажется испорченной? Смогут ли они восстановить данные из резервной копии?

Встречаются ситуации, в которых приходится пожертвовать надежностью ради снижения стоимости разработки (например, при создании прототипа продукта для нового рынка) или стоимости эксплуатации (например, для сервиса с очень низкой маржой прибыли) — но «срезать углы» нужно очень осторожно.

1.3. Масштабируемость

Даже если на сегодняшний момент система работает надежно, нет гарантий, что она будет так же работать в будущем. Одна из частых причин снижения эффективности — рост нагрузки: например, система выросла с 10 тыс. работающих одновременно пользователей до 100 тыс. или с 1 до 10 млн. Это может быть и обработка значительно бóльших объемов данных, чем ранее.

Масштабируемость (scalability) — термин, который я буду использовать для описания способности системы справляться с возросшей нагрузкой. Отмечу, однако, что это не одномерный ярлык, который можно «навесить» на систему: фразы «X — масштабируемая» или «Y — немасштабируемая» бессмысленны. Скорее, обсуждение масштабирования означает рассмотрение следующих вопросов: «Какими будут наши варианты решения проблемы, если система вырастет определенным образом?» и «Каким образом мы можем расширить вычислительные ресурсы в целях учета дополнительной нагрузки?».

Описание нагрузки

Во-первых, нужно сжато описать текущую нагрузку на систему: только тогда мы сможем обсуждать вопросы ее роста («Что произойдет, если удвоить нагрузку?»). Нагрузку можно описать с помощью нескольких чисел, которые мы будем называть *параметрами нагрузки*. Оптимальный выбор таких параметров зависит от архитектуры системы. Это может быть количество запросов к веб-серверу в секунду, отношение количества операций чтения к количеству операций записи в базе данных, количество активных одновременно пользователей в комнате чата, частота успешных обращений в кэш или что-то еще. Возможно, для вас будет важно среднее значение, а может, узкое место в вашей ситуации будет определяться небольшим количеством предельных случаев.

Чтобы прояснить эту идею, рассмотрим в качестве примера социальную сеть Twitter, задействуя данные, опубликованные в ноябре 2012 года [16]. Две основные операции сети Twitter таковы:

- *публикация твита* — пользователь может опубликовать новое сообщение для своих подписчиков (в среднем 4600 з/с, пиковое значение — более 12 000 з/с);
- *домашняя лента* — пользователь может просматривать твиты, опубликованные теми, на кого он подписан (300 000 з/с).

Просто обработать 12 тысяч записей в секунду (пиковая частота публикации твитов) должно быть несложно. Однако проблема с масштабированием сети Twitter состоит не в количестве твитов, а в *коэффициенте разветвления по выходу*[1] — каждый пользователь подписан на множество людей и, в свою очередь, имеет много подписчиков. Существует в общих чертах два способа реализации этих двух операций.

1. Публикация твита просто приводит к вставке нового твита в общий набор записей. Когда пользователь отправляет запрос к своей домашней ленте, выполняется поиск всех людей, на которых он подписан, поиск всех твитов для каждого из них и их слияние (с сортировкой по времени). В реляционной базе данных, такой как на рис. 1.2, это можно выполнить путем следующего запроса:

```
SELECT tweets.*, users.* FROM tweets
  JOIN users   ON tweets.sender_id    = users.id
  JOIN follows ON follows.followee_id = users.id
  WHERE follows.follower_id = current_user
```

2. Поддержка кэша для домашней ленты каждого пользователя — аналог почтового ящика твитов для каждого получателя (рис. 1.3). Когда пользователь *публикует твит*, выполняется поиск всех его подписчиков и вставка нового твита во все кэши их домашних лент. Запрос на чтение домашней ленты при этом становится малозатратным, поскольку его результат уже был заранее вычислен.

Первая версия социальной сети Twitter использовала подход 1, но система еле справлялась с нагрузкой от запросов домашних лент, вследствие чего компания переключилась на подход 2. Этот вариант работал лучше, поскольку средняя частота публикуемых твитов почти на два порядка ниже, чем частота чтения домашних лент, так что в данном случае предпочтительнее выполнять больше операций во время записи, а не во время чтения.

Однако недостатком подхода 2 является необходимость в значительном количестве дополнительных действий для публикации твита. В среднем твит выдается 75 подписчикам, поэтому 4,6 тысяч твитов в секунду означает 345 тысяч записей в секунду в кэши домашних лент. Но приведенное здесь среднее значение маскирует тот факт, что количество подписчиков на пользователя сильно варьируется, и у некоторых пользователей насчитывается более 30 миллионов подписчиков[2]. То есть один твит может привести к более чем 30 миллионов записей в домашние ленты! Выполнить их своевременно — Twitter пытается выдавать твиты подписчикам в течение пяти секунд — непростая задача.

---

[1] Термин, заимствованный из электроники, где описывает число логических вентилей, чьи входы подключаются к выходу данного вентиля. Выход должен обеспечивать ток, достаточный для работы всех подключенных входов. В системах обработки транзакций термин используется для описания числа запросов к другим сервисам, которое нужно выполнить, чтобы обслужить один входящий запрос.

[2] Этот рекорд давно побит, количество подписчиков у певицы Кэти Перри в июне 2017 года превысило 100 миллионов — *Примеч. пер.*

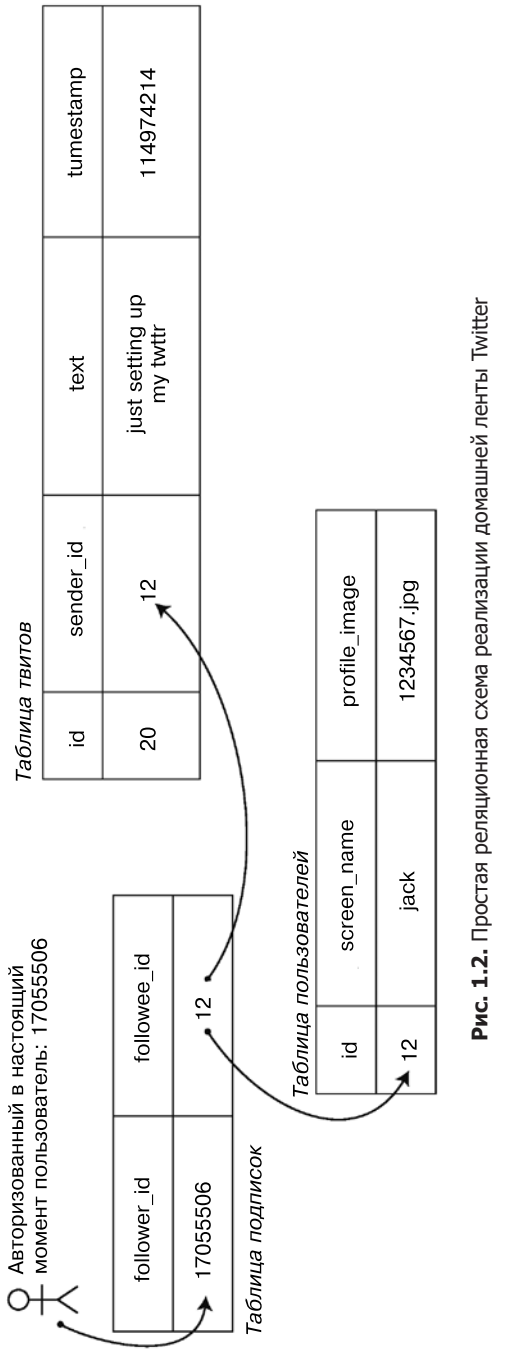


Рис. 1.2. Простая реляционная схема реализации домашней ленты Twitter

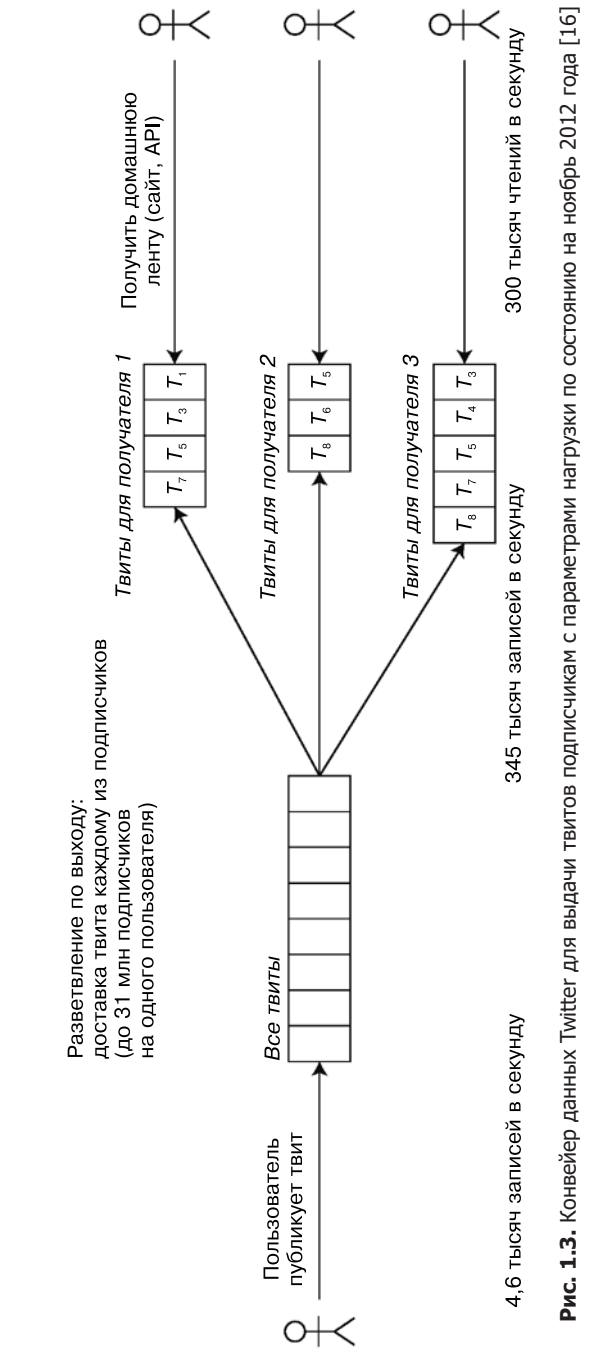


Рис. 1.3. Конвейер данных Twitter для выдачи твитов подписчикам с параметрами нагрузки по состоянию на ноябрь 2012 года [16]

В примере с сетью Twitter распределение подписчиков на одного пользователя (возможно, взвешенное с учетом частоты твитов этих пользователей) — ключевой параметр нагрузки для анализа масштабирования, ведь именно он определяет нагрузку разветвления по выходу. Характеристики ваших приложений могут очень сильно отличаться, но для рассуждений относительно их нагрузки все равно можно применять аналогичные принципы.

И последний поворот истории Twitter: реализовав ошибкоустойчивый подход 2, сеть понемногу начинает двигаться в сторону объединения обоих подходов. Большинство записей пользователей по-прежнему распространяются по домашним лентам в момент их публикации, но некое количество пользователей с очень большим количеством подписчиков (то есть знаменитости) исключены из этого процесса. Твиты от знаменитостей, на которых подписан пользователь, выбираются отдельно и сливаются с его домашней лентой при ее чтении, подобно подходу 1. Такая гибридность обеспечивает одинаково хорошую производительность во всех случаях. Мы вернемся к этому примеру в главе 12, после того как рассмотрим ряд других технических вопросов.

Описание производительности

После описания нагрузки на систему, можно выяснить, что произойдет при ее возрастании. Следует обратить внимание на два аспекта.

- Как изменится производительность системы, если увеличить параметр нагрузки при неизменных ресурсах системы (CPU, оперативная память, пропускная способность сети и т. д.)?
- Насколько нужно увеличить ресурсы при увеличении параметра нагрузки, чтобы производительность системы не изменилась?

Для ответа на оба вопроса понадобятся характеристики производительности, так что рассмотрим вкратце описание производительности системы.

В системах пакетной обработки данных, таких как Hadoop, нас обычно волнует пропускная способность — количество записей, которые мы можем обработать в секунду, или общее время, необходимое для выполнения задания на наборе данных определенного размера[1]. В онлайн-системах важнее, как правило, *время ответа* сервиса, то есть время между отправкой запроса клиентом и получением ответа.

---

[1] В идеальном мире время работы пакетного задания равно размеру набора данных, деленному на пропускную способность. На практике время работы обычно оказывается больше из-за асимметрии (данные распределяются между исполнительными процессами неравномерно), и приходится ожидать завершения самой медленной задачи.



Время ожидания и время отклика

Термины «время ожидания» (latency) и «время отклика» (response time) часто используются как синонимы, хотя это не одно и то же. Время отклика — то, что видит клиент: помимо фактического времени обработки запроса (время обслуживания, service time), оно включает задержки при передаче информации по сети и задержки сообщений в очереди. Время ожидания — длительность ожидания запросом обработки, то есть время, на протяжении которого он ожидает обслуживания [17].

Даже если повторять раз за разом один и тот же запрос, время отклика будет несколько различаться при каждой попытке. На практике в обрабатывающей множество разнообразных запросов системе время отклика способно существенно различаться. Следовательно, необходимо рассматривать время отклика не как одно число, а как *распределение* значений, характеристики которого можно определить.

На рис. 1.4 каждый серый столбец отражает запрос к сервису, а его высота показывает длительность выполнения этого запроса. Большинство запросов выполняется достаточно быстро, но встречаются и отдельные аномальные запросы со значительно большей длительностью. Возможно, медленные запросы более затратны, например, так как связаны с обработкой больших объемов данных. Но даже при сценарии, когда можно было бы думать, что обработка всех запросов будет занимать одинаковое время, в реальности длительность отличается. Дополнительное время ожидания может понадобиться из-за переключения контекста на фоновый процесс, потери сетевого пакета и повторную передачу по протоколу TCP, паузу на сборку мусора, сбой страницы, требующий чтения с диска, механические вибрации в серверной стойке [18] и по многим другим причинам.

Довольно часто смотрят именно на *среднее* время отклика. (Строго говоря, термин «среднее» не подразумевает значения, вычисленного по какой-либо конкретной формуле, но на практике под ним обычно понимается *арифметическое среднее*: при n значениях сложить их все и разделить на n.) Однако среднее значение далеко не лучшая метрика для случаев, когда нужно знать «типичное» время отклика, поскольку оно ничего не говорит о том, у какого количества пользователей фактически была такая задержка.

Обычно удобнее применять *процентили*. Если отсортировать список времен отклика по возрастанию, то *медиана* — средняя точка: например, медианное время отклика, равное 200 мс, означает, что ответы на половину запросов возвращаются менее чем через 200 мс, а половина запросов занимает более длительное время.

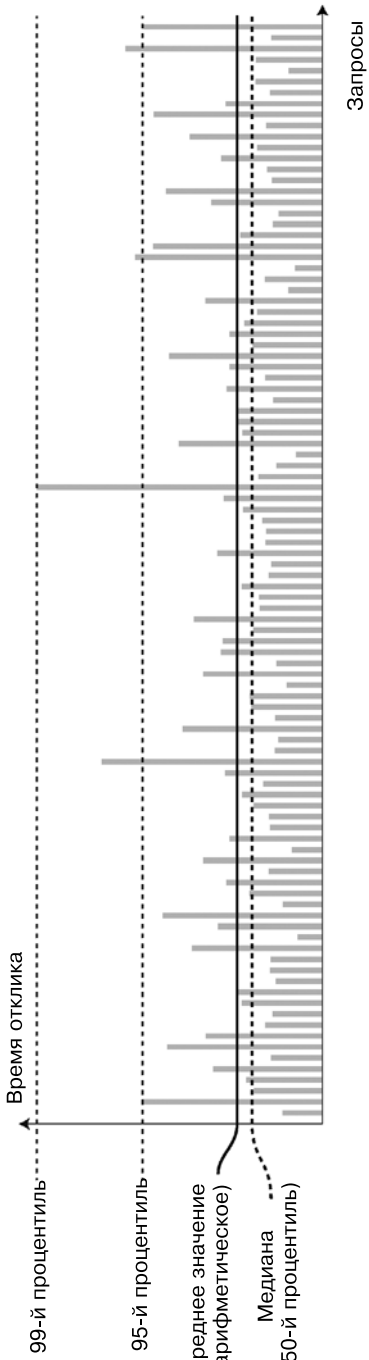


Рис. 1.4. Иллюстрирует среднее значение и процентили: время отклика для выборки из 100 запросов к сервису

Это делает медиану отличной метрикой, когда нужно узнать, сколько пользователям обычно приходится ждать: половина запросов пользователя обслуживается за время отклика меньше медианного, а оставшиеся обслуживаются более длительное время. Медиана также называется *50-м процентилем*, который иногда обозначают *p50*. Обратите внимание: медиана относится к отдельному запросу; если пользователь выполняет несколько запросов (за время сеанса или потому, что в одну страницу включено несколько запросов), вероятность выполнения хотя бы одного из них медленнее медианы значительно превышает 50 %.

Чтобы выяснить, насколько плохи аномальные значения, можно обратить внимание на более высокие процентили: часто применяются *95-й*, *99-й* и *99.9-й* (сокращенно обозначаемые *p95*, *p99* и *p999*). Это пороговые значения времени отклика, для которых 95 %, 99 % или 99,9 % запросов выполняются быстрее соответствующего порогового значения времени. Например, то, что время отклика для 95-го процентиля равно 1,5 с, означает следующее: 95 из 100 запросов занимают менее 1,5 с, а 5 из 100 занимают 1,5 с либо дольше. Данная ситуация проиллюстрирована на рис. 1.4.

Верхние процентили времени отклика, известные также под названием *«хвостовых» времен ожидания*, важны потому, что непосредственно оказывают влияние на опыт взаимодействия пользователя с сервисом. Например, Amazon описывает требования к времени отклика для внутренних сервисов в терминах 99.9х процентилей, хотя это касается лишь 1 из 1000 запросов. Дело в том, что клиенты с самыми медленными запросами зачастую именно те, у кого больше всего данных в учетных записях, поскольку они сделали много покупок, — то есть они являются самыми ценными клиентами [19]. Важно, чтобы эти клиенты оставались довольны; необходимо обеспечить быструю работу сайта именно для них: компания Amazon обнаружила, что рост времени отклика на 100 мс снижает продажи на 1 % [20], а по другим сообщениям, замедление на 1 с снижает удовлетворенность пользователей на 16 % [21, 22].

С другой стороны, оптимизация по 99.99-му процентилю (самому медленному из 10 000 запросов) считается слишком дорогостоящей и не приносящей достаточно выгоды с точки зрения целей Amazon. Снижать время отклика на очень высоких процентилях — непростая задача, поскольку на них могут оказывать влияние по независящим от вас причинам различные случайные события, а выгоды от этого минимальны.

Например, процентили часто используются в *требованиях к уровню предоставления сервиса* (service level objectives, SLO) и *соглашениях об уровне предоставления сервиса* (service level agreements, SLA) — контрактах, описывающих ожидаемые производительность и доступность сервиса. В SLA, например, может быть указано: сервис рассматривается как функционирующий нормально, если его медианное время отклика менее 200 мс, а 99-й процентиль меньше 1 с (когда время отклика больше, это равносильно неработающему сервису), причем в требованиях может быть указано, что сервис должен работать нормально не менее 99,9 % времени. Благодаря этим метрикам клиентские приложения знают, чего ожидать от сервиса,

и обеспечивают пользователям возможность потребовать возмещения в случае несоблюдения SLA.

За значительную часть времени отклика на верхних процентилях часто несут ответственность задержки сообщений в очереди. Так как сервер может обрабатывать параллельно лишь небольшое количество заданий (ограниченное, например, количеством ядер процессора), даже небольшого количества медленных запросов достаточно для задержки последующих запросов — явление, иногда называемое *блокировкой головы очереди*. Даже если последующие запросы обрабатываются сервером быстро, клиентское приложение все равно будет наблюдать низкое общее время отклика из-за времени ожидания завершения предыдущего запроса. Принимая во внимание это явление, важно измерять время отклика на стороне клиента.

При искусственной генерации нагрузки с целью тестирования масштабируемости системы клиент, генерирующий нагрузку, должен отправлять запросы независимо от времени отклика. Если клиент станет ожидать завершения предыдущего запроса перед отправкой следующего, то это будет равносильно искусственному сокращению очередей при тестировании по сравнению с реальностью, что исказит полученные результаты измерений [23].

Процентили на практике

Верхние процентили приобретают особое значение в прикладных сервисах, вызываемых неоднократно при обслуживании одного запроса конечного пользователя. Даже при выполнении вызовов параллельно запросу конечного пользователя все равно приходится ожидать завершения самого медленного из параллельных вызовов. Достаточно одного медленного вызова, чтобы замедлить весь запрос конечного пользователя, как показано на рис. 1.5. Даже если лишь небольшой процент прикладных вызовов медленные, шансы попасть на медленный вызов возрастают, если запрос конечного пользователя нуждается в том, чтобы их было много, так что больший процент запросов конечных пользователей оказывается медленными (это явление известно под названием *«усиление "хвостового" времени ожидания»* [24]).

Если нужно добавить в мониторинговые инструментальные панели ваших сервисов процентили времени отклика, необходимо эффективно организовать их регулярное вычисление. Например, можно использовать скользящее окно времени отклика запросов за последние 10 минут с вычислением каждую минуту медианы и различных процентилей по значениям из данного окна с построением графика этих метрик.

В качестве «наивной» реализации можно предложить список времен отклика для всех запросов во временном окне с сортировкой этого списка ежеминутно. Если такая операция представляется вам недостаточно эффективной, существуют алгоритмы приближенного вычисления процентилей при минимальных затратах процессорного времени и оперативной памяти, например forward decay [25], t-digest [26] и HdrHistogram [27]. Учтите, что усреднение процентилей с целью понижения временного разрешения или объединения данных с нескольких машин математически бессмысленно — правильнее будет агрегировать данные о времени отклика путем сложения гистограмм [28].

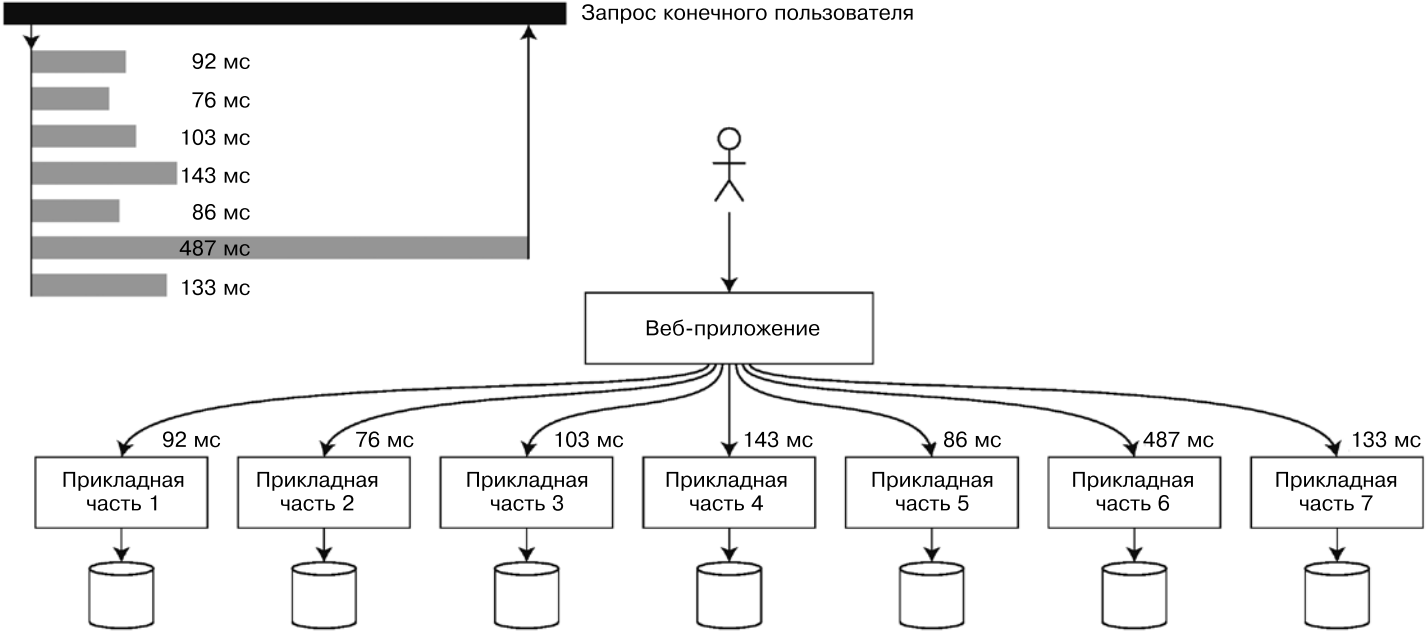


Рис. 1.5. Если для обслуживания запроса необходимо несколько обращений к прикладным сервисам, то один-единственный медленный прикладной запрос может затормозить работу всего запроса конечного пользователя

Как справиться с нагрузкой

Обсудив параметры для описания нагрузки и метрики для оценки производительности, можно всерьез перейти к изучению вопроса масштабируемости: как сохранить хорошую производительность даже в случае определенного увеличения параметров нагрузки?

Подходящая для определенного уровня нагрузки архитектура, вероятно, не сможет справиться с десятикратным ее увеличением. Если вы имеете дело с быстрорастущим сервисом, то, вероятно, вам придется пересматривать архитектуру при каждом возрастании нагрузки на порядок или даже еще чаще.

Зачастую проводят дифференциацию между *вертикальным масштабированием* — переходом на более мощную машину — и *горизонтальным масштабированием* — распределением нагрузки по нескольким меньшим машинам. Распределение нагрузки по нескольким машинам известно также под названием *архитектуры, не предусматривающей разделения ресурсов*. Системы, которые способны работать на отдельной машине, обычно проще, а высококлассные машины могут оказаться весьма недешевыми, так что при высокой рабочей нагрузке часто нельзя избежать горизонтального масштабирования. На практике хорошая архитектура обычно представляет собой прагматичную смесь этих подходов: например, может оказаться проще и дешевле использовать несколько весьма мощных компьютеров, чем много маленьких виртуальных машин.

Некоторые системы способны *адаптироваться*, то есть умеют автоматически добавлять вычислительные ресурсы при обнаружении прироста нагрузки, в то время как другие системы необходимо масштабировать вручную (человек анализирует производительность и решает, нужно ли добавить в систему дополнительные машины). Способные к адаптации системы полезны в случае непредсказуемого характера нагрузки, но масштабируемые вручную системы проще и доставляют меньше неожиданностей при эксплуатации (см. раздел 6.4).

Хотя распределение сервисов без сохранения состояния по нескольким машинам особых сложностей не представляет, преобразование информационных систем с сохранением состояния из одноузловых в распределенные может повлечь значительные сложности. Поэтому до недавнего времени считалось разумным держать базу данных на одном узле (вертикальное масштабирование) до тех пор, пока стоимость масштабирования или требования по высокой доступности не заставят сделать ее распределенной.

По мере усовершенствования инструментов и абстракций для распределенных систем это мнение подвергается изменениям, по крайней мере для отдельных видов приложений. Вполне возможно, что распределенные информационные системы в будущем станут «золотым стандартом» даже для сценариев применения, в которых не идет речь об обработке больших объемов данных или трафика. В оставшейся части книги мы рассмотрим многие виды распределенных информационных систем

и обсудим их с точки зрения не только масштабирования, но и удобства использования и сопровождения.

Архитектура крупномасштабных систем обычно очень сильно зависит от приложения — не существует такой вещи, как одна масштабируемая архитектура на все случаи жизни (на сленге именуемая *волшебным масштабирующим соусом*). Проблема может заключаться в количестве чтений, количестве записей, объеме хранимых данных, их сложности, требованиях к времени отклика, паттернах доступа или (зачастую) какой-либо смеси всего перечисленного, а также во многом другом.

Например, система, рассчитанная на обработку 100 000 з/с по 1 Кбайт каждый, выглядит совсем не так, как система, рассчитанная на обработку 3 з/мин. по 2 Гбайт каждый — хотя пропускная способность обеих систем в смысле объема данных одинакова.

Хорошая масштабируемая для конкретного приложения архитектура базируется на допущениях о том, какие операции будут выполняться часто, а какие — редко, то есть на параметрах нагрузки. Если эти допущения окажутся неверными, то работа архитекторов по масштабированию окажется в лучшем случае напрасной, а в худшем — приведет к обратным результатам. На ранних стадиях обычно важнее быстрая работа существующих возможностей в опытном образце системы или непроверенном программном продукте, чем его масштабируемость под гипотетическую будущую нагрузку.

Несмотря на зависимость от конкретного приложения, масштабируемые архитектуры обычно создаются на основе универсальных блоков, организованных по хорошо известным паттернам. В нашей книге мы будем обсуждать эти блоки и паттерны.

1.4. Удобство сопровождения

Широко известно, что стоимость программного обеспечения состоит по большей части из затрат не на изначальную разработку, а на текущее сопровождение — исправление ошибок, поддержание работоспособности его подсистем, расследование отказов, адаптацию к новым платформам, модификацию под новые сценарии использования, возврат технического долга[1] и добавление новых возможностей.

Однако, к сожалению, многие работающие над программными системами люди ненавидят сопровождение так называемых *унаследованных* систем — вероятно, потому, что приходится исправлять чужие ошибки либо работать с устаревшими платформами или системами, которые заставили делать то, для чего они никогда не были предназначены. Каждая унаследованная система неприятна по-своему, поэтому столь сложно дать какие-либо общие рекомендации о том, что с ними делать.

[1] См.: https://ru.wikipedia.org/wiki/Технический_долг. — *Примеч. пер.*

Однако можно и нужно проектировать программное обеспечение так, чтобы максимально минимизировать «головную боль» при сопровождении, а следовательно, избегать создания унаследованных систем своими руками. Отталкиваясь от этого соображения, мы уделим особое внимание трем принципам проектирования программных систем.

- *Удобство эксплуатации*. Облегчает обслуживающему персоналу поддержание беспрепятственной работы системы.
- *Простота*. Облегчает понимание системы новыми инженерами путем максимально возможного ее упрощения. (Обратите внимание: это не то же самое, что простота пользовательского интерфейса.)
- *Возможность развития*. Упрощает разработчикам внесение в будущем изменений в систему, адаптацию ее для непредвиденных сценариев использования при смене требований. Известна под названиями *«расширяемость»* (extensibility), *«модифицируемость»* (modifiability) и *«пластичность»* (plasticity).

Как и в случае надежности и масштабируемости, не существует простых решений для достижения этих целей. Следует просто иметь в виду удобство эксплуатации, простоту и возможность развития при проектировании систем.

Удобство эксплуатации

Считается, что «хороший обслуживающий персонал часто может обойти ограничения плохого (или несовершенного) программного обеспечения, но хорошее программное обеспечение не способно надежно работать под управлением плохих операторов» [12]. Хотя некоторые аспекты эксплуатации можно и нужно автоматизировать, выполнение этой автоматизации и проверка правильности ее работы все равно остается прежде всего задачей обслуживающего персонала.

Данный персонал жизненно важен для бесперебойной работы программной системы. Хорошая команда операторов обычно отвечает за пункты, перечисленные ниже, и не только [29]:

- мониторинг состояния системы и восстановление сервиса в случае его ухудшения;
- выяснение причин проблем, например, отказов системы или снижения производительности;
- поддержание актуальности программного обеспечения и платформ, включая исправления безопасности;
- отслеживание влияния различных систем друг на друга во избежание проблемных изменений до того, как они нанесут ущерб;
- предупреждение и решение возможных проблем до их возникновения (например, планирование производительности);

- введение в эксплуатацию рекомендуемых практик и инструментов для развертывания, управления конфигурацией и т. п.;
- выполнение сложных работ по сопровождению, например переноса приложения с одной платформы на другую;
- поддержание безопасности системы при изменениях в конфигурации;
- формирование процессов, которые бы обеспечили прогнозируемость операций и стабильность операционной среды;
- сохранение знаний организации о системе, несмотря на уход старых сотрудников и приход новых.

Удобство эксплуатации означает облегчение выполнения стандартных задач, благодаря чему обслуживающий персонал может сосредоточить усилия на чем-то более важном. Информационные системы способны делать многое для облегчения выполнения стандартных задач, в том числе:

- обеспечивают хороший мониторинг и предоставляют информацию о поведении системы и происходящем внутри нее во время работы;
- обеспечивают хорошую поддержку автоматизации и интеграции со стандартными утилитами;
- позволяют не зависеть от отдельных машин (благодаря чему можно отключать некоторые из них для технического обслуживания при сохранении бесперебойной работы системы в целом);
- предоставляют качественную документацию и понятную операционную модель («если я выполню действие X, то в результате произойдет действие Y»);
- обеспечивают разумное поведение по умолчанию, но вместе с тем и возможности для администраторов при необходимости переопределять настройки по умолчанию;
- запускают самовосстановление по мере возможности, но вместе с тем и позволяют администраторам вручную управлять состоянием системы при необходимости;
- демонстрируют предсказуемое поведение, минимизируя неожиданности.

Простота: регулируем сложность

Код небольших программных проектов может быть восхитительно простым и выразительным, но по мере роста проекта способен стать очень сложным и трудным для понимания. Подобная сложность замедляет работу над системой, еще более увеличивая стоимость сопровождения. Увязнувший в сложности проект иногда описывают как *большой ком грязи* [30].

Существуют различные возможные симптомы излишней сложности: скачкообразный рост пространства состояний, тесное сцепление модулей, запутанные

зависимости, несогласованные наименования и терминология, «костыли» для решения проблем с производительностью, выделение частных случаев для обхода проблем и многое другое. В литературе об этом было сказано немало [31–33].

Когда сложность достигает уровня, сильно затрудняющего сопровождение, зачастую происходит превышение бюджетов и срыв сроков. В сложном программном обеспечении увеличивается и шанс внесения ошибок при выполнении изменений: когда разработчикам сложнее понимать систему и обсуждать ее, гораздо проще упустить какие-либо скрытые допущения, непреднамеренные последствия и неожиданные взаимодействия. И напротив, снижение сложности резко повышает удобство сопровождения ПО, а следовательно, простота должна быть основной целью создаваемых систем.

Упрощение системы не обязательно означает сокращение ее функциональности. Оно может означать также исключение *побочной* сложности. Мозли и Маркс [32] определяют сложность как побочную, если она возникает вследствие конкретной реализации, а не является неотъемлемой частью решаемой программным обеспечением задачи (с точки зрения пользователей).

Один из лучших инструментов для исключения побочной сложности — *абстракция*. Хорошая абстракция позволяет скрыть бо́льшую часть подробностей реализации за аккуратным и понятным фасадом. Хорошую абстракцию можно также задействовать для широкого диапазона различных приложений. Такое многократное применение не только эффективнее реализации заново одного и тоже же несколько раз, но и приводит к более качественному ПО, по мере усовершенствования используемого всеми приложениями абстрактного компонента.

Например, высокоуровневые языки программирования — абстракции, скрывающие машинный код, регистры CPU и системные вызовы. SQL — тоже абстракция, скрывающая сложные структуры данных, находящиеся на диске и в оперативной памяти, конкурентные запросы от других клиентов и возникающие после фатальных сбоев несогласованности. Конечно, при создании продукта с помощью языка программирования высокого уровня мы по-прежнему используем машинный код, просто не задействуем его *напрямую*, поскольку абстракция языка программирования позволяет не задумываться об этом.

Однако найти хорошую абстракцию очень непросто. В сфере распределенных систем, несмотря на наличие множества хороших алгоритмов, далеко не так ясно, как объединить их в абстракции, которые бы дали возможность сохранить приемлемый уровень сложности системы.

В этой книге мы будем отслеживать хорошие абстракции, позволяющие выделить части большой системы в четко очерченные компоненты, допускающие повторное использование.

Возможность развития: облегчаем внесение изменений

Вероятность того, что ваши системные требования навсегда останутся неизменными, стремится к нулю. Гораздо вероятнее их постоянное преобразование: будут открываться новые факты, возникать непредвиденные сценарии применения, меняться коммерческие приоритеты, пользователи будут требовать новые возможности, новые платформы — заменять старые, станут меняться правовые и нормативные требования, рост системы потребует архитектурных изменений и т. д.

В терминах организационных процессов рабочие паттерны *Agile* обеспечивают инфраструктуру адаптации к изменениям. Сообщество создателей Agile разработало также технические инструменты и паттерны, полезные при проектировании программного обеспечения в часто меняющейся среде, например при разработке через тестирование (test-driven development, TDD) и рефакторинге.

Большинство обсуждений этих методов Agile ограничивается довольно небольшими, локальными масштабами (пара файлов исходного кода в одном приложении). В книге мы займемся поиском способов ускорения адаптации на уровне больших информационных систем, вероятно состоящих из нескольких различных приложений или сервисов с различными характеристиками. Например, как бы вы провели рефакторинг архитектуры сети Twitter для компоновки домашних лент (см. подраздел «Описание нагрузки» раздела 1.3) с подхода 1 на подход 2?

Степень удобства модификации информационной системы и адаптации ее к меняющимся требованиям тесно связана с ее простотой и абстракциями: простые и понятные системы обычно легче менять, чем сложные. Но в силу исключительной важности этого понятия мы будем использовать другой термин для быстроты адаптации на уровне информационных систем — *возможность развития* (evolvability) [34].

1.5. Резюме

В настоящей главе мы рассмотрели некоторые основополагающие подходы к работе с высоконагруженными данными приложениями. Мы будем руководствоваться этими принципами далее в книге, где перейдем к техническим подробностям.

Чтобы приносить пользу, приложение должно соответствовать различным требованиям. Выделяют *функциональные требования* (что приложение должно делать, например обеспечивать возможность хранения, извлечения, поиска и обработки информации различными способами) и некоторые *нефункциональные требования* (такие общие характеристики, как безопасность, надежность, соответствие нормативным документам, масштабируемость, совместимость и удобство сопровождения).

В этой главе мы подробно обсудили надежность, масштабируемость и удобство сопровождения.

Надежность означает обеспечение правильной работы системы даже в случае сбоев. Они могут происходить в аппаратном обеспечении (обычно случайные и невзаимосвязанные), ПО (ошибки обычно носят системный характер и слабо контролируемы) и привноситься операторами (неминуемо допускающими ошибки время от времени). Методы обеспечения устойчивости к сбоям позволяют скрывать некоторые виды сбоев от конечного пользователя.

Масштабируемость означает наличие стратегий поддержания хорошей производительности даже в случае роста нагрузки. Для обсуждения масштабируемости необходимы прежде всего способы количественного описания нагрузки и производительности. Мы вкратце рассмотрели домашние ленты социальной сети Twitter в качестве примера описания нагрузки и процентили времени отклика как способ оценки производительности. В масштабируемой системе есть возможность добавлять вычислительные мощности в целях сохранения надежной работы системы под высокой нагрузкой.

Удобство сопровождения многолико, но, по существу, подразумевается облегчение жизни командам разработчиков и операторов, работающим с системой. Помочь снизить сложность и упростить модификацию системы и ее адаптацию к новым сценариям использования могут хорошие абстракции. Удобство эксплуатации означает хороший мониторинг состояния системы и эффективные способы управления им.

К сожалению, не существует легкого пути для обеспечения надежности, масштабируемости и удобства сопровождения приложений. Однако есть определенные паттерны и методики, которые снова и снова встречаются в различных видах приложений. В нескольких следующих главах мы рассмотрим отдельные примеры информационных систем и проанализируем, что они делают для достижения этих целей.

Далее в нашей книге, в части III, мы рассмотрим паттерны для систем, состоящих из нескольких совместно работающих компонентов, наподобие показанной на рис. 1.1.

1.6. Библиография

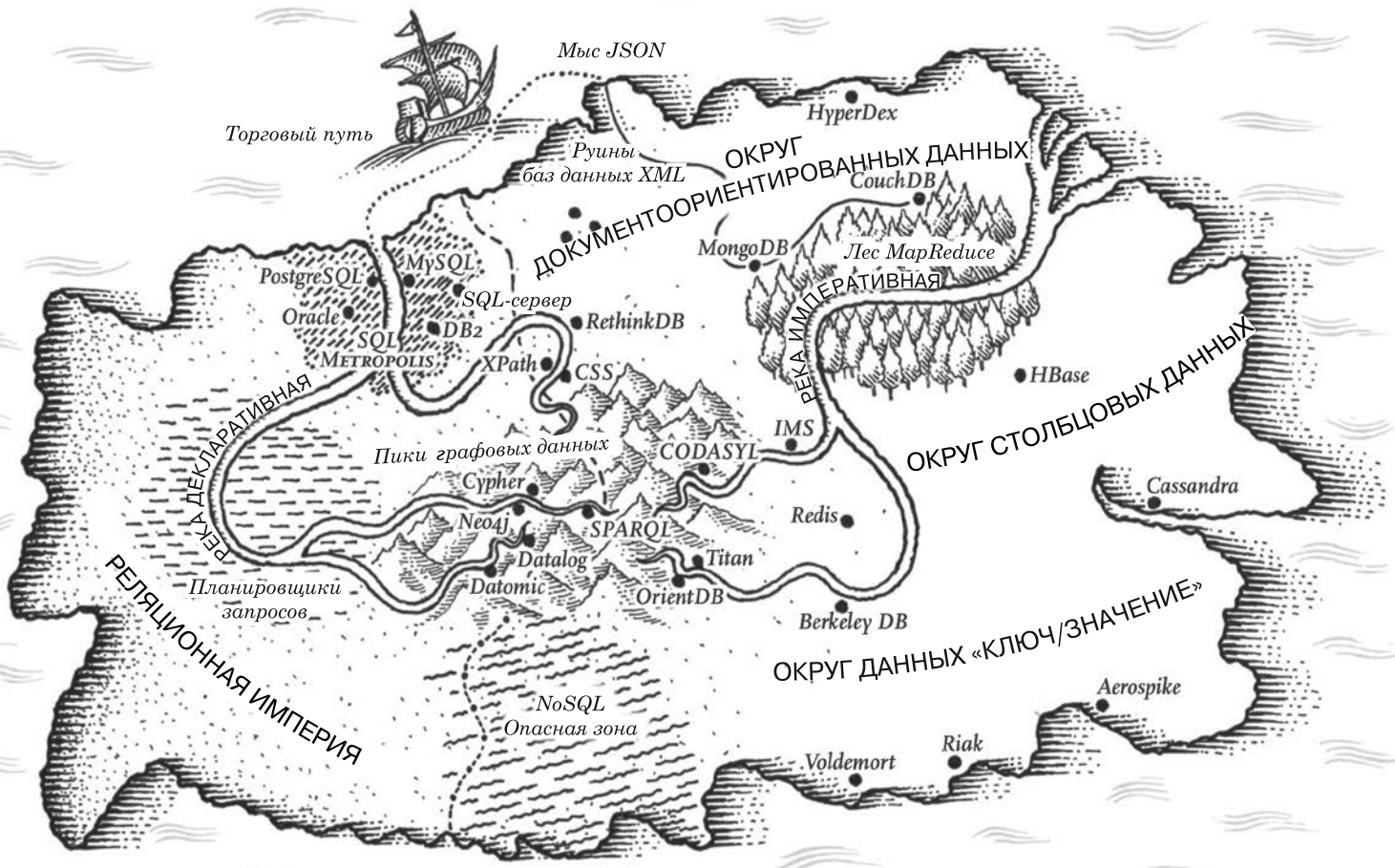
1. *Stonebraker M., Çetintemel U.* 'One Size Fits All': An Idea Whose Time Has Come and Gone // 21st International Conference on Data Engineering (ICDE), April 2005 [Электронный ресурс]. — Режим доступа: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.68.9136&rep=rep1&type=pdf.
2. *Heimerdinger W. L., Weinstock C. B.* A Conceptual Framework for System Fault Tolerance // Technical Report CMU/SEI-92-TR-033, Software Engineering

Institute, Carnegie Mellon University, October 1992 [Электронный ресурс]. — Режим доступа: https://www.sei.cmu.edu/reports/92tr033.pdf.

3. *Yuan D., Luo Y., Zhuang X., et al.* Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems // 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), October 2014 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf.
4. *Izrailevsky Y., Tseitlin A.* The Netflix Simian Army. July 19, 2011 [Электронный ресурс]. — Режим доступа: https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116.
5. *Ford D., Labelle F., Popovici F. I., et al.* Availability in Globally Distributed Storage Systems // 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), October 2010 [Электронный ресурс]. — Режим доступа: https://static.googleusercontent.com/media/research.google.com/ru//pubs/archive/36737.pdf.
6. *Beach B.* Hard Drive Reliability Update Sep 2014. — September 23, 2014 [Электронный ресурс]. — Режим доступа: https://www.backblaze.com/blog/hard-drive-reliability-update-september-2014/.
7. *Voss L.* AWS: The Good, the Bad and the Ugly. December 18, 2012 [Электронный ресурс]. — Режим доступа: https://web.archive.org/web/20160406004621/http://blog.awe.sm/2012/12/18/aws-the-good-the-bad-and-the-ugly/.
8. *Gunawi H. S., Hao M., Leesatapornwongsa T., et al.* What Bugs Live in the Cloud? // 5th ACM Symposium on Cloud Computing (SoCC), November 2014 [Электронный ресурс]. — Режим доступа: http://ucare.cs.uchicago.edu/pdf/socc14-cbs.pdf.
9. *Minar N.* Leap Second Crashes Half the Internet. July 3, 2012 [Электронный ресурс]. — Режим доступа: http://www.somebits.com/weblog/tech/bad/leap-second-2012.html.
10. Amazon Web Services: Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. — April 29, 2011 [Электронный ресурс]. — Режим доступа: https://aws.amazon.com/ru/message/65648/.
11. *Cook R. I.* How Complex Systems Fail // Cognitive Technologies Laboratory, April 2000 [Электронный ресурс]. — Режим доступа: http://web.mit.edu/2.75/resources/random/How%20Complex%20Systems%20Fail.pdf.
12. *Kreps J.* Getting Real About Distributed System Reliability. March 19, 2012 [Электронный ресурс]. — Режим доступа: http://blog.empathybox.com/post/19574936361/getting-real-about-distributed-system-reliability.
13. *Oppenheimer D., Ganapathi A., Patterson D. A.* Why Do Internet Services Fail, and What Can Be Done About It? // 4th USENIX Symposium on Internet Technologies and Systems (USITS), March 2003 [Электронный ресурс]. — Режим доступа: http://static.usenix.org/legacy/events/usits03/tech/full_papers/oppenheimer/oppenheimer.pdf.

14. *Marz N.* Principles of Software Engineering, Part 1. April 2, 2013 [Электронный ресурс]. — Режим доступа: http://nathanmarz.com/blog/principles-of-software-engineering-part-1.html.
15. *Jurewitz M.* The Human Impact of Bugs. — March 15, 2013 [Электронный ресурс]. — Режим доступа: http://jury.me/blog/2013/3/14/the-human-impact-of-bugs.
16. *Krikorian R.* Timelines at Scale // QCon San Francisco, November 2012 [Электронный ресурс]. — Режим доступа: https://www.infoq.com/presentations/Twitter-Timeline-Scalability.
17. *Fowler M.* Patterns of Enterprise Application Architecture. — Addison Wesley, 2002.
18. *Sommers K.* After all that run around, what caused 500ms disk latency even when we replaced physical server? November 13, 2014 [Электронный ресурс]. — Режим доступа: https://twitter.com/kellabyte/status/532930540777635840.
19. *DeCandia G., Hastorun D., Jampani M., et al.* Dynamo: Amazon's Highly Available Key-Value Store // 21st ACM Symposium on Operating Systems Principles (SOSP), October 2007 [Электронный ресурс]. — Режим доступа: http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf.
20. *Linden G.* Make Data Useful // slides from presentation at Stanford University Data Mining class (CS345), December 2006 [Электронный ресурс]. — Режим доступа: http://glinden.blogspot.com.by/2006/12/slides-from-my-talk-at-stanford.html.
21. *Everts T.* The Real Cost of Slow Time vs Downtime. November 12, 2014 [Электронный ресурс]. — Режим доступа: https://blog.radware.com/applicationdelivery/wpo/2014/11/real-cost-slow-time-vs-downtime-slides/.
22. *Brutlag J.* Speed Matters for Google Web Search. June 22, 2009 [Электронный ресурс]. — Режим доступа: https://research.googleblog.com/2009/06/speed-matters.html.
23. *Treat T.* Everything You Know About Latency Is Wrong. December 12, 2015 [Электронный ресурс]. — Режим доступа: http://bravenewgeek.com/everything-you-know-about-latency-is-wrong/.
24. *Dean J., Barroso L. A.* The Tail at Scale // Communications of the ACM, volume 56, number 2, pages 74–80, February 2013 [Электронный ресурс]. — Режим доступа: https://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/abstract.
25. *Cormode G., Shkapenyuk V., Srivastava D., Xu B.* Forward Decay: A Practical Time Decay Model for Streaming Systems // 25th IEEE International Conference on Data Engineering (ICDE), March 2009 [Электронный ресурс]. — Режим доступа: http://dimacs.rutgers.edu/~graham/pubs/papers/fwddecay.pdf.
26. *Dunning T., Ertl O.* Computing Extremely Accurate Quantiles Using t-Digests. March 2014 [Электронный ресурс]. — Режим доступа: https://github.com/tdunning/t-digest.
27. *Tene G.* HdrHistogram [Электронный ресурс]. — Режим доступа: http://www.hdrhistogram.org/.

28. *Schwartz B.* Why Percentiles Don't Work the Way You Think. December 7, 2015 [Электронный ресурс]. — Режим доступа: https://www.vividcortex.com/blog/why-percentiles-dont-work-the-way-you-think.
29. *Hamilton J.* On Designing and Deploying Internet-Scale Services // 21st Large Installation System Administration Conference (LISA), November 2007 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/legacy/events/lisa07/tech/full_papers/hamilton/hamilton.pdf.
30. *Foote B., Yoder J.* Big Ball of Mud // 4th Conference on Pattern Languages of Programs (PLoP), September 1997 [Электронный ресурс]. — Режим доступа: http://www.laputan.org/pub/foote/mud.pdf.
31. *Brooks F. P.* No Silver Bullet — Essence and Accident in Software Engineering // The Mythical Man-Month, Anniversary edition, Addison-Wesley, 1995.
32. *Moseley B., Marks P.* Out of the Tar Pit // BCS Software Practice Advancement (SPA), 2006 [Электронный ресурс]. — Режим доступа: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.93.8928.
33. *Hickey R.* Simple Made Easy // Strange Loop, September 2011 [Электронный ресурс]. — Режим доступа: https://www.infoq.com/presentations/Simple-Made-Easy.
34. *Breivold H. P., Crnkovic I., Eriksson P. J.* Analyzing Software Evolvability // 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC), July 2008 [Электронный ресурс]. — Режим доступа: http://www.mrtc.mdh.se/publications/1478.pdf.


Мыс JSON
Торговый путь
HyperDex
Руины баз данных XML
ОКРУГ ДОКУМЕНТООРИЕНТИРОВАННЫХ ДАННЫХ
CouchDB
MongoDB
Лес MapReduce
РЕКА ИМПЕРАТИВНАЯ
PostgreSQL
MySQL
SQL-сервер
Oracle
DB2
RethinkDB
SQL
METROPOLIS
XPath
CSS
HBase
РЕКА ДЕКЛАРАТИВНАЯ
IMS
Пики графовых данных
CODASYL
ОКРУГ СТОЛБЦОВЫХ ДАННЫХ
Cypher
Cassandra
Redis
Neo4j
SPARQL
Datalog
Titan
Планировщики запросов
Datomic
OrientDB
Berkeley DB
РЕЛЯЦИОННАЯ ИМПЕРИЯ
ОКРУГ ДАННЫХ «КЛЮЧ/ЗНАЧЕНИЕ»
NoSQL
Опасная зона
Aerospike
Riak
Voldemort

2 Модели данных и языки запросов

Границы моего языка — это границы моего мира.

Людвиг Витгенштейн.
Логико-философский трактат (1922)

Модели данных — вероятно, важнейшая часть разработки программного обеспечения в силу оказываемого ими глубочайшего воздействия не только на процесс разработки, но и на наше *представление о решаемой проблеме*.

Большинство приложений создаются путем наслоения одной модели данных поверх другой. Ключевой вопрос для каждого слоя: как его *представить* на языке непосредственно прилегающего к нему более низкого слоя? Можно привести следующие примеры.

1. Разработчик приложений смотрит на окружающий мир (с людьми, фирмами, товарами, действиями, денежными потоками, датчиками и т. п.) и моделирует его на языке объектов или структур данных, а также API для манипуляции этими структурами данных. Такие структуры часто отражают специфику конкретного приложения.
2. При необходимости сохранить эти структуры их выражают в виде универсальной модели данных, например документов в формате JSON или XML, графовой модели или таблиц в реляционной базе данных.
3. Разработчики, создающие программное обеспечение для БД, выбирают для себя способ представления этих JSON/XML/реляционных/графовых данных в виде байтов памяти/дисковой памяти/трафика в сети. Благодаря этому появляется возможность отправлять запросы к данным, производить поиск по ним, выполнять с ними различные манипуляции и обрабатывать.

4. На еще более низких уровнях инженеры аппаратного обеспечения занимаются тем, что находят способ выразить байты в терминах электрического тока, световых импульсов, магнитных полей и т. п.

В сложном приложении может быть много промежуточных уровней (например, API, создаваемые поверх других API), но исходная идея остается неизменной: каждый слой инкапсулирует сложность нижележащих слоев с помощью новой модели данных. Благодаря таким абстракциям становится возможной совместная эффективная работа различных групп людей — например, специалистов компании-производителя СУБД и использующих эту базу данных разработчиков приложений.

Существует множество различных видов моделей данных, причем каждая из них воплощает какие-либо допущения относительно ее предполагаемого использования. Одни сценарии применения поддерживаются, а другие — нет; какие-то операции выполняются быстро, а какие-то — медленно; некоторые преобразования данных можно произвести легко и естественно, а некоторые выглядят трудновыполнимыми.

Для освоения одной-единственной модели данных придется потратить немало усилий (задумайтесь, сколько уже написано книг по реляционному моделированию данных). Создание ПО — непростая задача даже при работе с одной моделью данных и без углубления в вопросы ее внутреннего функционирования. Но поскольку от нее так сильно зависит функциональность основанного на ней программного обеспечения, важно выбрать подходящую для приложения модель.

В текущей главе мы рассмотрим группу универсальных моделей, ориентированных на хранение данных и выполнение запросов (пункт 2 в вышеприведенном списке). В частности, сравним реляционную модель, документоориентированную модель и несколько моделей данных на основе графов. Кроме того, рассмотрим несколько языков запросов и сравним их сценарии использования. В главе 3 мы обсудим функционирование подсистем хранения: как эти модели данных реализованы на самом деле (пункт 3 в нашем списке).

2.1. Реляционная модель в сравнении с документоориентированной моделью

Наиболее известной моделью данных на сегодняшний день, вероятно, является модель данных SQL, основанная на предложенной в 1970 году Эдгаром Коддом [1] реляционной модели: данные организованы в *отношения* (именуемые в SQL *таблицами*), где каждое отношение представляет собой неупорядоченный набор *кортежей* (*строк* в SQL).

Реляционная модель была теоретической конструкцией, и многие сомневались, возможна ли ее эффективная реализация. Однако к середине 1980-х годов реляционные системы управления базами данных (relational database management system, RDBMS) и SQL стали оптимальными инструментами для большинства тех, кому нужно было хранить более или менее структурированные данные и выполнять

к ним запросы. Господство реляционных БД длилось около 25–30 лет — целая вечность по масштабам истории вычислительной техники.

Истоки реляционных баз данных лежат в *обработке коммерческих данных* (business data processing), выполнявшейся в 1960-х и 1970-х годах на больших ЭВМ. С сегодняшней точки зрения их сценарии использования представляются довольно рутинными: обычно это была *обработка транзакций* (ввод транзакций, связанных с продажами или банковской деятельностью, бронирование авиабилетов, складирование товаров) и *пакетная обработка* (выставление счетов покупателям, платежные ведомости, отчетность).

С течением времени появилось немало других подходов к хранению данных и выполнению к ним запросов. В 1970-х и начале 1980-х годов основными альтернативами реляционной были *сетевая* (network model) и *иерархическая* модели, но реляционная все равно побеждала. Объектные базы данных появились в конце 1980-х — начале 1990-х годов и снова вышли из моды. В начале 2000-х появились базы данных XML, но нашли только узкое применение. Каждый из соперников реляционных БД наделал в свое время много шума, но ненадолго [2].

По мере значительного повышения мощности компьютеров и соединения их в се́ти цели их использования стали все более разнообразными. И что примечательно, применение реляционных БД распространилось на широкое множество сценариев далеко за пределами первоначальной обработки коммерческих данных. Работа большей части Интернета до сих пор обеспечивается реляционными БД: онлайн-публикации, дискуссионные форумы, социальные сети, интернет-магазины, игры, предоставляемые как сервис (по модели SaaS), офисные приложения и многое другое.

Рождение NoSQL

Сейчас, в 2010-х, *NoSQL* — самая недавняя попытка свергнуть господство реляционной модели. Название NoSQL — неудачное, ведь на самом деле ничего не говорит об используемой технологии, изначально оно было предложено в 2009 году просто в качестве броского хештега в Twitter для семинара по распределенным нереляционным БД с открытым исходным кодом [3]. Тем не менее термин задел за живое и быстро распространился в среде создателей интернет-стартапов и не только их. С хештегом #NoSQL сейчас связано несколько интересных систем баз данных, и его задним числом интерпретировали как *«Не Только SQL»* (Not Only SQL) [4].

Существует несколько основных причин широкого внедрения баз данных NoSQL, включая:

- потребность в бóльших возможностях масштабирования, чем у реляционных БД, включая обработку очень больших наборов данных или очень большую пропускную способность по записи;
- предпочтение свободного программного обеспечения вместо коммерческих продуктов;

- специализированные запросные операции, плохо поддерживаемые реляционной моделью;
- разочарование ограниченностью реляционных схем и стремление к более динамичным и выразительным моделям данных [5].

У каждого приложения свои требования, и оптимальная технология может различаться в зависимости от сценария использования. А потому вероятно, что в ближайшем будущем реляционные БД будут продолжать задействовать наряду с множеством разнообразных нереляционных баз данных — об этой идее иногда говорят как о *применении нескольких систем хранения данных в одном приложении* (polyglot persistence) [3].

Объектно-реляционное несоответствие

Большая часть разработки приложений сегодня выполняется на объектно-ориентированных языках программирования, что привело к всеобщей критике модели данных SQL: при хранении данных в реляционных таблицах необходим неуклюжий промежуточный слой между объектами кода приложения и моделью таблиц, строк и столбцов БД. Эту расстыковку моделей иногда называют *рассогласованием* (impedance mismatch)[1].

Фреймворки объектно-реляционного отображения (ORM), такие как ActiveRecord и Hibernate, снижают количество шаблонного кода, необходимого для слоя трансляции, но не в состоянии полностью скрыть различия между двумя моделями.

Например, рис. 2.1 иллюстрирует возможность выражать резюме (профиль в социальной сети LinkedIn) на языке реляционной схемы. Профиль в целом определяется уникальным идентификатором, `user_id`. Такие поля, как `first_name` и `last_name`, встречаются ровно один раз для одного пользователя, так что их можно сделать столбцами в таблице `users`. Однако у большинства людей за время карьеры бывает больше одной работы (должности), а также могут меняться периоды обучения и число элементов контактной информации. Между пользователем и этими элементами существует связь «один-ко-многим», которое можно представить несколькими способами.

- В обычной SQL-модели (до SQL:1999) в случае наиболее распространенного нормализованного представления должности образование и контактная информация помещаются в отдельные таблицы, со ссылкой на таблицу `users` в виде внешнего ключа, как показано на рис. 2.1.

---

[1] Термин, позаимствованный из электроники. У каждой электрической цепи есть определенный импеданс (сопротивление переменному току) на ее входах и выходах. При соединении выхода одной цепи с входом другой передаваемая через это соединение мощность максимальна в случае совпадения импедансов двух цепей. Рассогласование импедансов может привести к отражению сигналов и другим проблемам.

- В более поздних версиях SQL-стандарта добавлена поддержка для структурированных типов данных и данных в формате XML. Таким образом, многоэлементные данные можно сохранять в отдельной строке с возможностью отправлять к ним запросы и совершать индексации по ним. Эти возможности поддерживаются в разной степени БД Oracle, IBM DB2, MS SQL Server и PostgreSQL [6, 7]. Тип данных JSON также поддерживается в разной степени несколькими БД, включая IBM DB2, MySQL и PostgreSQL [8].
- Третий возможный вариант — кодирование должностей, образования и контактной информации в виде JSON- или XML-документа, сохранение его в текстовом столбце в базе данных с интерпретацией приложением его структуры и содержимого. Здесь отсутствует возможность использования БД для выполнения запросов к содержимому этого кодированного столбца.

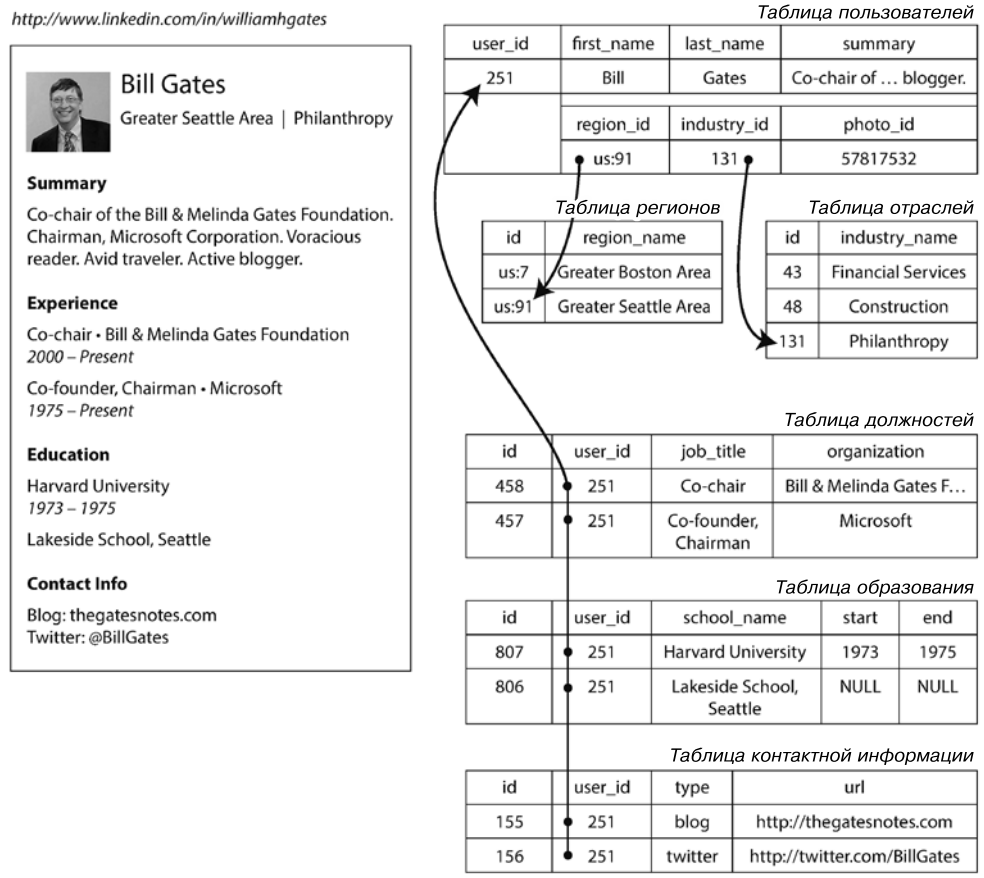


Рис. 2.1. Представление профиля LinkedIn с помощью реляционной схемы. Фото Билла Гейтса любезно предоставлено «Вики-складом», фотограф — Рикардо Стикерт, информагентство «Аженсия Бразил»

Для структур данных типа резюме, обычно являющихся самостоятельным *документом*, вполне подойдет представление в формате JSON (пример 2.1). Преимущество данного формата в том, что он намного проще, чем XML. Эту модель данных поддерживают такие документоориентированные БД, как MongoDB [9], RethinkDB [10], CouchDB [11] и Espresso [12].

Пример 2.1. Представление профиля LinkedIn в виде JSON-документа

```
   {
  "user_id":     251,
  "first_name":  "Bill",
  "last_name":   "Gates",
  "summary":     "Co-chair of the Bill & Melinda Gates... Active blogger.",
  "region_id":   "us:91",
  "industry_id": 131,
  "photo_url":   "/p/7/000/253/05b/308dd6e.jpg",
  "positions": [
    {"job_title": "Co-chair", "organization": "Bill & Melinda Gates Foundation"},
    {"job_title": "Co-founder, Chairman", "organization": "Microsoft"}
  ],
  "education": [
    {"school_name": "Harvard University",       "start": 1973, "end": 1975},
    {"school_name": "Lakeside School, Seattle", "start": null, "end": null}
  ],
  "contact_info": {
    "blog":    "http://thegatesnotes.com",
    "twitter": "http://twitter.com/BillGates"
  }
}
```

Некоторые разработчики считают, что модель JSON снижает рассогласование между кодом приложения и слоем хранения. Однако, как мы увидим в главе 4, у JSON как формата кодирования данных тоже есть проблемы. Чаще всего отсутствие схемы рассматривается как преимущество, мы обсудим это в пункте «Гибкость схемы в документной модели» подраздела «Реляционные и документоориентированные базы данных сегодня» текущего раздела.

У JSON-представления — лучшая *локальность*, чем у многотабличной схемы на рис. 2.1. Для извлечения профиля в реляционном примере необходимо выполнить несколько запросов (по запросу к каждой таблице по `user_id`) или запутанное многостороннее соединение таблицы `users` с подчиненными ей таблицами. В JSON-представлении же вся нужная информация находится в одном месте, и одного запроса вполне достаточно.

Связи «один-ко-многим» профиля пользователя с его должностями, местами обучения и контактной информацией означает древовидную структуру данных, а JSON-представление делает эту структуру явной (рис. 2.2).

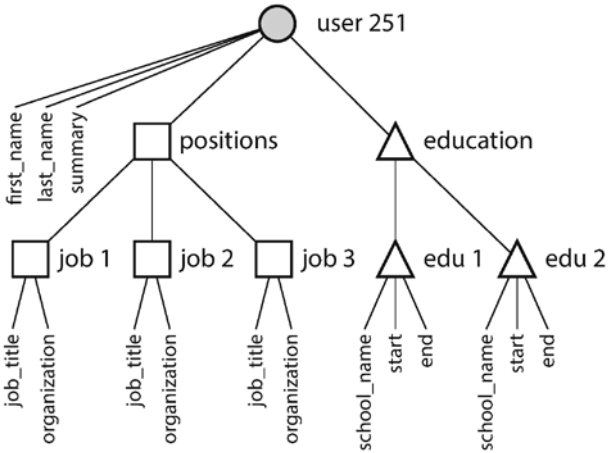


Рис. 2.2. Связи «один-ко-многим» формируют древовидную структуру

Связи «многие-к-одному» и «многие-ко-многим»

В примере 2.1 элементы `region_id` и `industry_id` представляют собой идентификаторы, а не текстовые строки вроде `"Greater Seattle Area"` и `"Philanthropy"`. Почему?

Если бы в UI были поля для ввода произвольного текста о географическом регионе и сфере деятельности, их имело бы смысл хранить в виде текстовых строк. Но в поддержке стандартизированных списков географических регионов и сфер деятельности с предоставлением пользователям возможности выбора из выпадающих списков или путем автодополнения есть свои преимущества:

- единообразный стиль и варианты написания в разных профилях;
- отсутствие неоднозначности (например, в случае нескольких городов с одним названием);
- удобство модификации — имя хранится только в одном месте, так что при необходимости можно легко поменять его во всей системе (например, в случае изменения названия города по политическим причинам);
- поддержка локализации — возможность при переводе сайта на различные языки локализовать стандартизированные списки, так что географический регион и сфера деятельности будут отображаться на языке пользователя;
- улучшенные возможности поиска — например, данный профиль может быть найден при поиске филантропов штата Вашингтон, поскольку в списке областей может быть закодирован факт нахождения Сиэтла в штате Вашингтон (что далеко не очевидно из строки `"Greater Seattle Area"`).

Хранить ли идентификатор или текстовую строку — вопрос дублирования данных. При использовании идентификатора значимая для людей информация (например, слово *philanthropy* — «благотворительность») хранится только в одном месте, а при ссылке на нее везде применяется ID (имеющий смысл только внутри этой базы данных). Однако при непосредственном хранении текста такая информация дублируется в каждой записи, задействующей эти сведения.

Преимущество идентификаторов состоит в том, что в силу отсутствия какого-либо смысла их для людей нет необходимости менять их в каких-либо случаях: ID способен оставаться тем же самым даже при изменении информации, на которую он указывает. А все значимое для людей может понадобиться поменять в будущем — и если эта информация дублируется, то придется обновлять все имеющиеся копии. Данное обстоятельство приводит к избыточности записи и риску несоответствий (когда одни копии информации обновлены, а другие — нет). Идея исключения подобного дублирования лежит в основе *нормализации* баз данных[1].



Администраторы и разработчики БД обожают спорить о нормализации и денормализации, но мы пока что не будем озвучивать свое мнение. В части III данной книги мы вернемся к этому вопросу и рассмотрим системный подход к кэшированию, денормализации и унаследованным данным.

К сожалению, нормализация этих данных требует связей *«многие-к-одному»* (множество людей живет в одной области или работает в одной сфере деятельности), которые плохо вписываются в документную модель. В реляционных БД считается нормальным ссылаться на строки в других таблицах по ID, поскольку выполнение соединений не представляет сложностей. В документоориентированных БД для древовидных структур соединения не нужны, так что их поддержка часто очень слаба[2].

Если СУБД сама по себе не поддерживает соединения, то приходится эмулировать соединение в коде приложения с помощью нескольких запросов к базе данных. (В этом случае списки географических регионов и сфер деятельности, вероятно, настолько невелики и меняются настолько редко, что приложение может их хранить в памяти. Тем не менее база делегирует работу по выполнению соединения коду приложения.)

[1] В литературе по реляционной модели выделяют несколько разных нормальных форм, но эти различия на практике особого интереса не представляют. В качестве эмпирического правила: если значения, которые могут храниться в одном месте, у вас дублируются, то схема не нормализована.

[2] На момент написания данной книги соединения поддерживаются в RethinkDB, не поддерживаются в MongoDB и поддерживаются только в заранее описанных представлениях в CouchDB.

Более того, даже если первоначальная версия приложения хорошо подходит для документной модели без соединений, внутренние связи данных имеют обыкновение усиливаться по мере добавления в приложение новых возможностей. Рассмотрим некоторые вероятные изменения в примере с резюме.

- *Компании и школы в виде сущностей.* В предыдущем описании элементы `organization` (компания, в которой работает пользователь) и `school_name` (место обучения) — просто строки. Могут ли они быть вместо этого ссылками на сущности? Тогда у каждой компании может быть своя веб-страница (с логотипом, лентой новостей и т. д.); каждое резюме будет включать ссылки на упоминаемые в нем компании и школы, а также их логотипы и другую информацию (см. пример из LinkedIn на рис. 2.3).
- *Рекомендации.* Допустим, вам нужно добавить новую возможность: создание пользователем рекомендации для другого пользователя. Рекомендации отображаются в резюме того пользователя, который их получил, вместе с именем и фотографией рекомендующего. Если дающий рекомендации обновляет свою фотографию, то во всех его рекомендациях должна отображаться новая фотография. Следовательно, рекомендация должна ссылаться на профиль ее автора.



Рис. 2.3. Название компании — не просто строка, а ссылка на сущность для этой компании. Снимок экрана с LinkedIn.com

Рисунок 2.4 иллюстрирует необходимость связей «многие-ко-многим» для этих новых возможностей. Данные внутри каждого пунктирного прямоугольника можно сгруппировать в один документ, но ссылки на компании, школы и других пользователей должны быть именно ссылками и потребуют соединений при выполнении запросов.

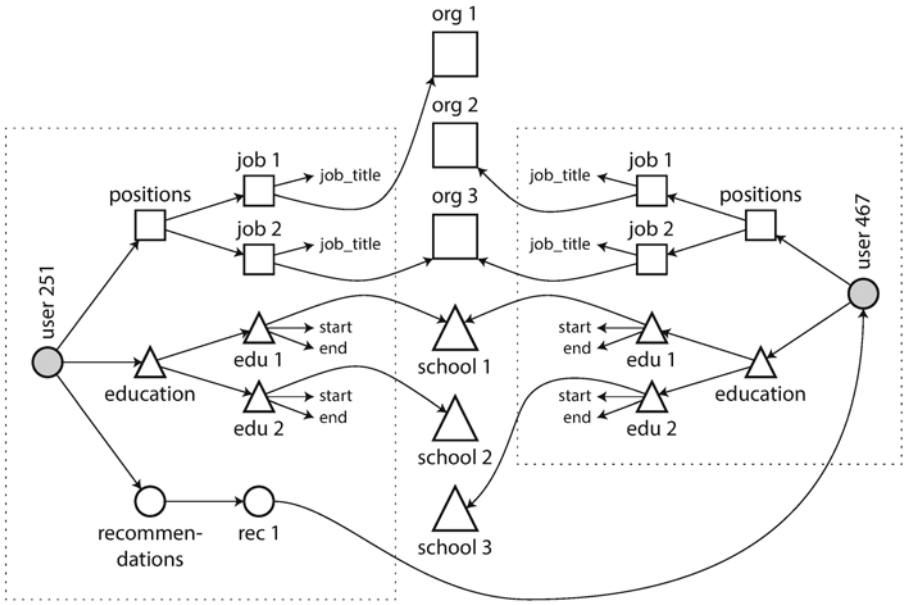


Рис. 2.4. Расширение резюме с помощью связей «многие-ко-многим»

Повторяется ли история в случае документоориентированных баз данных

Хотя связи «многие-ко-многим» и соединения регулярно используются в реляционных БД, документоориентированные базы данных и NoSQL возвращают нас к дискуссии о том, как лучше представить подобные отношения в БД. Эта дискуссия намного старше, чем NoSQL, — на самом деле она возвращает нас к первым дням компьютеризованных систем баз данных.

В 1970-х годах самой популярной СУБД для обработки коммерческих данных была *Система управления информацией* (Information Management System, IMS) компании IBM, изначально разработанная для управления запасами в космической программе «Аполлон» и впервые выпущенная в коммерческий оборот в 1968 году [13]. Она до сих пор используется и поддерживается, работая на операционной системе OS/390 на больших ЭВМ компании IBM [14].

Архитектура IMS применяет довольно простую модель данных под названием *«иерархическая модель»*, весьма схожую с JSON-моделью, которую задействуют документоориентированные БД [2]. Все данные в ней представлены в виде дерева записей, вложенных в другие записи, что весьма напоминает структуру JSON на рис. 2.2.

Как и документоориентированные базы данных, IMS хорошо работает в случае связей «один-ко-многим», но со связями «многие-ко-многим» дела у нее обстоят хуже и она не поддерживает соединения. Разработчикам приходится определять, дублировать (денормализовать) ли данные или вручную разрешать ссылки одной записи на другую. Эти проблемы 1960-х и 1970-х годов были очень похожи на те проблемы, с которыми сталкиваются разработчики документоориентированных баз данных сегодня [15].

Для устранения ограничений иерархической модели было предложено множество решений. Два наиболее известных — *реляционная модель* (ставшая впоследствии SQL и завладевшая всем миром) и *сетевая модель* (сначала имевшая множество поклонников, но впоследствии канувшая в Лету). «Великий спор» между двумя лагерями длился большую часть 1970-х [2].

Поскольку проблема, на решение которой были нацелены эти две модели, столь актуальна сейчас, имеет смысл вкратце изучить данный спор с современной точки зрения.

Сетевая модель

Сетевая модель была стандартизирована комитетом «Конференция по языкам систем обработки данных» (Conference on Data Systems Languages, CODASYL) и реализована несколькими разными производителями БД. Она известна также под названием *модели CODASYL* [16].

Модель CODASYL была обобщением иерархической модели. В древовидной структуре последней у каждой записи была ровно одна родительская запись. В сетевой же у каждой записи могло быть несколько родительских. Например, в базе могла быть одна запись для области `"Greater Seattle Area"` со ссылкой на нее для каждого живущего там пользователя. Благодаря этому появлялась возможность моделировать связи «многие-к-одному» и «многие-ко-многим».

Ссылки между записями в сетевой модели представляли собой не внешние ключи, а скорее нечто напоминающее указатели в языках программирования (хотя и хранящиеся на диске). Единственный способ обращения к записи заключался в следовании от корневой записи по этим цепочкам ссылок. Такой путь назывался *путем доступа* (access path).

В простейшем случае путь доступа напоминает обход связного списка: начинаем с головного элемента списка и просматриваем по одной записи за раз до тех пор, пока не найдем нужную. Но в мире связей «многие-ко-многим» к одной записи может вести несколько различных путей, и работавшим с сетевой моделью программистам приходилось держать в голове эти различные пути доступа.

Выполнение запросов в модели CODASYL представляло собой перемещение указателя по БД с помощью итерации по спискам записей и следования по путям доступа. Если у записи было несколько родительских записей (то есть несколько

указывающих на него ссылок с других записей), то код приложения должен был отслеживать все ее различные связи. Даже члены комитета CODASYL признали, что это выглядело как навигация по *n*-мерному пространству данных [17].

Хотя в 1970-х годах ручной выбор путей доступа позволял использовать чрезвычайно ограниченные аппаратные возможности (например, накопители на магнитной ленте с исключительно медленным поиском) наиболее эффективным образом, это приводило к сложному и негибкому коду для обновления БД и выполнения запросов к ней. Как в иерархической, так и в сетевой модели, если путь доступа к необходимым данным был вам неизвестен, то вы оказывались в сложном положении. Чтобы изменить пути доступа, приходилось пробираться сквозь завалы написанного вручную кода запросов и переписывать его для новых путей доступа.

Реляционная модель

Реляционная модель, напротив, выставляла все данные напоказ: отношение (таблица) — просто набор кортежей (строк), вот и все. Никаких лабиринтообразных вложенных структур, никаких запутанных путей доступа, необходимых для просмотра данных. Можно прочитать любую строку в таблице или даже все строки, выбирая те, что соответствуют произвольному условию. Можно прочитать конкретную строку путем объявления некоторых столбцов ключом и выполнения поиска по этому ключу. Можно вставить новую строку в любую таблицу, не беспокоясь о связях по внешним ключам с другими таблицами[1].

В реляционной базе данных оптимизатор запросов автоматически принимает решение о том, в каком порядке выполнять части запроса и какие индексы использовать. По сути, эти решения представляют собой «путь доступа», но важное различие заключается в том, что они принимаются автоматически оптимизатором запросов, а не вручную разработчиком приложения, благодаря чему нам редко приходится об этом задумываться.

При необходимости выполнять запросы данных по-другому достаточно объявить новый индекс, и запросы будут автоматически использовать наиболее подходящие индексы. Нет необходимости менять запросы для применения нового индекса. (См. также раздел 2.2.) Реляционная модель, таким образом, сильно упрощает добавление новых возможностей в приложение.

Оптимизаторы запросов для реляционных баз данных — непростые вещи, они потребовали многих лет исследований и разработки [18]. Но ключевым моментом реляционной модели было следующее: достаточно один раз создать оптимизатор

[1] Ограничения внешних ключей дают возможность ограничивать изменения, но они не являются обязательными в реляционной модели. Даже если они есть, соединения по внешним ключам выполняются при запросе, в то время как в CODASYL соединение выполняется при вставке.

запросов, и все использующие БД приложения смогут извлечь из него выгоду. Если оптимизатора запросов у вас нет, то будет легче вручную написать код путей доступа для конкретного запроса, чем создавать универсальный оптимизатор, — но в долгосрочной перспективе универсальное решение имеет преимущество.

Сравнение с документоориентированными базами данных

Документоориентированные базы данных снова возвращаются к иерархической модели в одном нюансе — сохранении вложенных записей (связей «один-ко-многим», таких как `positions`, `education` и `contact_info` на рис. 2.1) внутри их родительской записи вместо отдельной таблицы.

Однако в вопросе представления связей «многие-к-одному» и «многие-ко-многим» реляционные и документоориентированные базы данных, по существу, не отличаются: в обоих случаях обращение к соответствующему элементу происходит с помощью уникального идентификатора, именуемого *внешним ключом* (foreign key) в реляционной модели и *ссылкой на документ* (document reference) в документной модели [9]. Этот идентификатор разрешается во время чтения с помощью соединения или дополнительных запросов. До сих пор документоориентированные БД не следуют пути, проложенному CODASYL.

Реляционные и документоориентированные базы данных сегодня

Существует множество различий между реляционными и документоориентированными базами данных, включая характеристики отказоустойчивости (см. главу 5) и реализацию конкурентного доступа (см. главу 7). В данной главе мы сосредоточим наше внимание только на отличиях в модели данных.

Основные доводы в пользу документной модели данных — гибкость схемы, лучшая производительность вследствие локальности и большая близость к применяемым структурам данных (для некоторых приложений). Реляционная модель отвечает на это лучшей поддержкой соединений, а также связей «многие-к-одному» и «многие-ко-многим».

Какая из моделей приводит к более простому коду приложения

Если структура данных в приложении — документоподобная (то есть представляет собой дерево связей «один-ко-многим», причем обычно все дерево загружается сразу), то использование документной модели, вероятно, будет хорошей идеей. Реляционная методика *расщепления* (shredding) — разложения документоподобной структуры на множество таблиц (таких как `positions`, `education`

и `contact_info` на рис. 2.1) — может привести к запутанным схемам и чрезмерно усложненному коду приложения.

У документной модели есть свои ограничения: например, нельзя непосредственно ссылаться на вложенный элемент внутри документа, приходится говорить что-то вроде «второй элемент списка должностей для пользователя 251» (очень напоминает пути доступа в иерархической модели). Однако если вложенность документов не слишком велика, то это обычно не представляет проблемы.

Плохая поддержка соединений в документоориентированных базах данных может быть, а может и не быть проблемой, в зависимости от приложения. Например, связи «многие-ко-многим» могут никогда не понадобиться в аналитическом приложении, использующем документоориентированную БД для записи времени наступления событий [19].

Однако если в приложении используются связи «многие-ко-многим», то документная модель представляется менее привлекательной. Количество необходимых соединений можно снизить путем денормализации, но коду приложения понадобится выполнять дополнительную работу по поддержанию денормализованных данных в согласованном состоянии. Соединения эмулируются в коде приложения с помощью нескольких запросов к БД, но это усложняет приложение и обычно работает медленнее, чем выполняемое специализированным кодом СУБД соединение. В подобных случаях применение документной модели может привести к намного более сложному коду приложения и снижению производительности [15].

Невозможно оценить в общем, какая из моделей ведет к более простому коду приложения, это зависит от видов связей, существующих между элементами данных. При тесных внутренних связях документная модель довольно неуклюжа, реляционная вполне приемлема, а графовые (см. раздел 2.3) наиболее естественны.

Гибкость схемы в документной модели

Большинство документоориентированных БД, а также реляционные БД с поддержкой JSON не навязывают какой-либо схемы для данных в документах. Поддержке XML в реляционных БД обычно сопутствует необязательная проверка схемы. Отсутствие последней означает, что в документ можно добавлять произвольные ключи и значения, и при чтении клиенты не будут уверенны в том, какие поля могут содержать документы.

Документоориентированные БД обычно называют *бессхемными* (schemaless) или *неструктурированными*. Но это название может ввести в заблуждение, ведь читающий данные код обычно предполагает наличие у последних определенной структуры — то есть какая-то неявная схема все равно есть, но она не навязывает-

ся базой [20]. Более точным было бы название *schema-on-read* (схема при чтении: структура данных неявна и их интерпретация происходит при чтении), в отличие от *schema-on-write* (схема при записи: традиционный подход реляционных БД, при котором имеется явная схема и база гарантирует, что все записываемые данные ей соответствуют) [21].

Схема при чтении аналогична динамической (во время выполнения) проверке типов данных в языках программирования, в то время как схема при записи аналогична статической (во время компиляции) проверке типов. Аналогично тому, как сторонники статической и динамической проверки типов жарко спорят об их относительных достоинствах и недостатках [22], навязывание схемы в базе данных — спорный вопрос, и в целом на него не существует правильного или неправильного ответа.

Различие между этими подходами особенно заметно в ситуациях, когда приложению необходимо изменить формат его данных. Например, допустим, что в настоящий момент мы храним полное имя пользователя в одном поле и нам хотелось бы хранить имя и фамилию отдельно [23]. В документоориентированной БД можно было бы просто начать писать новые документы с новыми полями и создать код в приложении, который обрабатывает случай чтения документов в старом формате. Например:

```
if (user && user.name && !user.first_name) {
    // В записанных до 8 декабря 2013 года документах нет поля first_name
    user.first_name = user.name.split(" ")[0];
}
```

С другой стороны, в схемы базы данных со статической проверкой типов обычно приходится выполнять *миграцию* примерно следующим образом:

```
ALTER TABLE users ADD COLUMN first_name text;
UPDATE users SET first_name = split_part(name, ' ', 1);      -- PostgreSQL
UPDATE users SET first_name = substring_index(name, ' ', 1);      -- MySQL
```

Об изменениях схемы обычно говорят как о медленных и требующих простоя системы. Это не совсем верно: большинство реляционных БД выполняют оператор `ALTER TABLE` за несколько миллисекунд. СУБД MySQL — известное исключение: она копирует всю таблицу при выполнении команды `ALTER TABLE`, что означает минуты или даже часы простоя в случае изменения схемы большой таблицы — хотя существуют различные утилиты для обхода данного ограничения [24–26].

Оператор `UPDATE` на большой таблице будет выполняться медленно в любой базе данных, поскольку приходится перезаписывать все строки. Если это неприемлемо, то приложение может оставить значение поля `first_name` равным `NULL` (по умолчанию) и заполнить его во время чтения, как в случае с документоориентированной БД.

Подход «схема при чтении» предпочтителен, если структура элементов набора различается по какой-либо причине (то есть данные разнородны), например, в таких ситуациях:

- существует множество различных типов объектов, и нет смысла помещать каждый из них в отдельную таблицу;
- структура данных определяется внешними системами, не подконтрольными вам и способными изменяться с течением времени.

В подобных ситуациях схема может принести больше вреда, чем пользы, а неструктурированные документы окажутся намного более естественной моделью данных. Но в случаях, когда структура всех записей предположительно одинакова, схемы — удобный механизм для документирования и обеспечения соблюдения этой структуры. Мы подробнее обсудим схемы и их эволюцию в главе 4.

Локальность данных и запросы

Документы обычно хранятся в виде единых непрерывных строк, закодированных в формат JSON, XML или их двоичную разновидность (например, формат BSON базы данных MongoDB). Если приложению часто требуется доступ ко всему документу (например, для визуализации веб-страницы), то *локальность хранилища* дает определенные преимущества. Если данные разбиты по нескольким таблицам, как на рис. 2.1, то понадобится несколько поисков по индексу для полного их извлечения, что потребует больше операций дискового поиска и займет больше времени.

Локальность дает преимущество только тогда, когда требуется получить большие части документа за один раз. Базе данных обычно приходится загружать весь документ целиком, даже если вам нужен только маленький его фрагмент, что в случае больших документов будет неэкономно. При обновлении документа, как правило, необходимо тоже переписать весь документ целиком — легко выполнить «на месте» можно только те изменения, которые не меняют закодированного размера документа [19]. Поэтому в большинстве случаев рекомендуется минимизировать размер документов и избегать увеличивающих этот размер операций записи [9]. Указанные ограничения по производительности значительно снижают диапазон случаев, когда документоориентированные БД могут оказаться полезны.

Заслуживает упоминания то, что идея группировать вместе родственные данные в целях локальности не ограничивается документной моделью. Например, БД Spanner компании Google обеспечивает те же свойства локальности в реляционной модели данных, благодаря тому что схема способна объявить о необходимости чередования (вложенности) строк таблицы внутри родительской [27].

В СУБД Oracle может быть то же самое благодаря возможности под названием *«многотабличные кластеризованные индекс-таблицы»* (multi-table index cluster tables) [28]. Идея *семейства столбцов* (column-family) в модели данных Bigtable (используемой в СУБД Cassandra и HBase) тоже заключается в управлении локальностью [29].

Мы также познакомимся с локальностью ближе в главе 3.

Постепенное сближение документоориентированных и реляционных баз данных

Большинство систем реляционных БД (кроме MySQL) поддерживают XML с середины 2000-х годов, включая функции для выполнения локальных изменений в XML-документах и возможность индексации последних и выполнения запросов к ним. Это позволяет приложениям задействовать модели данных, очень похожие на те, которые были бы у них при использовании документоориентированной БД.

У СУБД PostgreSQL, начиная с версии 9.3 [8], MySQL, начиная с версии 5.7, и IBM DB2, начиная с версии 10.5 [30], был примерно такой же уровень поддержки JSON-документов. Учитывая широкое использование JSON в веб-API, вполне вероятно, что другие реляционные базы данных пойдут по их следам и добавят поддержку формата JSON.

Что касается документоориентированных баз данных, RethinkDB поддерживает в своем языке запросов соединения, напоминающие реляционные, а некоторые драйверы MongoDB автоматически разрешают ссылки на БД. (По существу, таким образом производится клиентское соединение, хотя и более медленное, чем выполняемое в БД, поскольку при нем данные перемещаются по сети в два конца и оно менее оптимизировано.)

Похоже, что реляционные и документоориентированные БД становятся все более схожими, и это хорошо: их модели данных дополняют друг друга[1].

Нам представляется, что будущее — за гибридами реляционной и документной моделей.

---

[1] Первоначальное описание реляционной модели [1], предложенное Коддом, на самом деле допускало использование в реляционной схеме чего-то довольно похожего на JSON-документы. Кодд называл это непростыми предметными областями (nonsimple domains). Идея заключалась в том, что тип значения в строке не обязательно должен быть простым (как число или строка символов), а может быть вложенным отношением (таблицей) — так что в роли значения способна выступить произвольная вложенная древовидная структура, очень похожая на JSON или XML, которые SQL начал поддерживать на 30 лет позже.

2.2. Языки запросов для данных

Одним из новшеств реляционной модели был новый способ запроса данных: SQL — *декларативный* язык запросов, в то время как IMS и CODASYL используют для выполнения запросов *императивный* код. Что это значит?

Множество распространенных языков программирования — императивные. Например, при наличии списка видов животных можно написать нечто подобное для получения из него только акул:

```
function getSharks() {
    var sharks = [];
    for (var i = 0; i < animals.length; i++) {
        if (animals[i].family === "Sharks") {
            sharks.push(animals[i]);
        }
    }
    return sharks;
}
```

А в реляционной алгебре вместо этого можно написать следующее:

$$sharks = \sigma_{family = Sharks}(animals)$$[1]

где σ (греческая буква сигма) — оператор выбора, возвращающий только тех животных, которые соответствуют условию *family = Sharks*.

SQL весьма близко следует структуре реляционной алгебры:

```
SELECT * FROM animals WHERE family = 'Sharks';
```

Императивный язык говорит компьютеру выполнить определенные операции в заданном порядке. Вы можете представить это как движение по коду строка за строкой, вычисление условных выражений, обновление переменных и принятие решения о том, пройти ли цикл еще раз.

В декларативных языках запросов, таких как SQL или реляционная алгебра, следует описать шаблон необходимых данных — каким условиям должны соответствовать результаты и как данные должны быть преобразованы (например, отсортированы, сгруппированы и агрегированы) — но не то, как добиться этого. Решение о том, какие индексы и методы соединения использовать и в каком порядке выполнять различные части запроса, должен принять оптимизатор запросов СУБД.

Декларативные языки запросов весьма привлекательны в силу большей сжатости команд и большего удобства работы с ними, чем с императивными API. Но что важнее, они скрывают подробности реализации ядра базы данных, благодаря чему

[1] акулы = $\sigma_{семейство = Акулы}$ (животные) — *Примеч. пер.*

у СУБД появляется возможность повышать производительность без необходимости вносить изменения в запросы.

Например, в показанном в начале данного раздела императивном коде список животных выводится в определенном порядке. Если базе данных понадобится незаметно вернуть в обращение неиспользуемое пространство на диске, то может возникнуть необходимость «перетасовать» записи, что приведет к изменению порядка вывода животных. Получится ли у базы сделать это безопасным образом, без нарушения работы существующих запросов?

Пример SQL не гарантирует конкретного порядка, следовательно, для него не важно, что порядок способен измениться. Но если запрос написан в виде императивного кода, то база данных не может быть уверена, важен ли порядок для кода. Функциональность SQL более ограничена, но это обстоятельство обеспечивает гораздо более широкие возможности для автоматической оптимизации.

Наконец, декларативные языки часто хорошо подходят для параллельного выполнения. Сегодня ускорение CPU происходит за счет добавления дополнительных ядер, а не повышения тактовой частоты [31]. Распараллелить императивный код по нескольким ядрам и нескольким машинам очень непросто, поскольку в нем задается определенный порядок выполнения команд. Шансы декларативных языков на ускорение за счет параллельного выполнения выше, поскольку они задают лишь шаблон результатов, а не используемый для их получения алгоритм. База данных при желании может задействовать параллельную реализацию языка запросов [32].

Декларативные запросы в Интернете

Преимущества декларативных языков запросов не ограничиваются использованием только в базах данных. Чтобы проиллюстрировать это утверждение, сравним декларативный и императивный подход в совершенно другой среде: браузере.

Допустим, у нас есть сайт, посвященный морским животным. Пользователь сейчас просматривает страницу об акулах, так что мы помечаем пункт меню Sharks как выбранный в настоящий момент, вот так:

```
<ul>
    <li class="selected"> (1)
        <p>Sharks</p> (2)
        <ul>
            <li>Great White Shark</li>
            <li>Tiger Shark</li>
            <li>Hammerhead Shark</li>
        </ul>
    </li>
    <li>
        <p>Whales</p>
        <ul>
            <li>Blue Whale</li>
```

```
            <li>Humpback Whale</li>
            <li>Fin Whale</li>
        </ul>
    </li>
</ul>
```

❶ Выбранный пункт помечен классом CSS `"selected"`.

❷ Заголовок выбранной в настоящий момент страницы — `<p>Sharks</p>`.

Теперь допустим, что фон заголовка выбранной в настоящий момент страницы должен быть синим — для визуального выделения. Это можно легко сделать с помощью CSS:

```
li.selected > p {
    background-color: blue;
}
```

Тут CSS-селектор `li.selected > p` объявляет шаблон для элементов, для которых мы выбираем синий стиль: это все элементы `<p>`, чьим непосредственным родителем является элемент `<li>` с CSS-классом `selected`. Элемент `<p>Sharks</p>` в примере соответствует этому шаблону, а `<p>Whales</p>` — нет, поскольку у его родительского класса отсутствует `class="selected"`.

Если же использовать XSL вместо CSS, то можно сделать нечто схожее:

```
<xsl:template match="li[@class='selected']/p">
    <fo:block background-color="blue">
        <xsl:apply-templates/>
    </fo:block>
</xsl:template>
```

В этом фрагменте кода XPath-выражение `li[@class='selected']/p` эквивалентно CSS-селектору `li.selected > p` из предыдущего примера. XSL и CSS объединяет то, что они оба — *декларативные* языки описания стилей документа.

Представьте только, как выглядела бы ваша жизнь, если бы пришлось задействовать императивный подход. В JavaScript при использовании базового API объектной модели документа (document object model, DOM) результат выглядел бы примерно так:

```
var liElements = document.getElementsByTagName("li");
for (var i = 0; i < liElements.length; i++) {
    if (liElements[i].className === "selected") {
        var children = liElements[i].childNodes;
        for (var j = 0; j < children.length; j++) {
            var child = children[j];
            if (child.nodeType === Node.ELEMENT_NODE && child.tagName === "P") {
                child.setAttribute("style", "background-color: blue");
            }
        }
    }
}
```

JavaScript императивно задает синий цвет фона для элемента `<p>Sharks</p>`, но код ужасен. Он не только намного длиннее и труднее для понимания, чем эквивалентный код XSL/CSS, но и включает некоторые довольно серьезные проблемы.

- При удалении класса `selected` (например, при щелчке пользователя на другой странице) синий фон останется даже в случае повторного выполнения кода, так что элемент останется выделенным до тех пор, пока не будет перезагружена страница целиком. В CSS браузер автоматически определит, когда правило `li.selected > p` перестанет действовать и уберет синий фон сразу же при удалении класса `selected`.
- Если вам хотелось бы воспользоваться преимуществами нового API, например `document.getElementsByClassName("selected")` или даже `document.evaluate()` — для улучшения производительности, то придется переписать код. С другой стороны, создатели браузеров имеют возможность улучшать производительность CSS и XPath без нарушения совместимости.

В браузере использование декларативных CSS-стилей намного удобнее императивного управления стилями из JavaScript. Аналогично в базах данных декларативные языки запросов, такие как SQL, оказываются намного более удобными, чем императивные API запросов[1].

Выполнение запросов с помощью MapReduce

MapReduce — модель программирования для обработки больших объемов данных блоками на множестве машин, активно продвигаемая компанией Google [33]. В ограниченном виде MapReduce поддерживается некоторыми NoSQL-хранилищами, включая MongoDB и CouchDB в качестве механизма выполняющих только чтение запросов по многим документам.

MapReduce в общем описывается подробнее в главе 10. Пока что мы просто вкратце обсудим использование этой модели в MongoDB.

MapReduce не является ни декларативным языком запросов, ни полностью императивным API запросов. Это нечто среднее: логика запроса выражается с помощью фрагментов кода, которые используемый для обработки фреймворк вызывает многократно. Он основывается на функциях `map` (известной также под названием `collect`) и `reduce` (либо иначе `fold` или `inject`), существующих во многих функциональных языках программирования.

В качестве примера допустим, что вы морской биолог и каждый раз, когда видите в океане животных, вносите запись о наблюдении в свою базу данных. А теперь вам нужно сформировать отчет о количестве увиденных за месяц акул.

---

[1] Как IMS, так и CODASYL используют императивные API запросов. Приложения обычно применяют код на языке программирования COBOL для итерации по записям в базе данных (одна запись за раз) [2, 16].

В PostgreSQL этот запрос мог бы выглядеть примерно следующим образом:

```
SELECT date_trunc('month', observation_timestamp) AS observation_month,
       sum(num_animals) AS total_animals ❶
FROM observations
WHERE family = 'Sharks'
GROUP BY observation_month;
```

❶ Функция `date_trunc('month', timestamp)` определяет содержащий `timestamp` календарный месяц и возвращает другую метку даты/времени, соответствующую началу этого месяца. Другими словами, метка даты/времени округляется до ближайшего месяца[1].

Данный запрос сначала фильтрует наблюдения так, чтобы остались только виды, относящиеся к семейству `Sharks`, после чего группирует наблюдения по календарному месяцу и, наконец, суммирует количества наблюдавшихся в каждом из месяцев животных.

Это же можно выразить с помощью представленной ниже возможности модели MapReduce базы данных MongoDB:

```
db.observations.mapReduce(
    function map() { ❷
        var year  = this.observationTimestamp.getFullYear();
        var month = this.observationTimestamp.getMonth() + 1;
        emit(year + "-" + month, this.numAnimals); ❸
    },
    function reduce(key, values) { ❹
        return Array.sum(values); ❺
    },
    {
        query: { family: "Sharks" }, ❶
        out: "monthlySharkReport" ❻
    }
);
```

❶ Фильтр, отсеивающий только акул, можно задать декларативным образом (это расширение MongoDB модели MapReduce).

❷ JavaScript-функция `map` вызывается однократно для каждого документа, соответствующего `query`, где `this` соответствует объекту документа.

❸ Функция `map` порождает ключ (строку, состоящую из года и месяца, например `"2013-12"` или `"2014-1"`) и значения (количества животных в данном наблюдении).

❹ Порождаемые функцией `map` пары «ключ — значение» группируются по ключу. Для всех таких пар с одинаковым ключом (то есть с одним и тем же месяцем и годом) однократно вызывается функция `reduce`.

[1] Иными словами, отбрасывается дробная часть месяца. — *Примеч. пер.*

❺ Функция `reduce` суммирует количество животных, наблюдавшихся в конкретном месяце.

❻ Итоговые результаты записываются в коллекцию `monthlySharkReport`.

Например, пусть коллекция `observations` содержит следующие два документа:

```
{
    observationTimestamp: Date.parse("Mon, 25 Dec 1995 12:34:56 GMT"),
    family:     "Sharks",
    species:    "Carcharodon carcharias",
    numAnimals: 3
}
{
    observationTimestamp: Date.parse("Tue, 12 Dec 1995 16:17:18 GMT"),
    family:     "Sharks",
    species:    "Carcharias taurus",
    numAnimals: 4
}
```

Функция `map` должна вызываться по разу для каждого документа, в результате чего вызывается `emit("1995-12", 3)` и `emit("1995-12", 4)`. Затем функция `reduce` будет вызвана с параметрами `reduce("1995-12", [3, 4])` и вернет 7.

Разрешенные функциям `map` и `reduce` действия несколько ограничены. Эти функции должны быть *чистыми* (pure), то есть могут использовать только передаваемые в них входные данные, но не могут выполнять дополнительные запросы к БД и не должны иметь каких-либо побочных эффектов. Указанные ограничения позволяют базе выполнять эти функции где угодно, в любом порядке и повторно их запускать в случае неудачи. Тем не менее они обладают немалыми возможностями: способны производить синтаксический разбор строк, вызывать библиотечные функции, выполнять вычисления и др.

MapReduce — модель программирования довольно низкого уровня, предназначенная для распределенных вычислений на кластерах машин. Языки запросов более высокого уровня, такие как SQL, можно реализовать в виде конвейера операций MapReduce (см. главу 10), но существует также множество распределенных реализаций SQL, не использующих MapReduce. Обратите внимание: в языке SQL нет ничего, что бы запрещало его применение на отдельной машине, а у MapReduce нет монополии на распределенное выполнение запросов.

Использование кода на языке JavaScript — отличная возможность для продвинутых запросов, но это позволительно не только в MapReduce: некоторые базы данных SQL тоже могут расширять запросы JavaScript-функциями [34].

Одно из неудобств использования MapReduce: приходится создавать две тщательно согласованные JavaScript-функции, что обычно сложнее, чем написать отдельный запрос. Более того, декларативные языки запросов обычно обеспечивают оптимизатору запросов больше возможностей по улучшению производительности запроса.

Поэтому в версии 2.2 MongoDB была добавлена поддержка декларативного языка запросов под названием *конвейер агрегирования* (aggregation pipeline) [9]. На этом языке тот же запрос подсчета акул выглядел бы следующим образом:

```
db.observations.aggregate([
    { $match: { family: "Sharks" } },
    { $group: {
        _id: {
            year:  { $year:  "$observationTimestamp" },
            month: { $month: "$observationTimestamp" }
        },
        totalAnimals: { $sum: "$numAnimals" }
    } }
]);
```

Язык конвейера агрегирования напоминает по своим выразительным возможностям подмножество SQL, но использует синтаксис на основе формата JSON, а не SQL-синтаксис в стиле английских фраз. Описанные различия, вероятно, дело вкуса. Мораль этой истории: NoSQL-система может случайно прийти к повторному изобретению SQL, хотя и в замаскированном виде.

2.3. Графоподобные модели данных

Мы уже видели, что связи «многие-ко-многим» — важная характеристика различных моделей данных. Если в вашем приложении связи между записями в основном «один-ко-многим» (данные структурированы в виде деревьев) или вообще отсутствуют, то вам вполне подойдет документная модель.

Но что, если связи «многие-ко-многим» встречаются в ваших данных очень часто? С помощью реляционной модели можно иметь дело с простыми случаями связей «многие-ко-многим», но по мере роста сложности взаимосвязей внутри данных моделирование данных в виде графа будет более естественным.

Графы состоят из двух типов объектов: *вершин* (vertice), известных также как *узлы* (node) или *сущности* (entity), и *ребер* (edge), известных также как *связи* (relationship) или *дуги* (arc). В виде графа можно смоделировать множество типов данных. В качестве типичных примеров рассмотрим следующие.

- *Социальные графы*. Вершины — люди, а ребра отражают знакомство людей друг с другом.
- *Веб-графы*. Вершины — веб-страницы, а ребра отражают HTML-ссылки на другие страницы.
- *Дороги или железнодорожные сети*. Вершины — перекрестки (узловые станции), а ребра соответствуют пролегающим между ними дорогам или железнодорожным линиям.

Для работы с этими графами предназначены хорошо известные алгоритмы: например, автомобильные навигационные системы выполняют поиск кратчайшего пути между двумя точками дорожной сети, а для определения популярности веб-страницы (а значит, и ее места в результатах поиска) можно использовать алгоритм PageRank на веб-графе.

В только что приведенных примерах все вершины графов представляют, по сути, одно и то же (людей, веб-страницы или перекрестки дорог соответственно). Однако использование графов не ограничивается подобными *однородными* (homogeneous) данными: графы предоставляют ничуть не меньшие возможности хранения различных типов объектов в одном хранилище данных. Например, Facebook поддерживает единый граф с множеством различных типов вершин и ребер: вершины означают людей, местоположения, события, входы в систему и написанные пользователями комментарии; ребра указывают, какие люди являются друзьями, где произошел конкретный вход в систему, кто какое сообщение прокомментировал, кто какое мероприятие посетил и т. д. [35].

В этом разделе мы воспользуемся показанным на рис. 2.5 примером. Он вполне мог бы быть взят из социальной сети или генеалогической базы данных: в нем показаны два человека — Люси (Lucy) из штата Айдахо и Ален (Alain) из Бона, Франция. Они женаты и живут в Лондоне.

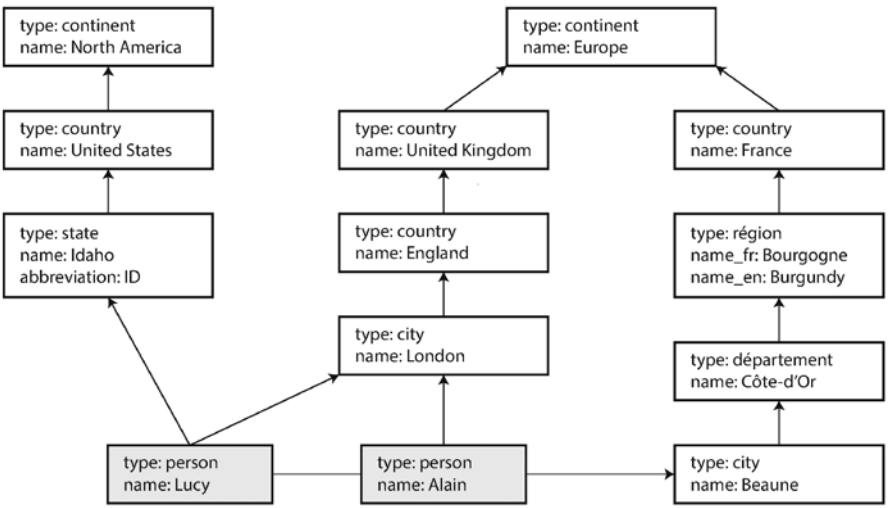


Рис. 2.5. Пример структурированных в виде графа данных (прямоугольники означают вершины, стрелки — ребра)

Существует несколько различных, хотя и схожих, способов структурирования графовых данных и выполнения к ним запросов. В этом разделе мы обсудим модель *графа свойств* (property graph), реализованную такими графовыми СУБД, как

Neo4j, Titan и InfiniteGraph, и модель *хранилища тройных кортежей* (triple-store), реализованную Datomic, AllegroGraph и др. Мы рассмотрим три декларативных языка запросов для графов: Cypher, SPARQL и Datalog. Кроме них, существует императивный графовый язык запросов под названием Gremlin [36] и фреймворк для работы с графами — Pregel (см. главу 10).

Графы свойств

В модели графов свойств каждая вершина состоит из:

- уникального идентификатора;
- множества исходящих ребер;
- множества входящих ребер;
- коллекции свойств (пар «ключ — значение»).

Каждое ребро состоит из:

- уникального идентификатора;
- вершины, с которой оно начинается *(начальная вершина)*;
- вершины, которой оно заканчивается *(конечная вершина)*;
- коллекции свойств (пар «ключ — значение»).

Графовое хранилище можно представить как состоящее из двух реляционных таблиц, одна для вершин, вторая — для ребер, как показано в примере 2.2 (здесь для хранения свойств вершин и ребер используется тип данных `json` СУБД PostgreSQL). Для каждого ребра хранится информация о начальной и конечной вершине, так что при необходимости получить множество входящих или исходящих ребер для данной вершины можно запросить поля `head_vertex` и `tail_vertex` соответственно из таблицы `edges`.

Пример 2.2. Представление графа свойств с использованием реляционой схемы

```
CREATE TABLE vertices (
    vertex_id   integer PRIMARY KEY,
    properties  json
);
CREATE TABLE edges (
    edge_id     integer PRIMARY KEY,
    tail_vertex integer REFERENCES vertices (vertex_id),
    head_vertex integer REFERENCES vertices (vertex_id),
    label       text,
    properties  json
);
CREATE INDEX edges_tails ON edges (tail_vertex);
CREATE INDEX edges_heads ON edges (head_vertex);
```

Некоторые важные аспекты этой модели представлены ниже.

1. Любая вершина может быть соединена ребром с любой другой вершиной. Схема не накладывает ограничения на то, какие элементы могут быть связаны.
2. Для любой вершины можно найти как ее входящие, так и исходящие ребра и таким образом выполнить *обход* графа: найти путь по цепочке вершин — как в прямом, так и обратном направлении (именно поэтому в примере 2.2 есть индекс как на столбце `tail_vertex`, так и столбце `head_vertex`).
3. Задействуя различные метки для разных видов связей, можно хранить в одном графе несколько разных видов информации, сохраняя при этом чистоту модели.

Как показано на рис. 2.5, представленные возможности придают графам значительную гибкость при моделировании данных. На этом рисунке показано несколько вещей, которые было бы непросто выразить с помощью обычной реляционной схемы. В качестве примера рассмотрим различные типы административных единиц в разных странах (Франция делится на департаменты и регионы, а США — на округа и штаты), такие исторические особенности, как страна внутри страны (проигнорируем пока что хитросплетения суверенных государств и наций), и различную степень детализации данных (в качестве текущего места жительства Люси указан город, а ее место рождения указано лишь на уровне штата).

Представим расширение графа в целях включения многих других фактов о Люси и Алене и других людях. Например, можно воспользоваться им для указания имеющихся у них аллергий на продукты питания (путем введения в граф вершины для каждого аллергена и ребра между человеком и аллергеном для указания на аллергию) и связав аллергены множеством вершин для демонстрации того, какие продукты питания содержат те или иные вещества. После этого написать запрос, чтобы узнать, кто какие продукты может есть без опаски. Графы удобны своими возможностями расширения: по мере добавления в приложение новых свойств можно легко расширить граф с целью учесть изменения в структурах данных приложения.

Язык запросов Cypher

Cypher — декларативный язык запросов для графов свойств, созданный для графовой базы данных Neo4j [37] (он получил название в честь персонажа фильма «Матрица» и не имеет никакого отношения к криптографическим шифрам [38]).

Пример 2.3 демонстрирует запрос на языке Cypher, предназначенный для вставки левой части рис. 2.5 в графовую базу данных. Оставшуюся часть графа можно вставить аналогично, поэтому я опустил ее ради удобства чтения. Каждой вершине дано условное название, например `USA` или `Idaho`. Остальные части запроса могут использовать эти названия для создания ребер между вершинами с помощью

стрелочной нотации: команда `(Idaho) -[:WITHIN]-> (USA)` создает ребро с меткой `WITHIN`, с начальной вершиной `Idaho` и конечной вершиной `USA`.

Пример 2.3. Подмножество данных с рис. 2.5, представленное в виде запроса Cypher

```
CREATE
  (NAmerica:Location {name:'North America', type:'continent'}),
  (USA:Location      {name:'United States', type:'country'  }),
  (Idaho:Location    {name:'Idaho',         type:'state'    }),
  (Lucy:Person       {name:'Lucy' }),
  (Idaho) -[:WITHIN]->  (USA)  -[:WITHIN]-> (NAmerica),
  (Lucy)  -[:BORN_IN]-> (Idaho)
```

После вставки всех вершин и ребер рис. 2.5 в базу данных можно начать задавать интересные вопросы: например, *найти имена всех людей, эмигрировавших из США в Европу*. Точнее, нам нужно найти все вершины, ребро которых `BORN_IN` указывает на географический пункт, расположенный в США, а ребро `LIVING_IN` — на расположенный в Европе, и вернуть свойство `name` всех этих вершин.

В примере 2.4 показано, как выразить этот запрос на языке Cypher. Для поиска заданных шаблонов в графе используется та же стрелочная нотация в предложении `MATCH`: `(person) -[:BORN_IN]-> ()` найдет любые две вершины, относящиеся к ребру с меткой `BORN_IN`. Начальная вершина данного ребра привязана к переменной `person`, а конечная оставлена безымянной.

Пример 2.4. Запрос Cypher для поиска людей, эмигрировавших из США в Европу

```
MATCH
  (person) -[:BORN_IN]->  () -[:WITHIN*0..]-> (us:Location {name:'United States'}),
  (person) -[:LIVES_IN]-> () -[:WITHIN*0..]-> (eu:Location {name:'Europe'})
RETURN person.name
```

Этот запрос расшифровывается следующим образом.

1. Найти все вершины (назовем их `person`), удовлетворяющие *обоим* условиям.
   - У вершины `person` имеется исходящее ребро `BORN_IN` к какой-либо вершине. Из этой вершины можно построить цепочку исходящих ребер `WITHIN` вплоть до достижения вершины типа `Location`, свойство `name` которой равно `"United States"`.
   - У той же вершины person имеется исходящее ребро `LIVES_IN`. Следуя по данному ребру, а затем по цепочке исходящих ребер `WITHIN`, можно в конце концов достичь вершины типа `Location`, свойство `name` которой равно `"Europe"`.
2. Вернуть свойство `name` для каждой подобной вершины `person`.

Существует несколько возможных способов выполнения подобного запроса. В соответствии с приведенным выше описанием необходимо начать с просмотра всех людей

в базе данных, проверки места рождения и жительства каждого из них с возвратом только тех людей, которые удовлетворяют установленным критериям.

Но точно так же можно начать с двух вершин `Location` и идти в обратном направлении. Если по свойству `name` построен индекс, то можно достаточно эффективно найти две вершины, соответствующие США и Европе. После этого найти все географические объекты (штаты, области, города и т. п.) в США и Европе соответственно, следуя по всем входящим ребрам `WITHIN`. Наконец, проверить людей, найденных по входящим в одну из вершин-местоположений ребрам `BORN_IN` и `LIVES_IN`.

Как это обычно бывает в декларативных языках запросов, нет необходимости указывать подобные подробности выполнения при написании запроса: оптимизатор запросов автоматически выберет наиболее эффективную с его точки зрения стратегию, а вы можете тем временем писать оставшуюся часть приложения.

Графовые запросы в SQL

Пример 2.2 предполагает, что графовые данные можно представить в реляционной БД. Но позволительно ли при размещении графовых данных в реляционной структуре выполнять к ним запросы с помощью SQL?

Ответ: да, хотя и не без проблем. В реляционной БД нужные для запроса соединения обычно известны заранее. В графовом же запросе может понадобиться обойти некоторое число ребер, прежде чем будет найдена искомая вершина, то есть количество соединений заранее не фиксировано.

В нашем примере это происходит в правиле `() -[:WITHIN*0..]-> ()` запроса Cypher. Ребро `LIVES_IN` конкретного человека может указывать на любой тип географического объекта: улицу, город, район, область, штат и т. д. Город может находиться в (`WITHIN`) области, область в штате, штат — в стране и т. д. Ребро `LIVES_IN` может указывать непосредственно на искомую вершину-местоположение, а может и находиться на расстоянии нескольких уровней иерархии местоположений.

В Cypher выражение `:WITHIN*0..` отражает это весьма буквально: оно означает «Следуй по ребру `WITHIN` ноль или более раз». Оно напоминает оператор * в регулярных выражениях.

Начиная с SQL:1999, идею путей обхода переменной длины в запросе можно выразить с помощью синтаксиса *рекурсивных обобщенных табличных выражений* (recursive common table expression) — синтаксиса `WITH RECURSIVE`. В примере 2.5 показан тот же запрос — поиск имен эмигрировавших из США в Европу, — выраженный на языке SQL с использованием этого приема (поддерживается в базах данных PostgreSQL, IBM DB2, Oracle и SQL Server). Однако такой синтаксис выглядит весьма неуклюже по сравнению с Cypher.

Пример 2.5. Тот же запрос, что и в примере 2.4, выраженный на языке SQL с помощью рекурсивных обобщенных табличных выражений

```
WITH RECURSIVE
  -- in_usa — множество идентификаторов вершин для всех
  -- расположенных в США географических объектов
  in_usa(vertex_id) AS (
      SELECT vertex_id FROM vertices WHERE properties->>'name' = 'United States' (1)
    UNION
      SELECT edges.tail_vertex FROM edges (2)
        JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
        WHERE edges.label = 'within'
  ),
  -- in_europe — множество идентификаторов вершин для всех
  -- расположенных в Европе географических объектов
  in_europe(vertex_id) AS (
      SELECT vertex_id FROM vertices WHERE properties->>'name' = 'Europe' (3)
    UNION
      SELECT edges.tail_vertex FROM edges
        JOIN in_europe ON edges.head_vertex = in_europe.vertex_id
        WHERE edges.label = 'within'
  ),
  -- born_in_usa — множество идентификаторов вершин
  -- для всех родившихся в США людей
  born_in_usa(vertex_id) AS ( (4)
    SELECT edges.tail_vertex FROM edges
      JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
      WHERE edges.label = 'born_in'
  ),
  -- lives_in_europe — множество идентификаторов вершин
  -- для всех живущих в Европе людей
  lives_in_europe(vertex_id) AS ( (5)
    SELECT edges.tail_vertex FROM edges
      JOIN in_europe ON edges.head_vertex = in_europe.vertex_id
      WHERE edges.label = 'lives_in'
  )
SELECT vertices.properties->>'name'
FROM vertices
-- Выполняем соединение для поиска людей, родившихся в США
-- *и* живущих в Европе
JOIN born_in_usa     ON vertices.vertex_id = born_in_usa.vertex_id (6)
JOIN lives_in_europe ON vertices.vertex_id = lives_in_europe.vertex_id;
```

(1) Сначала находим вершину, значение свойства `name` которой равно `"United States"`, и делаем ее первым элементом множества вершин `in_usa`.

(2) Проходим по всем входящим ребрам `within` от вершин из множества `in_usa` и добавляем их в то же множество до тех пор, пока не посетим все входящие ребра `within`.

(3) Проделываем то же самое с вершиной, значение свойства `name` которой равно `"Europe"`, и формируем множество вершин `in_europe`.

4. Для каждой вершины из множества `in_usa` проходим по всем входящим ребрам `born_in`, чтобы найти всех людей, родившихся где-то в США.

5. Аналогично для каждой вершины из множества `in_europe` проходим по всем входящим ребрам `lives_in`, чтобы найти всех людей, живущих в Европе.

6. Наконец, выполняем пересечение множества людей, родившихся в США, со множеством людей, живущих в Европе, соединяя их.

Один и тот же запрос может быть написан в четыре строки на одном языке запросов, но требует 29 строк на другом. Такое положение дел демонстрирует, что различные модели данных адаптированы для разных сценариев использования. Поэтому важно выбрать подходящую для конкретного приложения модель.

Хранилища тройных кортежей и SPARQL

Модель тройных кортежей практически эквивалентна модели графов свойств и использует разные слова для описания одних и те же идей. Тем не менее имеет смысл ее обсудить, поскольку для хранилищ тройных кортежей существуют различные утилиты и языки, которые могут быть полезны при создании приложений.

В хранилище тройных кортежей вся информация хранится в форме очень простых трехкомпонентных высказываний: *(субъект, предикат, объект)*. Например, в тройном кортеже *(Джим, любит, бананы) Джим* — субъект (подлежащее), *любит* — предикат (сказуемое), а *бананы* — объект (дополнение).

Субъект тройного кортежа эквивалентен вершине графа. Объект представляет собой одну из двух вещей.

1. Значение простого типа данных, например строчного или числового. В этом случае предикат и объект тройного кортежа эквивалентны ключу и значению свойства вершины-субъекта. Например, *(lucy, age, 33)* эквивалентно вершине *lucy* со свойствами `{"age":33}`.
2. Другую вершину графа. В этом случае предикат представляет собой ребро графа, субъект — начальную вершину, а объект — конечную вершину. Например, в тройном кортеже *(lucy, marriedTo, alain)* субъект и объект — *lucy* и *alain* — оба представляют собой вершины, а предикат *marriedTo* — метку на соединяющем их ребре.

В примере 2.6 показаны те же данные, что и в примере 2.3, переписанные в виде тройных кортежей в формате *Turtle* — подмножестве так называемой *Нотации 3* (Notation3, N3) [39].

Пример 2.6. Подмножество данных с рис. 2.5, представленное в виде тройных кортежей в формате Turtle

```
@prefix : <urn:example:>.
_:lucy     a       :Person.
_:lucy     :name   "Lucy".
_:lucy     :bornIn _:idaho.
_:idaho    a       :Location.
_:idaho    :name   "Idaho".
_:idaho    :type   "state".
_:idaho    :within _:usa.
_:usa      a       :Location.
_:usa      :name   "United States".
_:usa      :type   "country".
_:usa      :within _:namerica.
_:namerica a       :Location.
_:namerica :name   "North America".
_:namerica :type   "continent".
```

Здесь вершины графа записаны в виде `_:некое_название`. Это название ничего не значит за пределами данного файла, оно существует только потому, что иначе мы бы не знали, какие тройные кортежи относятся к одной и той же вершине. Допустим, предикат представляет собой ребро, тогда объект — это вершина, как в `_:idaho :within _:usa`. В случае же когда предикат представляет собой свойство, объект — это строчный литерал, как в `_:usa :name "United States"`.

Повторять один и тот же субъект снова и снова довольно скучно, но, к счастью, можно воспользоваться точкой с запятой, чтобы указать несколько относящихся к одному субъекту вещей. Благодаря этому формат Turtle становится достаточно приятным и удобочитаемым (пример 2.7).

Пример 2.7. Более сжатый способ записи данных из примера 2.6

```
@prefix : <urn:example:>.
_:lucy     a :Person;   :name "Lucy";          :bornIn _:idaho.
_:idaho    a :Location; :name "Idaho";         :type "state";   :within _:usa.
_:usa      a :Location; :name "United States"; :type "country"; :within _:namerica.
_:namerica a :Location; :name "North America"; :type "continent".
```

Семантическая паутина

Если вы захотите прочесть больше литературы о хранилищах тройных кортежей, то легко можете утонуть в водовороте статей о *семантической паутине* (semantic web). Модель данных тройных кортежей совершенно независима от семантической паутины: например, Datomic [40] — это хранилище тройных кортежей, которое вообще не упоминает о какой-либо связи с ней[1]. Но раз уж в представлении многих людей они так тесно связаны, мы ее кратко обсудим.

[1] Формально в Datomic используются пятерные, а не тройные кортежи, два дополнительных поля содержат метаданные для управления версиями.

Семантическая паутина, по сути, представляет собой простую и разумную идею: сайты уже публикуют информацию в удобном для чтения людьми виде — в виде текста и картинок, так почему бы им не публиковать и информацию в формате, удобном для чтения компьютерами? *Resource Description Framework* (RDF, фреймворк описания ресурсов) [41] был задуман как механизм публикации данных различными сайтами в едином формате, благодаря чему данные с различных сайтов могли бы автоматически объединяться в *паутину данных* — своеобразную «всеобщую базу данных» Интернета.

К сожалению, значение семантической паутины было сильно переоценено в начале 2000-х годов, но она до сих пор не была реализована на практике, из-за чего многие начали в ней сомневаться. Она сильно пострадала и от головокружительного обилия аббревиатур, предложенных слишком сложных стандартов и завышенных оценок.

Но стоит взглянуть сквозь пальцы на эти неудачи — и окажется, что из проекта семантической паутины получилось немало хорошего. Тройные кортежи — неплохая внутренняя модель данных для приложения, даже если вы не собираетесь публиковать RDF-данные в семантической паутине.

Модель данных RDF

Использованный нами в примере 2.7 язык Turtle представляет собой удобочитаемый для человека вариант данных модели RDF. Иногда RDF записывается в формате XML — то же самое, но в более развернутом виде (пример 2.8). Формат Turtle/N3 предпочтительнее, как более приятный для глаз. Такие утилиты, как Apache Jena [42], умеют при необходимости выполнять преобразование между различными форматами модели RDF.

Пример 2.8. Данные из примера 2.7, выраженные с помощью синтаксиса RDF/XML

```
<rdf:RDF xmlns="urn:example:"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

  <Location rdf:nodeID="idaho">
    <name>Idaho</name>
    <type>state</type>
    <within>
      <Location rdf:nodeID="usa">
        <name>United States</name>
        <type>country</type>
        <within>
          <Location rdf:nodeID="namerica">
            <name>North America</name>
            <type>continent</type>
          </Location>
        </within>
      </Location>
```

```
    </within>
  </Location>

  <Person rdf:nodeID="lucy">
    <name>Lucy</name>
    <bornIn rdf:nodeID="idaho"/>
  </Person>
</rdf:RDF>
```

У RDF есть несколько специфических особенностей в силу его нацеленности на обмен данными в масштабе всего Интернета. Субъект, предикат и объект тройного кортежа часто представляют собой URI. Например, предикат может быть вот таким URI: `<http://my-company.com/namespace#within>` или вот таким: `<http://my-company.com/namespace#lives_in>`, а не просто `WITHIN` или `LIVES_IN`. Такой вариант обоснован тем, что должна быть возможность комбинировать ваши данные с чужими, и если в чужих данных смысл слов `WITHIN` или `LIVES_IN` отличается, то конфликта не произойдет, поскольку на самом деле их предикаты — `<http://other.org/foo#within>` и `<http://other.org/foo#lives_in>`.

URL `<http://my-company.com/namespace>` не обязательно должен разрешаться — с точки зрения модели RDF это просто пространство имен. Чтобы избежать потенциально путаницы с URL типа `http://`, здесь в примерах используются неразрешаемые URI вида `urn:example:within`. К счастью, данный префикс можно задать один раз вверху файла и забыть про него.

Язык запросов SPARQL

SPARQL — язык запросов для хранилищ тройных кортежей, использующих модель данных RDF (это аббревиатура для *SPARQL Protocol and RDF Query Language* (Протокол SPARQL и язык запросов RDF), произносится ['spɑ:kl]). Он предшествовал Cypher и, поскольку поиск по шаблону языка Cypher заимствован из него, они весьма схожи [37].

На языке SPARQL можно выразить тот же запрос — поиск людей, эмигрировавших из США в Европу, — даже еще более сжатым образом, чем на Cypher (пример 2.9).

Пример 2.9. Запрос из примера 2.4 на языке SPARQL

```
PREFIX : <urn:example:>

SELECT ?personName WHERE {
  ?person :name ?personName.
  ?person :bornIn  / :within* / :name "United States".
  ?person :livesIn / :within* / :name "Europe".
}
```

Структура почти такая же. Следующие два выражения эквивалентны (переменные в языке SPARQL начинаются с вопросительного знака):

```
(person) -[:BORN_IN]-> () -[:WITHIN*0..]-> (location)   # Cypher

?person :bornIn / :within* ?location.                   # SPARQL
```

В силу того что модель RDF не различает свойства и ребра, а просто использует предикаты и для тех и для других, можно задействовать для поиска соответствия свойств тот же синтаксис. В следующем выражении переменная `usa` привязана ко всем вершинам, значение свойства `name` которых равно `"United States"`:

```
(usa {name:'United States'})   # Cypher

?usa :name "United States".    # SPARQL
```

SPARQL — прекрасный язык запросов. Хотя семантическая паутина так и не была реализована, он отлично подходит для внутреннего использования в приложениях.

Графовые базы данных в сравнении с сетевой моделью

В подразделе «Повторяется ли история в случае документоориентированных баз данных» раздела 2.1 мы обсудили способ решения проблемы связей «многие-ко-многим» в IMS моделью CODASYL и реляционной моделью. На первый взгляд, сетевая модель CODASYL похожа на графовую. Так, может, графовые БД — завуалированное второе пришествие CODASYL?

Нет. Они различаются несколькими важными нюансами.

- В CODASYL у БД есть схема, определяющая, какие типы записей могут быть вложены в записи других типов. В графовых базах данных подобных ограничений нет: любая вершина может быть соединена ребром с любой другой. Это позволяет приложениям гораздо более гибко адаптироваться к меняющимся требованиям.
- В CODASYL единственный способ добраться до конкретной записи — пройти по одному из путей доступа к ней. В графовой БД можно сослаться непосредственно на любую вершину по ее уникальному ID или воспользоваться индексом для поиска вершины с конкретным значением.
- В CODASYL потомки записи были упорядоченным множеством, вследствие чего базе данных приходилось поддерживать это упорядочение (что оказывало свое влияние на структуру хранилища), а вставлявшим в БД новые записи приложениям необходимо было заботиться о расположении записей в этих множествах. В графовой БД вершины и ребра не упорядочены (можно отсортировать только результаты выполнения запроса).
- В CODASYL все запросы были императивны, сложны в написании и легко могли быть нарушены при внесении изменений в схему. В графовой базе данных при желании можно написать императивный код обхода, но большинство графовых БД поддерживают и высокоуровневые языки запросов, такие как Cypher или SPARQL.

Фундамент: Datalog

Datalog — гораздо более старый язык, чем Cypher или SPARQL, он широко изучался в вузах в 1980-х годах [44–46]. Он не так широко известен среди разработчиков, но тем не менее очень важен, поскольку является фундаментом для последующих языков запросов.

На практике Datalog используется лишь в немногих информационных системах: например, он является языком запросов БД Datomic [40], а язык Cascalog [47] представляет собой реализацию Datalog, предназначенную для выполнения запросов к большим наборам данных в Hadoop[1].

Модель данных Datalog аналогична модели хранилищ тройных кортежей, разве что немного обобщенной. Вместо написания тройного кортежа в виде `(субъект, предикат, объект)` он записывается как `предикат(субъект, объект)`. В примере 2.10 показано, как записать данные из нашего примера на языке Datalog.

Пример 2.10. Подмножество данных из примера 2.5, представленное в виде фактов языка Datalog

```
name(namerica, 'North America').
type(namerica, continent).

name(usa, 'United States').
type(usa, country).
within(usa, namerica).

name(idaho, 'Idaho').
type(idaho, state).
within(idaho, usa).

name(lucy, 'Lucy').
born_in(lucy, idaho).
```

Теперь, после описания данных, можно написать уже виденный нами запрос, как показано в примере 2.11. Он выглядит немного иначе, чем эквивалентный запрос на Cypher или SPARQL, но пусть это вас не смущает. Datalog — подмножество языка Prolog, с которым вы уже могли сталкиваться, если изучали компьютерные науки.

Пример 2.11. Тот же запрос, что и в примере 2.4, на языке Datalog

```
within_recursive(Location, Name) :- name(Location, Name).   /* Правило 1 */
within_recursive(Location, Name) :- within(Location, Via),  /* Правило 2 */
                                    within_recursive(Via, Name).
```

[1] Datomic и Cascalog используют для Datalog синтаксис S-выражений. В следующих примерах мы будем применять несколько более удобочитаемый синтаксис Prolog — никакого функционального различия между ними нет.

```
migrated(Name, BornIn, LivingIn) :- name(Person, Name),     /* Правило 3 */
                                    born_in(Person, BornLoc),
                                    within_recursive(BornLoc, BornIn),
                                    lives_in(Person, LivingLoc),
                                    within_recursive(LivingLoc, LivingIn).
?- migrated(Who, 'United States', 'Europe').
/* Who = 'Lucy'. */
```

Cypher и SPARQL сразу начинают с `SELECT`, но Datalog движется более мелкими шажками. Сначала мы описываем *правила*, информирующие БД о новых предикатах: в нашем случае это два новых предиката, `within_recursive` и `migrated`. Такие предикаты унаследованы от других правил или данных, а не являются хранящимися в БД тройными кортежами. Правила могут ссылаться на другие правила, аналогично тому, как функции могут вызывать другие функции или рекурсивно вызывать себя. Аналогичным образом сложные запросы можно создавать по частям.

В правилах слова, начинающиеся с буквы в верхнем регистре, являются переменными, а соответствие предикатам ищется аналогично Cypher и SPARQL. Например, для правила `name(Location, Name)` находится тройной кортеж `name(namerica, 'North America')` с привязками переменных `Location = namerica` и `Name = 'North America'`.

Правило работает, если системе удается найти соответствие для *всех* предикатов справа от оператора `:-`. Применение правила аналогично добавлению в базу данных левой части оператора `:-` (где переменные заменены найденными для них значениями).

Один из возможных способов применения правил.

1. Тройной кортеж `name(namerica, 'North America')` существует в базе данных, так что правило 1 применимо. Генерируется предикат `within_recursive(namerica, 'North America')`.
2. Предикат `within(usa, namerica)` существует в базе данных, а на предыдущем шаге сгенерирован предикат `within_recursive(namerica, 'North America')`, так что правило 2 применимо. Генерируется предикат `within_recursive(usa, 'North America')`.
3. Предикат `within(idaho, usa)` существует в базе данных, а на предыдущем шаге сгенерирован предикат `within_recursive(usa, 'North America')`, так что правило 2 применимо. Генерируется предикат `within_recursive(idaho, 'North America')`.

Путем многократного применения правил 1 и 2 предикат `within_recursive` возвращает все содержащиеся в нашей базе данных географические пункты в Северной Америке (или в любом другом месте). Этот процесс показан на рис. 2.6.

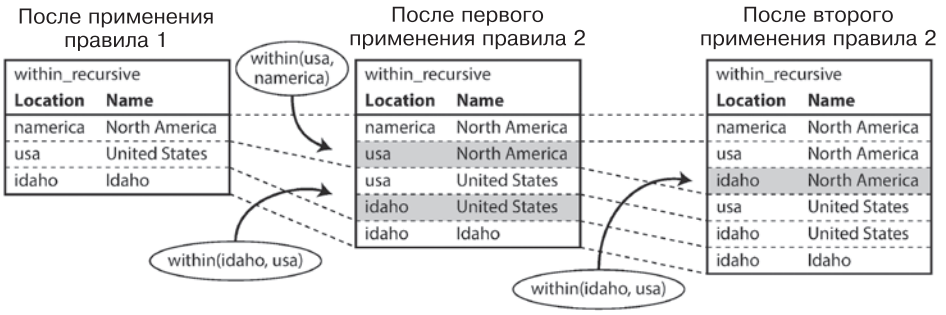


Рис. 2.6. Выясняем, что Айдахо (Idaho) находится в Северной Америке, с помощью правил Datalog из примера 2.11

Теперь, задействуя правило 3, мы можем найти людей, родившихся в некоем географическом пункте `BornIn` и живущих в некоем географическом пункте `LivingIn`. С помощью запроса с `BornIn = 'United States'` и `LivingIn = 'Europe'`, оставив людей в виде переменной `Who`, мы просим систему Datalog найти, какие значения может принимать переменная `Who`. И в итоге получаем тот же результат, что и вышеприведенные запросы Cypher и SPARQL.

Подход Datalog требует иного типа мышления по сравнению с другими языками запросов, обсуждавшимися выше. Тем не менее он очень многообещающ, поскольку правила можно комбинировать и повторно использовать в различных запросах. Для простых одноразовых запросов это менее удобно, но весьма полезно в случае сложных данных.

2.4. Резюме

Модели данных — обширная тема для обсуждения, и в настоящей главе мы лишь вкратце рассмотрели несколько различных моделей. У нас было недостаточно места для подробного изучения каждой модели, но надеюсь, что этого обзора было достаточно, чтобы разжечь ваш интерес и желание узнать больше о моделях, лучше всего соответствующих требованиям приложения.

Исторически сначала данные представлялись в виде одного большого дерева (иерархическая модель), но такой вариант плохо подходил для представления связей «многие-ко-многим» и для решения этой проблемы была придумана реляционная модель. А немного позже разработчики обнаружили, что она не слишком хорошо подходит для некоторых приложений. Новые нереляционные хранилища данных NoSQL делятся на две основные разновидности.

1. *Документоориентированные БД* предназначены для тех сценариев использования, при которых данные поступают в виде отдельных документов и связи между документами редки.

2. *Графовые БД* нацелены в противоположном направлении: они предназначены для сценариев применения, в которых любые данные потенциально могут быть взаимосвязаны.

Все три модели (документная, реляционная и графовая) в настоящее время широко используются, и все отлично работают в своих предметных областях. Одну модель можно эмулировать на языке другой (например, графовые данные разместить в реляционной БД), но результат этого часто достаточно неудобен в применении. Именно поэтому для разных целей прибегают к разным системам, а не к единому универсальному решению.

Одна из общих черт документоориентированных и графовых БД — отсутствие обязательной схемы для хранимых данных, что облегчает адаптацию приложений к изменению требований. Однако чаще всего приложение предполагает наличие какой-либо структуры данных, вопрос только в том, является ли схема явной (навязываемой при записи) или неявной (контролируемой при чтении).

У каждой модели данных есть собственный язык запросов или фреймворк, и мы рассмотрели несколько их примеров: SQL, MapReduce, конвейер агрегирования MongoDB, Cypher, SPARQL и Datalog. Мы также коснулись вопроса CSS и XSL/XPath, не являющихся языками запросов БД, но имеющих с ними интересные параллели.

Хотя мы охватили множество вопросов, немало моделей данных не было упомянуто. Вот несколько примеров.

- Исследователям, работающим с геномными данными, часто приходится выполнять *поиск сходства последовательностей*, означающий сравнение очень длинной строки (соответствующей молекуле ДНК) со схожими, но не идентичными строками из большой базы данных. Ни одна из описанных выше БД не способна справиться с этим, вследствие чего исследователями было написано такое специализированное ПО для геномной базы данных, как GenBank [48].
- Специалисты по физике элементарных частиц занимаются широкомасштабным анализом информации в стиле «больших данных» уже десятилетиями, и такие проекты, как Большой адронный коллайдер (Large Hadron Collider, LHC), имеют дело сейчас с сотнями петабайт! При таких масштабах необходимы индивидуальные программные решения, чтобы стоимость аппаратного обеспечения не зашкаливала [49].
- *Полнотекстовый поиск* — вид модели данных, часто используемый в БД. Информационный поиск — большая отдельная тема, которую мы не станем рассматривать очень подробно в этой книге, хотя коснемся поисковых индексов в главе 3 и части III.

На этом завершим обсуждение. В следующей главе мы рассмотрим некоторые плюсы и минусы *реализации* описанных в настоящей главе моделей данных.

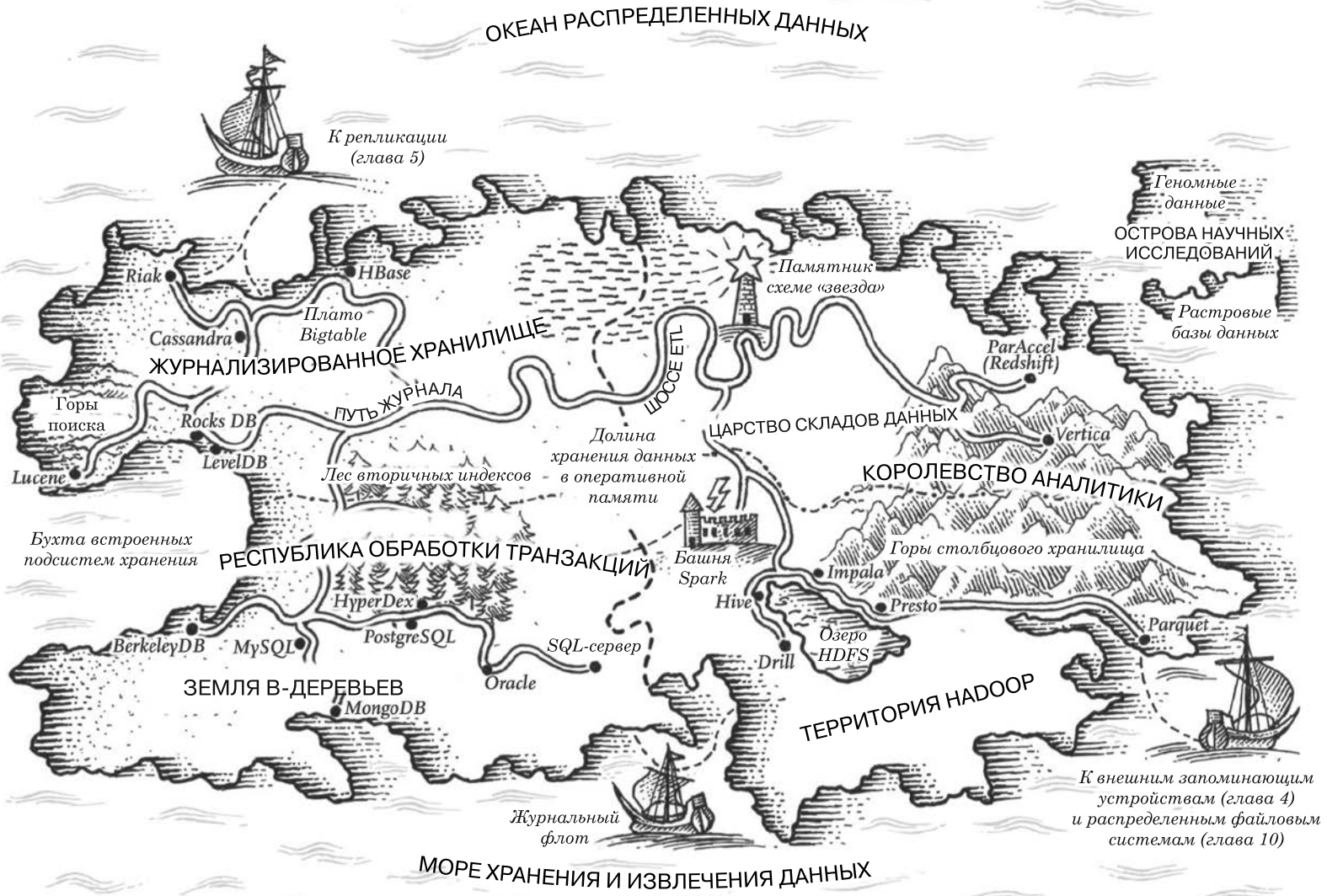
2.5. Библиография

1. *Codd E. F.* A Relational Model of Data for Large Shared Data Banks // Communications of the ACM, volume 13, number 6, pages 377–387, June 1970 [Электронный ресурс]. — Режим доступа: https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf.
2. *Stonebraker M., Hellerstein J. M.* What Goes Around Comes Around // Readings in Database Systems, 4th edition, MIT Press, pages 2–41, 2005. http://mitpress2.mit.edu/books/chapters/0262693143chapm1.pdf.
3. *Sadalage P. J., Fowler M.* NoSQL Distilled. Addison-Wesley, August 2012.
4. *Evans E.* NoSQL: What's in a Name? October 30, 2009 [Электронный ресурс]. — Режим доступа: http://blog.sym-link.com/2009/10/30/nosql_whats_in_a_name.html.
5. *Phillips J.* Surprises in Our NoSQL Adoption Survey. February 8, 2012 [Электронный ресурс]. — Режим доступа: https://blog.couchbase.com/nosql-adoption-survey-surprises/.
6. *Wagner M.* SQL/XML:2006 — Evaluierung der Standardkonformität ausgewählter Datenbanksysteme. — Diplomica Verlag, Hamburg, 2010.
7. XML Data in SQL Server // SQL Server 2012 Documentation, 2013 [Электронный ресурс]. — Режим доступа: https://docs.microsoft.com/en-us/sql/relational-databases/xml/xml-data-sql-server.
8. PostgreSQL 9.3.1 Documentation // The PostgreSQL Global Development Group, 2013 [Электронный ресурс]. — Режим доступа: https://www.postgresql.org/docs/9.3/static/index.html.
9. The MongoDB 2.4 Manual // MongoDB, Inc., 2013 [Электронный ресурс]. — Режим доступа: https://docs.mongodb.com/manual/.
10. RethinkDB 1.11 Documentation, 2013 [Электронный ресурс]. — Режим доступа: https://rethinkdb.com/docs/.
11. Apache CouchDB 1.6 Documentation, 2014 [Электронный ресурс]. — Режим доступа: http://docs.couchdb.org/en/latest/.
12. *Qiao L., Surlaker K., Das S., et al.* On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform // ACM International Conference on Management of Data (SIGMOD), June 2013 [Электронный ресурс]. — Режим доступа: https://www.slideshare.net/amywtang/espresso-20952131.
13. *Long R., Harrington M., Hain R., Nicholls G.* IMS Primer // IBM Redbook SG24-5352-00, IBM International Technical Support Organization, January 2000 [Электронный ресурс]. — Режим доступа: http://www.redbooks.ibm.com/redbooks/pdfs/sg245352.pdf.
14. *Bartlett S. D.* IBM's IMS — Myths, Realities, and Opportunities // The Clipper Group Navigator, TCG2013015LI, July 2013 [Электронный ресурс]. — Режим доступа: ftp://public.dhe.ibm.com/software/data/ims/pdf/TCG2013015LI.pdf.

15. *Mei S.* Why You Should Never Use MongoDB. November 11, 2013 [Электронный ресурс]. — Режим доступа: http://www.sarahmei.com/blog/2013/11/11/why-you-should-never-use-mongodb/.
16. *Knowles J. S., Bell D. M. R.* The CODASYL Model // Databases — Role and Structure: An Advanced Course, edited by P. M. Stocker, P. M. D. Gray, and M. P. Atkinson. — Cambridge University Press, 1984. — Pages 19–56.
17. *Bachman C. W.* The Programmer as Navigator // Communications of the ACM, volume 16, number 11, pages 653–658, November 1973 [Электронный ресурс]. — Режим доступа: https://dl.acm.org/citation.cfm?id=362534.
18. *Hellerstein J. M., Stonebraker M., Hamilton J.* Architecture of a Database System // Foundations and Trends in Databases, volume 1, number 2, pages 141–259, November 2007 [Электронный ресурс]. — Режим доступа: http://db.cs.berkeley.edu/papers/fntdb07-architecture.pdf.
19. *Parikh S., Stirman K.* Schema Design for Time Series Data in MongoDB. October 30, 2013 [Электронный ресурс]. — Режим доступа: https://www.mongodb.com/blog/post/schema-design-for-time-series-data-in-mongodb.
20. *Fowler M.* Schemaless Data Structures. January 7, 2013 [Электронный ресурс]. — Режим доступа: https://martinfowler.com/articles/schemaless/.
21. *Awadallah A.* Schema-on-Read vs. Schema-on-Write // Berkeley EECS RAD Lab Retreat, Santa Cruz, CA, May 2009 [Электронный ресурс]. — Режим доступа: https://www.slideshare.net/awadallah/schemaonread-vs-schemaonwrite.
22. *Odersky M.* The Trouble with Types // Strange Loop, September 2013 [Электронный ресурс]. — Режим доступа: https://www.infoq.com/presentations/data-types-issues.
23. *Irwin C.* MongoDB — Confessions of a PostgreSQL Lover // HTML5DevConf, October 2013 [Электронный ресурс]. — Режим доступа: https://speakerdeck.com/conradirwin/mongodb-confessions-of-a-postgresql-lover.
24. Percona Toolkit Documentation: pt-online-schema-change // Percona Ireland Ltd., 2013 [Электронный ресурс]. — Режим доступа: https://www.percona.com/doc/percona-toolkit/2.2/pt-online-schema-change.html.
25. *Keddo R., Bielohlawek T., Schmidt T.* Large Hadron Migrator // SoundCloud, 2013 [Электронный ресурс]. — Режим доступа: https://github.com/soundcloud/lhm.
26. *Noach S.* gh-ost: GitHub's Online Schema Migration Tool for MySQL. August 1, 2016 [Электронный ресурс]. — Режим доступа: https://githubengineering.com/gh-ost-github-s-online-migration-tool-for-mysql/.
27. *Corbett J. C., Dean J., Epstein M., et al.* Spanner: Google's Globally-Distributed Database // 10th USENIX Symposium on Operating System Design and Implementation (OSDI), October 2012 [Электронный ресурс]. — Режим доступа: https://research.google.com/archive/spanner.html.
28. *Burleson D. K.* Reduce I/O with Oracle Cluster Tables [Электронный ресурс]. — Режим доступа: http://www.dba-oracle.com/oracle_tip_hash_index_cluster_table.htm.

29. *Chang F., Dean J., Ghemawat S., et al.* Bigtable: A Distributed Storage System for Structured Data // 7th USENIX Symposium on Operating System Design and Implementation (OSDI), November 2006 [Электронный ресурс]. — Режим доступа: https://research.google.com/archive/bigtable.html.
30. *Cochrane B. J., McKnight K. A.* DB2 JSON Capabilities, Part 1: Introduction to DB2 JSON," IBM developerWorks, June 20, 2013 [Электронный ресурс]. — Режим доступа: https://www.ibm.com/developerworks/data/library/techarticle/dm-1306nosqlforjson1/.
31. *Sutter H.* The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software // Dr. Dobb's Journal, volume 30, number 3, pages 202–210, March 2005 [Электронный ресурс]. — Режим доступа: http://www.gotw.ca/publications/concurrency-ddj.htm.
32. *Hellerstein Joseph M.* The Declarative Imperative: Experiences and Conjectures in Distributed Logic // Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech report UCB/EECS-2010-90, June 2010 [Электронный ресурс]. — Режим доступа: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-90.pdf.
33. *Dean J., Ghemawat S.* MapReduce: Simplified Data Processing on Large Clusters // 6th USENIX Symposium on Operating System Design and Implementation (OSDI), December 2004 [Электронный ресурс]. — Режим доступа: https://research.google.com/archive/mapreduce.html.
34. *Kerstiens C.* JavaScript in Your Postgres. June 5, 2013 [Электронный ресурс]. — Режим доступа: https://blog.heroku.com/javascript_in_your_postgres.
35. *Bronson N., Amsden Z., Cabrera G., et al.* TAO: Facebook's Distributed Data Store for the Social Graph // USENIX Annual Technical Conference (USENIX ATC), June 2013 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson.
36. Apache TinkerPop3.2.3 Documentation. October 2016 [Электронный ресурс]. — Режим доступа: http://tinkerpop.apache.org/docs/3.2.3/reference/.
37. The Neo4j Manual v2.0.0 // Neo Technology, 2013 [Электронный ресурс]. — Режим доступа: http://neo4j.com/docs/2.0.0/index.html.
38. *Eifrem E.* Twitter correspondence. January 3, 2014 [Электронный ресурс]. — Режим доступа: https://twitter.com/emileifrem/status/419107961512804352.
39. *Beckett D., Berners-Lee T.* Turtle — Terse RDF Triple Language // W3C Team Submission, March 28, 2011 [Электронный ресурс]. — Режим доступа: https://www.w3.org/TeamSubmission/turtle/.
40. Datomic Development Resources // Metadata Partners, LLC, 2013 [Электронный ресурс]. — Режим доступа: http://docs.datomic.com/
41. W3C RDF Working Group // Resource Description Framework (RDF). w3.org, 10 February 2004 [Электронный ресурс]. — Режим доступа: https://www.w3.org/RDF/.

42. Apache Jena // Apache Software Foundation [Электронный ресурс]. — Режим доступа: http://jena.apache.org/.
43. *Harris S., Seaborne A., Prud'hommeaux E.* SPARQL 1.1 Query Language // W3C Recommendation, March 2013 [Электронный ресурс]. — Режим доступа: https://www.w3.org/TR/sparql11-query/.
44. *Green T. J., Huang S. S., Loo B. T., Zhou W.* Datalog and Recursive Query Processing // Foundations and Trends in Databases, volume 5, number 2, pages 105–195, November 2013 [Электронный ресурс]. — Режим доступа: http://blogs.evergreen.edu/sosw/files/2014/04/Green-Vol5-DBS-017.pdf.
45. *Ceri S. Gottlob G., Tanca L.* What You Always Wanted to Know About Datalog (And Never Dared to Ask) // IEEE Transactions on Knowledge and Data Engineering, volume 1, number 1, pages 146–166, March 1989 [Электронный ресурс]. — Режим доступа: https://www.researchgate.net/profile/Letizia_Tanca/publication/3296132_What_you_Always_Wanted_to_Know_About_Datalog_And_Never_Dared_to_Ask/links/0fcfd50ca2d20473ca000000/What-you-Always-Wanted-to-Know-About-Datalog-And-Never-Dared-to-Ask.pdf.
46. *Abiteboul S., Hull R., Vianu V.* Foundations of Databases. — Addison-Wesley, 1995. webdam.inria.fr/Alice.
47. *Marz N.* Cascalog [Электронный ресурс]. — Режим доступа: cascalog.org.
48. *Benson D. A., Karsch-Mizrachi I., Lipman D. J., et al.:* GenBank // Nucleic Acids Research, volume 36, Database issue, pages D25–D30, December 2007 [Электронный ресурс]. — Режим доступа: https://academic.oup.com/nar/article/36/suppl_1/D25/2507746/GenBank.
49. *Rademakers F.* ROOT for Big Data Analysis // Workshop on the Future of Big Data Management, London, UK, June 2013 [Электронный ресурс]. — Режим доступа: https://indico.cern.ch/event/246453/contributions/1566610/attachments/423154/587535/ROOT-BigData-Analysis-London-2013.pdf.


ОКЕАН РАСПРЕДЕЛЕННЫХ ДАННЫХ
К репликации (глава 5)
Геномные данные
ОСТРОВА НАУЧНЫХ ИССЛЕДОВАНИЙ
Растровые базы данных
Riak
HBase
Памятник схеме «звезда»
Плато Bigtable
Cassandra
ЖУРНАЛИЗИРОВАННОЕ ХРАНИЛИЩЕ
ШОССЕ ETL
ParAccel (Redshift)
Горы поиска
Rocks DB
ПУТЬ ЖУРНАЛА
ЦАРСТВО СКЛАДОВ ДАННЫХ
Vertica
LevelDB
Lucene
Лес вторичных индексов
Долина хранения данных в оперативной памяти
КОРОЛЕВСТВО АНАЛИТИКИ
Бухта встроенных подсистем хранения
РЕСПУБЛИКА ОБРАБОТКИ ТРАНЗАКЦИЙ
Башня Spark
Горы столбцового хранилища
Impala
HyperDex
Hive
Presto
Parquet
BerkeleyDB
MySQL
PostgreSQL
SQL-сервер
Озеро HDFS
Drill
Oracle
ЗЕМЛЯ В-ДЕРЕВЬЕВ
ТЕРРИТОРИЯ HADOOP
MongoDB
К внешним запоминающим устройствам (глава 4) и распределенным файловым системам (глава 10)
Журнальный флот
МОРЕ ХРАНЕНИЯ И ИЗВЛЕЧЕНИЯ ДАННЫХ

3 Подсистемы хранения и извлечение данных

Wer Ordnung hält, ist nur zu faul zum Suchen.

(Если вы поддерживаете в вещах порядок, значит, вы просто слишком ленивы, чтобы их искать.)

Немецкая поговорка

По существу, база данных должна решать две задачи: сохранять при получении от вас данные и позднее предоставлять их вам по запросу.

В главе 2 мы обсудили модели данных и языки запросов, то есть формат, в котором вы (разработчик приложений) передаете данные в базу, а также механизм их дальнейшего запроса. В этой главе мы обсудим то же самое с точки зрения БД: как сохранить полученные от пользователя данные и как найти их снова в случае запроса.

Почему вас как разработчика приложений должны волновать внутренние нюансы того, как БД хранит данные и как она их находит? Вряд ли вы собираетесь реализовать собственную подсистему хранения данных с нуля, но вам определенно *нужно* выбрать из множества существующих подсистему хранения, подходящую именно для вашего приложения. Чтобы настроить его на оптимальную работу при вашей нагрузке, не помешает иметь хотя бы приблизительное представление о том, каковы внутренние механизмы функционирования подсистемы хранения.

В частности, существуют немалые отличия между подсистемами хранения, оптимизированными на транзакционные нагрузки и оптимизированными для аналитики. Мы рассмотрим эти различия позднее, в разделе 3.2, а в разделе 3.3 обсудим семейство подсистем хранения, оптимизированных для аналитики.

Однако начну я эту главу с представления подсистем хранения, использующихся в таких базах, которые вам уже, вероятно, знакомы: традиционные реляционные БД,

а также большинство так называемых баз данных NoSQL. Мы исследуем два семейства подсистем хранения: *журналированные* (log-structured storage engine) и *постраничные* (page-oriented storage engine), например B-деревья.

3.1. Базовые структуры данных БД

Рассмотрим самую простую БД в мире, реализованную в виде двух функций командной оболочки Bash:

```
#!/bin/bash

db_set () {
    echo "$1,$2" >> database
}

db_get () {
    grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

Обе функции реализуют хранилище типа «ключ — значение». Можно вызвать команду `db_set key value` для сохранения `key` и `value` в базе данных. Ключ и значение могут быть (почти) всем, чем вы пожелаете, — например, значение может быть JSON-документом. Затем можно вызвать команду `db_get key` для поиска последнего относящегося к искомому ключу значения и его возврата.

И это работает:

```
$ db_set 123456 '{"name":"London","attractions":["Big Ben","London Eye"]}'

$ db_set 42 '{"name":"San Francisco","attractions":["Golden Gate Bridge"]}'

$ db_get 42
{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
```

Лежащий в их основе формат хранения очень прост: он представляет собой текстовый файл, в котором каждая строка содержит пару «ключ — значение», разделенную запятой (похоже на CSV-файл, если не учитывать нюансы экранирования). Каждый вызов функции `db_set` приводит к добавлению данных в конец файла, так что при обновлении ключа несколько раз старые версии значений не будут затерты — для поиска самого последнего значения необходимо найти последнее вхождение ключа в файле (отсюда `tail -n 1` в вызове `db_get`):

```
$ db_set 42 '{"name":"San Francisco","attractions":["Exploratorium"]}'

$ db_get 42
{"name":"San Francisco","attractions":["Exploratorium"]}

$ cat database
123456,{"name":"London","attractions":["Big Ben","London Eye"]}
42,{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
42,{"name":"San Francisco","attractions":["Exploratorium"]}
```

Производительность нашей функции `db_set` на самом деле довольно неплоха для настолько простой реализации, поскольку добавление данных в конец файла обычно весьма эффективно. Многие БД используют очень похожий на `db_set` механизм, *журнал*, представляющий собой файл, предназначенный только для добавления данных в его конец. У настоящих БД гораздо больше забот (управление конкурентным доступом, возврат в обращение пространства на диске, чтобы журнал не рос до бесконечности, а также обработка ошибок и записей, которые были записаны на диск только частично), но основной принцип остается тем же. Журналы исключительно удобны, и они еще не раз встретятся нам в оставшейся части книги.



Слово «журнал» часто используют, говоря о журналах приложения, в которые программа выводит текст, описывающий выполняемые действия. В этой книге оно применяется в более общем смысле: последовательность записей, предназначенная только для добавления в ее конец данных. Журнал не обязательно должен быть удобочитаемым для человека; он может быть двоичным и предназначенным только для чтения другими программами.

С другой стороны, производительность нашей функции `db_get` ужасна в случае большого количества записей в БД. Каждый раз, когда нужно найти ключ, `db_get` приходится просматривать всю базу от начала до конца, выискивая вхождения ключа. Говоря в алгоритмических терминах, сложность поиска — порядка $O(n)$: при удвоении количества записей n в БД поиск занимает вдвое больше времени. Это не очень хорошо.

Для эффективного поиска значения конкретного ключа в БД необходима другая структура данных: *индекс*. В этой главе мы рассмотрим несколько индексных структур и сравним их. Общая их идея заключается в дополнительном хранении определенных метаданных, служащих своеобразным дорожным указателем, помогающим найти нужные данные. При необходимости поиска одних и те же данных различными способами может понадобиться несколько разных индексов по различным частям данных.

Индекс — *дополнительная* структура, производная от основных данных. Многие БД предоставляют возможность добавлять и удалять индексы без какого-либо воздействия на содержимое базы, это влияет только на производительность запросов. Поддержка дополнительных структур влечет рост накладных расходов, особенно при записи на диск. В последнем случае превзойти производительность простого дописывания данных в конец файла сложно, поскольку это простейшая из возможных операций записи. Любые индексы обычно замедляют запись, так как индекс тоже приходится обновлять всякий раз при записи данных на диск.

Это важный компромисс в системах хранения данных: хорошо подобранные индексы ускоряют запросы на чтение, но замедляют запись. Поэтому БД обычно не индексируют по умолчанию все, что можно, а заставляют вас — разработчика

приложения или администратора БД — выбирать индексы вручную, на основе знания типичных для приложения паттернов запросов. При этом вы можете выбрать те индексы, выгода от которых для приложения максимальна, а накладные расходы не превышают необходимого объема.

Хеш-индексы

Начнем с индексов для данных типа «ключ — значение». Это не единственный тип данных, которые можно индексировать, но один из самых распространенных, а также удобный стандартный блок для построения более сложных индексов.

Хранилища данных типа «ключ — значение» очень схожи с типом *«словарь»*, встречающимся в большинстве языков программирования, реализуемым обычно в виде хеш-карты (hash map)/хеш-таблицы (hash table). Хеш-карты описываются во многих учебниках по алгоритмам [1, 2], так что мы не станем углубляться в подробности их работы. А раз у нас уже используются хеш-карты для структур данных в памяти, то почему бы не задействовать их для индексации данных на диске?

Предположим, что наше хранилище работает только путем добавления в конец файла, как в предыдущем примере. Тогда простейшая стратегия индексации такова: хранить в оперативной памяти хеш-карту, в которой каждому ключу поставлено в соответствие смещение (относительный адрес) в файле данных — место, где находится значение, как показано на рис. 3.1. Всякий раз при добавлении в файл новой пары «ключ — значение» происходит также обновление хеш-карты для отражения в ней относительного адреса только что записанных данных (описанный механизм работает как при вставке новых ключей, так и при обновлении существующих). Если нужно найти значение, то можно воспользоваться хеш-картой для поиска относительного адреса в файле данных, перейти в это место и прочитать значение.

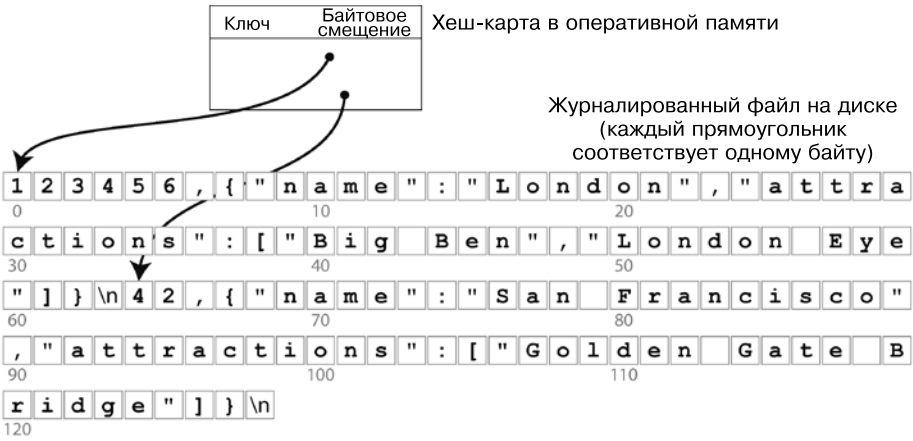


Рис. 3.1. Сохранение журнала пар «ключ — значение» в CSV-подобном формате, индексированном с помощью хеш-карты в оперативной памяти

Представленный подход может показаться некоторым упрощением, но вполне жизнеспособен. Фактически именно так работает Bitcask (подсистема хранения в распределенной NoSQL СУБД Riak) [3]. Bitcask обеспечивает высокопроизводительное чтение и запись, а также соответствие требованию о том, что все ключи должны помещаться в доступной оперативной памяти, поскольку хеш-карта целиком хранится в памяти. Значения могут использовать пространства больше, чем имеется в оперативной памяти, поскольку их можно загрузить с диска за один переход к нужной позиции. Если эта часть данных уже находится в кэше файловой системы, то для ее чтения вообще не потребуется дисковых операций ввода/вывода.

Такая подсистема хранения, как Bitcask, отлично подходит для случаев частого обновления значений для всех ключей. Например, ключ может быть URL видео с кошками, а значение — количество его просмотров (увеличивается каждый раз, когда кто-нибудь нажимает кнопку воспроизведения). При таком типе нагрузки количество операций записи велико, но уникальных ключей не слишком много, то есть количество операций записи на один ключ велико, но вполне можно хранить все ключи в оперативной памяти.

Напомню: мы пока что только дописываем в конец файла. Так как же избежать ситуации исчерпания места на диске? Хорошим решением будет разбить журнал на сегменты определенного размера, закрывая файл сегмента при достижении им определенного размера и записывая последующие данные уже в новый файл. Затем можно выполнить *уплотнение* (compaction) этих сегментов, как показано на рис. 3.2. Уплотнение означает отбрасывание дублирующихся ключей из журнала и сохранение только последней версии данных для каждого ключа.



Рис. 3.2. Уплотнение журнала типа «ключ — значение» (учет количества воспроизведений каждого видео с кошками) с сохранением только последнего значения для каждого ключа

Более того, поскольку уплотнение часто приводит к значительному уменьшению размера сегментов (допустим, если ключ переписывался в среднем несколько раз в пределах одного сегмента), можно также слить несколько сегментов в один во время уплотнения, как показано на рис. 3.3. Сегменты никогда не меняются после записи, так что объединенный сегмент записывается в новый файл. Слияние и уплотнение «замороженных» сегментов можно выполнять в фоновом потоке,

продолжая при этом обслуживать запросы на чтение и запись в обычном режиме, используя старые файлы сегментов. По завершении слияния можно переключить запросы на чтение, чтобы применить новые объединенные сегменты вместо старых, после чего старые файлы сегментов просто удалить.



Рис. 3.3. Одновременное выполнение уплотнения и слияния сегментов

Теперь у каждого сегмента имеется своя хеш-таблица в оперативной памяти, ставящая ключам в соответствие смещение в файле. Чтобы найти значение для ключа, необходимо сначала заглянуть в хеш-карту последнего сегмента: если ключа там нет, то проверяется следующий по времени сегмент и т. д. Благодаря процессу слияния количество сегментов остается небольшим, поэтому при поиске не придется проверять слишком много хеш-карт.

Чтобы воплотить эту простую идею в жизнь, понадобится учесть немало нюансов. Вот краткий перечень вопросов, которые имеют значение при настоящей ее реализации.

- *Формат файлов.* CSV — не лучший формат для журнала. Быстрее и проще будет воспользоваться двоичным форматом, в котором сначала кодируется в байтах длина строки, а затем — строка неформатированных данных (при этом нужды в экранировании нет).
- *Удаление записей.* При необходимости удалить ключ и соответствующее ему значение приходится добавлять в файл данных специальную запись об удалении (иногда называемую *отметкой об удалении* (tombstone)). При слиянии сегментов журнала эта отметка указывает процессу слияния игнорировать все предыдущие значения для удаленного ключа.
- *Восстановление после сбоев.* При перезапуске базы данных находящиеся в оперативной памяти хеш-карты теряются. В принципе, восстановить хеш-карту

сегмента можно, прочитав весь файл сегмента с начала до конца и отмечая в процессе смещение последних значений для каждого ключа. Однако если файлы сегментов велики, то это может занять много времени, что сделает перезапуск сервера неприятной процедурой. Bitcask ускоряет восстановление за счет сохранения на диске копий состояния хеш-карт всех сегментов, которые можно загрузить в память достаточно быстро.

- *Недописанные записи*. Фатальный сбой базы данных может произойти в любую минуту, в том числе и в момент добавления записи в журнал. Файлы Bitcask включают контрольные суммы, что позволяет обнаруживать и игнорировать подобные поврежденные части журналов.
- *Управление конкурентным доступом*. Поскольку записи в журнал добавляются в строго последовательном порядке, в обычной реализации используется ровно один поток записи. Сегменты файлов данных допускают только дописывание, будучи в остальных отношениях неизменяемыми, так что их можно читать параллельно в нескольких потоках выполнения.

На первый взгляд, журналы, допускающие только добавление в конец файла, кажутся довольно неэкономными: почему бы не обновлять файл на месте, заменяя старое значение новым? Но вариант с такими журналами оказывается удачным по нескольким причинам.

- Добавление в конец файла и слияние сегментов — последовательные операции записи, в большинстве случаев выполняющиеся намного быстрее случайной записи, особенно на магнитных жестких дисках с раскручивающимися пластинами. Последовательная запись до некоторой степени предпочтительна и в случае *твердотельных накопителей* (solid state drive, SSD) на основе флеш-памяти [4]. Мы обсудим этот вопрос подробнее в подразделе «Сравнение B- и LSM-деревьев» текущего раздела.
- Конкурентный доступ и восстановление после сбоев сильно упрощаются в случае допускающих только добавление или вообще неизменяемых файлов данных. Например, не нужно заботиться о сбоях во время перезаписи значения, приводящих в результате к файлу, в котором склеены воедино части старого и нового значений.
- Слияние старых сегментов позволяет решить проблему фрагментирования файлов данных с течением времени.

Однако у индексов хеш-таблиц тоже есть ограничения.

- Хеш-таблица должна помещаться в оперативной памяти, так что если у вас очень много ключей, то вам не повезло. В принципе, можно поддерживать хеш-карту на диске, но, к сожалению, добиться хорошей ее производительности непросто. Она требует большого количества операций ввода/вывода с произвольным доступом, ее расширение при заполнении — дорогостоящая операция, а разрешение конфликтов хеша требует запутанной логики [5].

- Запросы по диапазону неэффективны. Например, невозможно с легкостью просмотреть все записи между `kitty00000` и `kitty99999` — необходимо искать каждый ключ отдельно в хеш-картах.

В следующем подразделе мы рассмотрим индексную структуру, у которой нет этих ограничений.

SS-таблицы и LSM-деревья

На рис. 3.3 каждый журналированный сегмент хранения представляет собой последовательность пар «ключ — значение». Эти пары находятся в том порядке, в котором они были записаны, и более поздние значения в журнале имеют приоритет над более ранними значениями для того же ключа. Кроме того, порядок пар «ключ — значение» в файле ни на что не влияет.

Мы собираемся сейчас поменять формат наших файлов сегментов: потребовать, чтобы последовательность пар «ключ — значение» была *отсортирована по ключу*. На первый взгляд, это требование помешает использовать последовательную запись, но мы решим данную проблему чуть позже.

Мы назовем новый формат *отсортированной строковой таблицей* (sorted string table, SSTable), сокращенно — SS-таблицей. Мы потребуем также, чтобы каждый ключ встречался лишь один раз в каждом объединенном файле сегмента (процесс уплотнения сразу гарантирует это). У SS-таблиц есть несколько больших преимуществ перед журнальными сегментами с хеш-индексами.

1. Объединение сегментов выполняется просто и эффективно, даже если размер файлов превышает объем доступной оперативной памяти. Этот подход близок к используемому в алгоритме *сортировки слиянием* (mergesort). Он показан на рис. 3.4: начинаем с одновременного чтения входных файлов, просматриваем первый ключ в каждом из файлов, копируем самый нижний ключ (в соответствии с порядком сортировки) в выходной файл и повторяем эти действия. В результате получаем новый объединенный файл сегмента, также отсортированный по ключу.

   А если один и тот же ключ встретится в нескольких входных сегментах? Как вы помните, каждый сегмент содержит все записанные в базу данных значения за некоторый период времени. Это значит, что все значения одного из входных сегментов должны оказаться более свежими, чем все значения другого (при условии обязательного слияния воедино соседних сегментов). Если несколько сегментов содержат один и тот же ключ, то можно взять значение из наиболее нового сегмента и отбросить значения из более старых.
2. Чтобы найти в файле конкретный ключ, не нужно больше хранить индекс всех ключей в оперативной памяти. Например (рис. 3.5), нам нужен ключ `handiwork`, но мы не знаем точного смещения этого ключа в файле сегмента. Однако нам

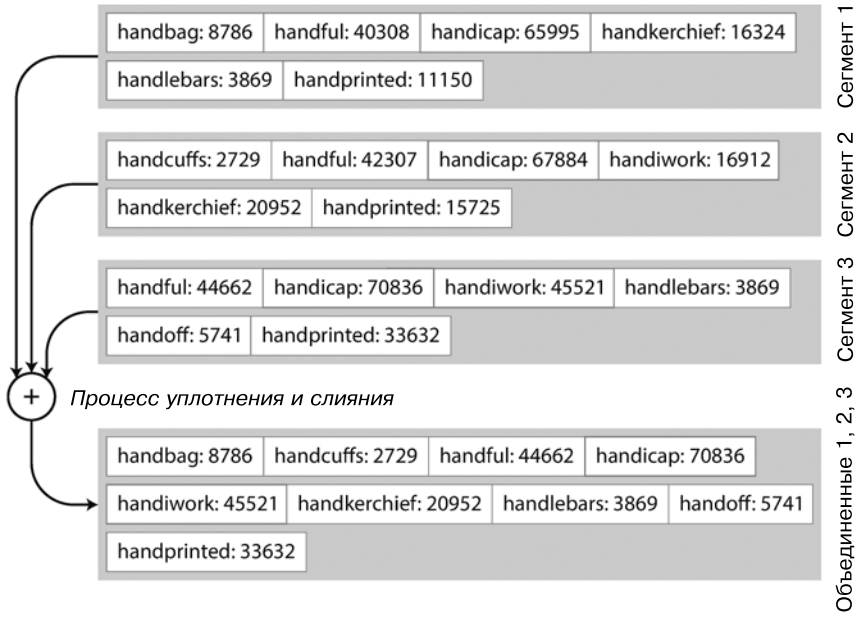


Рис. 3.4. Слияние нескольких сегментов SS-таблицы с сохранением лишь самого последнего значения по каждому ключу

известно смещение для ключей `handbag` и `handsome` и (благодаря сортировке) то, что `handiwork` должен находиться между ними. Как следствие, можно перейти по смещению для `handbag` и просматривать, начиная с этого места, до тех пор, пока не найдем `handiwork` (впрочем, можем и не найти, если ключ отсутствует в данном файле).

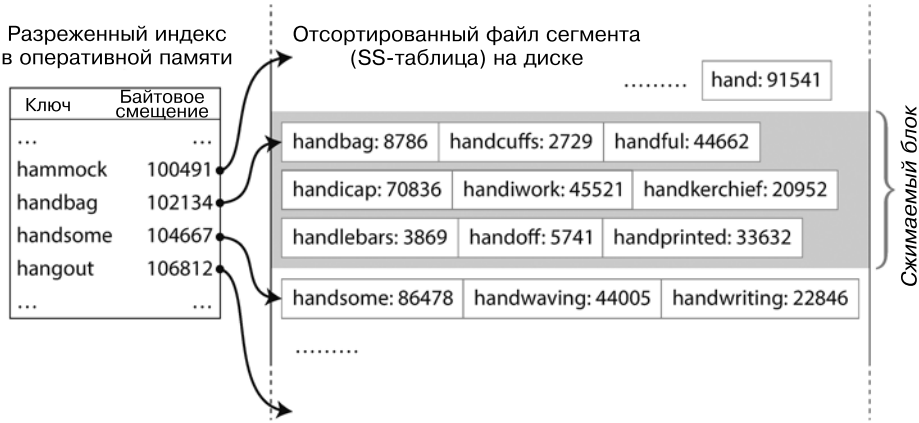


Рис. 3.5. SS-таблица с индексом в оперативной памяти

Нам все еще нужен индекс в оперативной памяти, чтобы узнавать смещение для некоторых ключей, но он может быть разреженным: одного ключа для каждых нескольких килобайт файла сегмента вполне достаточно, поскольку несколько килобайт можно просмотреть очень быстро[1].

3. Поскольку для выполнения запроса на чтение все равно необходимо просмотреть несколько пар «ключ — значение» из заданного диапазона, вполне можно сгруппировать эти записи в блок и сжать его перед записью на диск (показано затененной областью на рис. 3.5). Каждая запись разреженного индекса в оперативной памяти затем будет указывать на начало сжатого блока. Помимо экономии пространства на диске, сжатие также снижает использование полосы пропускания ввода/вывода.

Создание и поддержание SS-таблиц

До сих пор все было отлично, но как нам прежде всего отсортировать данные по ключу? Входящие операции записи могут производиться в любом порядке.

Поддержание отсортированной структуры на диске возможно (см. подраздел «B-деревья» текущего раздела), но гораздо проще будет делать это в оперативной памяти. Существует множество доступных для использования хорошо известных древовидных структур данных, например красно-черные деревья или AVL-деревья [2]. При использовании этих структур можно вставлять ключи в любой последовательности и затем читать их в нужном порядке.

Теперь мы организуем работу нашей подсистемы хранения следующим образом.

- При поступлении записи добавляем ее в располагающуюся в оперативной памяти сбалансированную структуру данных (например, красно-черное дерево). Это располагающееся в оперативной памяти дерево называется *MemTable* (от memory table — «таблица, расположенная в памяти»).
- Когда размер MemTable превышает определенное пороговое значение — обычно несколько мегабайт, — записываем его на диск в виде файла SS-таблицы. Эта операция выполняется достаточно эффективно, поскольку дерево поддерживает пары «ключ — значение» в отсортированном по ключу виде. Новый файл SS-таблицы становится последним сегментом базы данных. А пока SS-таблица записывается на диск, операции записи продолжают выполняться в новый экземпляр MemTable.
- Для обслуживания запроса на чтение сначала пробуем найти ключ в MemTable, затем в последнем по времени сегменте на диске, затем в предпоследнем и т. д.

---

[1] Если размер всех ключей и значений фиксированный, то можно воспользоваться двоичным поиском по файлу сегмента и избежать применения индекса в оперативной памяти полностью. Однако обычно их размеры меняются, что усложняет определение без индекса места, где заканчивается одна запись и начинается другая.

- Время от времени запускаем в фоне процесс слияния и уплотнения, чтобы объединить файлы сегментов и отбросить перезаписанные или удаленные значения.

Представленная схема работает отлично. У нее есть только одна проблема: если происходит фатальный сбой БД, то записанные позже всего данные (находящиеся в MemTable, но еще не записанные на диск) теряются. Чтобы избежать этой проблемы, можно держать на диске отдельный журнал, в конец которого немедленно добавляются все записываемые данные, точно так же, как в предыдущем разделе. Сам журнал неупорядочен, но это неважно, ведь его единственное назначение — восстановление MemTable после сбоя. Всякий раз, когда MemTable записывается в SS-таблицу, соответствующий журнал можно удалять.

Создание LSM-дерева из SS-таблиц

Описываемый тут алгоритм, по сути, используется в LevelDB [6] и RocksDB [7] — библиотеках подсистем хранения на основе пар «ключ — значение», предназначенных для встраивания в другие приложения. Помимо прочего, LevelDB можно применять в распределенной СУБД Riak в качестве альтернативы Bitcask. Аналогичные подсистемы хранения задействуются в Cassandra и HBase [8], вдохновленных статьей о Bigtable разработчиков из компании Google [9] (в этой статье впервые появились термины *SSTable* и *MemTable*).

Изначально эта индексная структура была описана Патриком О'Нилом и др. под названием *журналированного дерева слияния* (Log-Structured Merge-Tree, LSM-Tree) [10], на основе более ранней работы, посвященной журналированным файловым системам [11]. Подсистемы хранения, основанные на принципе слияния и уплотнения отсортированных файлов, часто называются LSM-подсистемами хранения.

Lucene, модуль индексации для полнотекстового поиска, применяемый серверами Elasticsearch и Solr, использует схожий метод хранения своего *словаря термов* (term dictionary) [12, 13]. Полнотекстовый индекс намного сложнее, чем индекс типа «ключ — значение», но основан на схожей идее: по заданному в поисковом запросе слову необходимо найти все документы (веб-страницы, описания товаров и т. п.), в которых упоминается искомое слово. Механизм реализуется с помощью структуры ключ/значение, где ключ — слово *(терм)*, а значение — список идентификаторов всех документов, содержащих это слово (*список словопозиций* (postings list)). В библиотеке Lucene данная карта соответствий термов списку словопозиций хранится в подобных SS-таблицам отсортированных файлах, слияние которых при необходимости выполняется в фоновом режиме [14].

Оптимизация производительности

Как и всегда, на практике для обеспечения хорошей производительности подсистемы хранения требуется учесть немало нюансов. Например, алгоритм на основе LSM-дерева может работать медленно при поиске отсутствующих в базе данных

ключей: приходится просматривать MemTable, затем все сегменты вплоть до самого старого (вероятно, читая каждый из них с диска), прежде чем удастся убедиться, что ключ отсутствует. Чтобы оптимизировать подобную разновидность доступа, подсистемы хранения часто применяют дополнительные *фильтры Блума* [15]. (Фильтр Блума — это эффективно использующая память структура данных для приближенного определения содержимого множества. Она позволяет установить, встречается ли ключ в базе данных, экономя таким образом множество лишних чтений диска для несуществующих ключей.)

Существуют также различные стратегии определения очередности и хронометража уплотнения и слияния SS-таблиц. Чаще всего используемые варианты — *уплотнение по слоям в соответствии с размером* (size-tiered compaction) и *поуровневое уплотнение* (leveled compaction). СУБД LevelDB и RocksDB задействуют поуровневое уплотнение (отсюда и название LevelDB), в СУБД HBase применяется уплотнение по слоям в соответствии с размером, а Cassandra поддерживает оба варианта [16]. При уплотнении по слоям в соответствии с размером меньшие и более новые SS-таблицы постепенно становятся частью более старых и бóльших SS-таблиц. При поуровневом уплотнении диапазон ключей разбивается на меньшие SS-таблицы и более старые данные перемещаются на отдельные уровни, благодаря чему уплотнение осуществляется пошагово и использует меньше дискового пространства.

Несмотря на множество нюансов, основная идея LSM-деревьев — применение каскада SS-таблиц, объединяемых в фоновом режиме, — проста и эффективна. Она хорошо работает даже в случае, когда размер набора данных значительно превышает доступный объем оперативной памяти. То, что данные хранятся в отсортированном виде, дает возможность эффективно выполнять запросы по диапазонам (просмотр всех ключей между установленными минимальным и максимальным значениями), а поскольку записи на диск осуществляются последовательно, LSM-дерево способно поддерживать весьма высокую пропускную способность по записи.

B-деревья

Журналированные индексы, которые мы обсудили выше, понемногу набирают популярность, но это отнюдь не самый распространенный тип индексов. Наиболее широко используемая индексная структура — B-дерево.

Появившиеся в 1970-м [17] и спустя менее десяти лет описываемые как «повсеместно используемые» [18], B-деревья очень хорошо выдержали испытание временем. Они остаются стандартной реализацией индексов практически во всех реляционных базах данных, да и многие нереляционные тоже их применяют.

Аналогично SS-таблицам B-деревья хранят пары «ключ — значение» в отсортированном по ключу виде, что позволяет эффективно выполнять поиск значения по

ключу и запросы по диапазонам. Но на этом сходство заканчивается: конструктивные принципы B-деревьев совершенно другие.

Журналированные индексы, которые мы видели ранее, разбивают базу данных на *сегменты* переменного размера, обычно несколько мегабайт или более, и всегда записывают их на диск последовательно. В отличие от них, B-деревья разбивают БД на *блоки* или *страницы* фиксированного размера, обычно 4 Кбайт (иногда больше), и читают/записывают по одной странице за раз. Такая конструкция лучше подходит для нижележащего аппаратного обеспечения, поскольку диски тоже разбиваются на блоки фиксированного размера.

Все страницы имеют свой адрес/местоположение, благодаря чему одни страницы могут ссылаться на другие — аналогично указателям, но на диске, а не в памяти. Этими ссылками на страницы можно воспользоваться для создания дерева страниц, как показано на рис. 3.6.

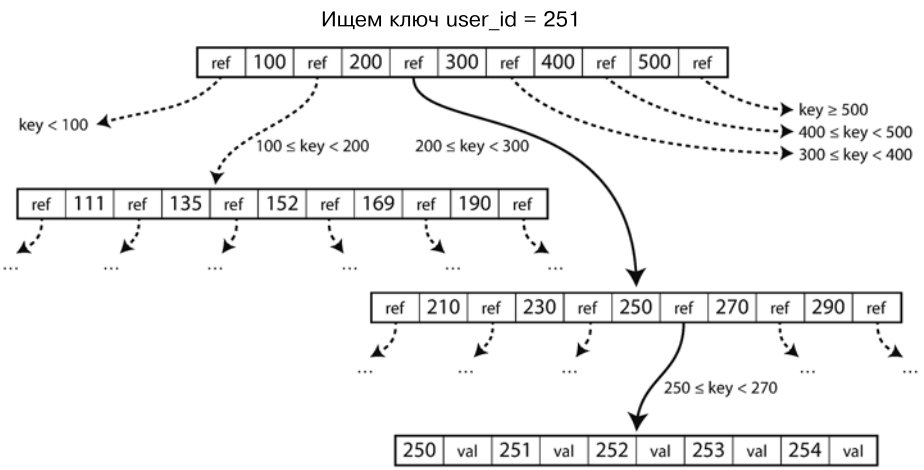


Рис. 3.6. Поиск ключа с помощью индекса на основе B-деревьев

Одна из страниц назначается *корнем* B-дерева, с него начинается любой поиск ключа в индексе. Данная страница содержит несколько ключей и ссылок на дочерние страницы. Каждая из них отвечает за непрерывный диапазон ключей, а ключи, располагающиеся между ссылками, указывают на расположение границ этих диапазонов.

В примере на рис. 3.6 мы ищем ключ 251, вследствие чего знаем, что нужно перейти по ссылке на страницу между границами 200 и 300. Это приводит нас на схожую страницу, разбивающую диапазон 200–300 на поддиапазоны. Постепенно мы добираемся до страницы, содержащей отдельные ключи *(страница-лист)*, которая содержит или значения для всех ключей, или ссылки на страницы, где можно найти эти значения.

Количество ссылок на дочерние страницы на одной странице B-дерева называется *коэффициентом ветвления* (branching factor). Например, на рис. 3.6 коэффициент ветвления равен 6. На практике этот коэффициент зависит от дискового пространства, которое требуется для хранения ссылок на страницы и границ диапазонов, но обычно равен нескольким сотням.

При необходимости обновить значение для существующего ключа в B-дереве нужно найти содержащую этот ключ страницу-лист, изменить там значение и записать ее обратно на диск (все ссылки на данную страницу останутся рабочими). В случае надобности добавить новый ключ следует найти страницу, в чей диапазон попадает новый ключ, и добавить его туда. Если на странице недостаточно места для него, то она разбивается на две полупустые страницы, а родительская страница обновляется, чтобы учесть это разбиение диапазона ключей на части (рис. 3.7)[1].

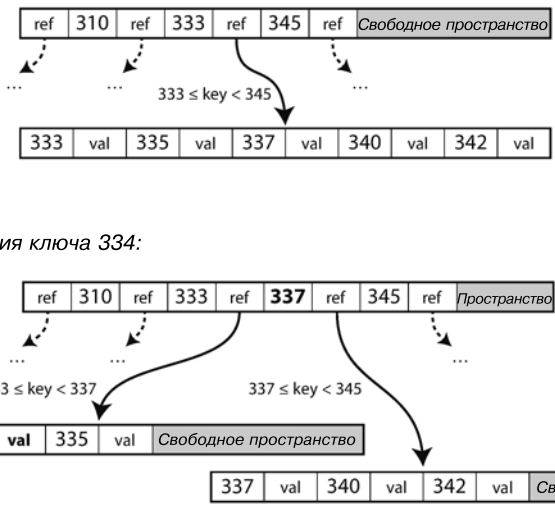


Рис. 3.7. Наращивание B-дерева с помощью разбиения страницы

Представленный алгоритм гарантирует, что дерево останется *сбалансированным*, то есть глубина B-дерева с n ключами будет равна $O(\log n)$. Большинству баз данных хватает деревьев глубиной три или четыре уровня, поэтому вам не придется проходить по множеству ссылок на страницы с целью найти нужную (четырехуровневое дерево страниц по 4 Кбайт с коэффициентом ветвления в 500 может хранить до 256 Тбайт информации).

[1] Процедура вставки нового ключа в B-дерево интуитивно понятна, но удаление его (с сохранением сбалансированности дерева) — задача не такая простая [2].

Обеспечение надежности B-деревьев

Базовая операция записи B-дерева — перезапись страницы на диске новыми данными. Предполагается, что эта перезапись не меняет расположения страницы, то есть все ссылки на нее остаются неизменными. Это резко отличается от журнализированных индексов, например, LSM-деревьев, в которых происходит только дописывание (и постепенное удаление устаревших файлов), но не изменение существующих файлов.

Перезаписывание страницы на диске можно рассматривать как реальную операцию, выполняемую средствами аппаратного обеспечения. На магнитном жестком диске это означает перемещение головки диска в нужное место, ожидание поворота вращающейся пластины в нужное положение и перезапись соответствующего сектора новыми данными. На твердотельных накопителях (SSD) все несколько сложнее, поскольку такой накопитель должен стирать и перезаписывать сразу довольно большие блоки на микросхеме памяти [19].

Более того, для отдельных операций необходимо перезаписать несколько различных страниц. Например, при разбиении страницы из-за ее переполнения вследствие вставки необходимо записать две новые страницы, а также перезаписать их родительскую страницу для обновления ссылок на две дочерние страницы. Это опасная операция, ведь в случае фатального сбоя базы данных в тот момент, когда записана только часть страниц, индекс окажется поврежден (например, может возникнуть так называемая *бесхозная* (orphan) страница, у которой нет ни одного родителя).

Чтобы сделать БД отказоустойчивой, реализации B-деревьев обычно включают дополнительную структуру данных на диске: *журнал упреждающей записи* (write-ahead log, WAL), также именуемый *журналом повтора* (redo log). Он представляет собой файл, предназначенный только для добавления, в который все модификации B-деревьев должны записываться еще до того, как применяться к самим страницам дерева. Когда база возвращается в норму после сбоя, этот журнал используется для восстановления B-дерева в согласованное состояние [5, 20].

Дополнительная сложность обновления страниц на месте заключается в том, что при одновременном доступе нескольких потоков выполнения к B-дереву необходима аккуратность в управлении конкурентным доступом, в противном случае поток может просмотреть дерево в несогласованном состоянии. Механизм обычно реализуется путем защиты структур данных дерева с помощью *защелок* (latch) — облегченного варианта блокировок. Журналированный подход в этом смысле проще, поскольку все слияние происходит в фоновом режиме, без создания помех входящим запросам, с атомарной заменой старых сегментов новыми время от времени.

Усовершенствования B-деревьев

Поскольку B-деревья применяются уже достаточно долго, неудивительно, что за эти годы они претерпели немало усовершенствований. Упомяну лишь несколько.

- Вместо перезаписи страниц и поддержания WAL в целях восстановления после сбоев некоторые базы данных (например, LMDB) используют схему копирования при записи [21]. Измененная страница записывается в другое место с созданием новых версий родительских страниц в дереве, указывающих на это новое местоположение. Такой подход полезен и для управления конкурентным доступом, как мы увидим в подразделе «Изоляция снимков состояния и воспроизводимое чтение» раздела 7.2.
- Можно сэкономить место на страницах, сохраняя не весь ключ, а только его сокращенный вариант. Ключи, особенно на страницах во внутренней части дерева, нужны только в качестве информации о границах между диапазонами ключей. Размещение на странице большего количества ключей позволяет повысить коэффициент ветвления дерева, а следовательно, уменьшить количество его уровней[1].
- В целом страницы могут находиться в любом месте диска, нет никаких требований о том, чтобы страницы с соседними диапазонами ключей располагались на диске рядом. Если для запроса необходимо просмотреть значительную часть диапазона ключей в отсортированном порядке, то подобная постраничная схема может оказаться непродуктивной, поскольку для каждой читаемой страницы понадобится перейти к нужной позиции на диске. Поэтому множество реализаций B-деревьев пытаются расположить дерево так, чтобы его листья находились на диске в последовательном порядке. Однако сохранять данный порядок при росте дерева непросто. А для LSM-деревьев, переписывающих за один раз большие сегменты хранилища во время слияния, напротив, легче хранить последовательные ключи на диске рядом друг с другом.
- В дерево добавляются дополнительные указатели. Например, каждая страница-лист может ссылаться на страницы того же уровня слева и справа, что позволяет упорядоченно просматривать ключи без возврата к родительским страницам.
- Такие варианты B-деревьев, как *фрактальные деревья* [22], заимствуют некоторые идеи у журналированных индексов с целью снизить количество необходимых переходов к позициям на диске (и у них нет ничего общего с фракталами).

---

[1] Этот вариант иногда называют B+-деревом, хотя такое усовершенствование настолько распространено, что его часто не отличают от других вариантов B-деревьев.

Сравнение B- и LSM-деревьев

Хотя реализации B-деревьев в целом более совершенны, чем реализации LSM-деревьев, последние тоже представляют некоторый интерес, благодаря своей производительности. Как правило, LSM-деревья обычно быстрее при записи, а B-деревья — при чтении [23]. Чтение выполняется медленнее на LSM-деревьях потому, что приходится просматривать несколько различных структур данных и SS-таблиц, находящихся на разных стадиях уплотнения.

Однако эти оценки часто неубедительны и сильно зависят от нюансов конкретной рабочей нагрузки. Необходимо тестировать системы именно под вашей конкретной нагрузкой, чтобы сравнение было достоверным. Ниже мы коротко обсудим несколько нюансов, которые имеет смысл учесть при оценке производительности подсистемы хранения.

Достоинства LSM-деревьев

Индекс на основе B-дерева должен записывать каждый элемент данных по крайней мере дважды: один раз в журнал с упреждающей записью и второй — на саму страницу дерева (и вероятно, снова при разбиении страниц). Кроме того, возникают дополнительные накладные расходы из-за того, что записывать приходится сразу всю страницу, даже если в ней поменялось лишь несколько байтов. А некоторые подсистемы хранения перезаписывают одну и ту же страницу дважды, чтобы не остаться с не полностью обновленной страницей в случае сбоя питания [24, 25].

Журналированные индексы тоже переписывают данные несколько раз из-за многократного уплотнения и слияния SS-таблиц. Эффект, при котором одна операция записи в БД приводит ко множеству операций записи на диск за все время жизни базы, известен под названием *усиления записи* (write amplification). Этот вопрос особенно важен в случае SSD, способных переписывать блоки лишь ограниченное количество раз до полного износа.

В требующих больших объемов записи приложениях узким местом по производительности способна оказаться скорость, с которой база записывает данные на диск. В этом случае усиление записи напрямую влияет на производительность: чем больше подсистема хранения записывает на диск, тем меньше операций записи в секунду она может выполнить в рамках доступной ей полосы пропускания диска.

Более того, LSM-деревья обычно способны обеспечить бо́льшую пропускную способность, чем B-деревья, частично из-за их более слабого усиления записи (хотя это зависит от настроек подсистемы хранения и нагрузки), а частично — в силу того, что они последовательно записывают компактные файлы SS-таблиц вместо перезаписи нескольких страниц дерева [26]. Описанная разница особенно существенна

для магнитных жестких дисков, на которых последовательные операции записи работают намного быстрее, чем произвольные.

LSM-деревья также лучше сжимаются, а следовательно, приводят к меньшему размеру файлов на диске, чем B-деревья. Подсистемы хранения на основе B-деревьев оставляют некоторое пространство на диске незадействованным в силу фрагментации: когда страница разбивается на части или строка не помещается на существующую страницу, некий объем пространства на странице остается незадействованным. Поскольку LSM-деревья не ориентированы на работу со страницами и периодически переписывают SS-таблицы для исключения фрагментации, избыточность у них ниже, особенно при использовании поуровневого уплотнения [27].

На многих твердотельных накопителях встроенное ПО применяет для внутренних целей журналированный алгоритм, чтобы превратить произвольные операции записи в последовательные записи на микросхемах памяти, поэтому влияние паттерна записи подсистемы хранения выражено не так сильно [19]. Однако более слабое усиление записи и пониженная фрагментация все равно приносят в случае SSD определенную пользу: более сжатое представление данных позволяет выполнить больше запросов на чтение и запись в рамках доступной пропускной способности ввода/вывода.

Недостатки LSM-деревьев

Один из недостатков журналированных подсистем хранения таков: процесс уплотнения иногда способен отрицательно влиять на производительность выполняемых в этот момент операций чтения и записи. Несмотря даже на то, что подсистемы хранения стараются выполнять уплотнение пошагово, не влияя на конкурентный доступ к данным, ресурсы дисков ограниченны. Здесь может легко возникнуть такая ситуация, при которой запросу придется ожидать завершения диском ресурсоемкой операции уплотнения. Влияние этого на пропускную способность и среднее время отклика обычно невелико, но на более высоких процентилях (см. подраздел «Описание производительности» раздела 1.3) время отклика запросов к журналированным подсистемам хранения может иногда быть весьма значительным, а B-деревья — оказаться более предсказуемыми [28].

Другая проблема уплотнения возникает при высоких объемах записи: ограниченную полосу пропускания диска по записи приходится делить между потоком, осуществляющим основные операции записи (журналирование и сброс MemTable на диск), и потоками, осуществляющими уплотнение, которые работают в фоновом режиме. При записи в пустую базу данных всю полосу пропускания диска по записи можно использовать для основной записи, но чем больше становится БД, тем большая часть полосы пропускания диска требуется для уплотнения.

Если объемы записи велики, а уплотнение не настроено должным образом, то может случиться так, что уплотнение не будет успевать за темпами входящих опе-

раций записи. В этом случае количество необъединенных сегментов на диске будет расти до тех пор, пока место на диске не закончится, а операции чтения тоже замедлятся, поскольку нужно будет просматривать большее количество файлов сегментов. Обычно подсистемы хранения на основе SS-таблиц не ограничивают темпы входящих операций записи, даже если уплотнение за ними не успевает, поэтому для обнаружения подобной ситуации понадобится отдельный мониторинг [29, 30].

Преимущество B-деревьев состоит в том, что каждый ключ встречается в индексе только в одном месте, в то время как в журналированных подсистемах хранения может быть несколько копий одного ключа в разных сегментах. Эта особенность функционирования делает B-деревья привлекательными для тех баз данных, которые стремятся обеспечить сильно выраженную транзакционность: во многих реляционных БД изоляция транзакций реализуется с помощью блокировок для диапазонов ключей, а в индексах на основе B-деревьев такие блокировки можно непосредственно привязать к дереву [5]. В главе 7 мы обсудим этот вопрос подробнее.

B-деревья глубоко укоренились в архитектуре БД и обеспечивают неизменно хорошую производительность для многих видов нагрузки, поэтому маловероятно, что в ближайшее время от них откажутся. А в новых хранилищах данных становятся все более популярны журналированные индексы. Не существует быстрого и простого правила, которое бы позволило решить, какой тип подсистемы хранения лучше в конкретном случае, это приходится проверять опытным путем.

Другие индексные структуры

До сих пор мы обсуждали только индексы типа «ключ — значение», напоминающие индекс *по первичному ключу* в реляционной модели. Первичный ключ однозначно идентифицирует одну строку в реляционной таблице, или один документ в документоориентированной базе данных, или одну вершину в графовой. Другие записи в БД могут ссылаться на эту строку/документ/вершину по ее первичному ключу (или идентификатору), а индекс используется для разрешения подобных ссылок.

Кроме того, распространенной практикой является применение *вторичных индексов* (secondary index). В реляционных базах данных в одной таблице можно создавать несколько вторичных индексов с помощью команды `CREATE INDEX`, причем они часто критически важны для эффективного выполнения соединений. Так, в примере, показанном на рис. 2.1 в главе 2, вы, вероятнее всего, создали бы вторичные индексы по столбцам `user_id` для поиска всех относящихся к одному пользователю столбцов во всех таблицах.

Вторичный индекс легко создается из индекса типа «ключ — значение». Основное отличие в том, что ключи неуникальны, то есть может быть несколько строк

(документов, вершин) с одним и тем же ключом. Эта проблема решается двумя способами: превратив каждое значение в индексе в список соответствующих идентификаторов строк (аналогично списку словопозиций в полнотекстовом индексе) или сделав все ключи уникальными, добавив к ним идентификатор строки. В любом случае как B-дерево, так и журналированные индексы можно использовать в качестве вторичных индексов.

Хранение значений в индексах

Запросы ищут в индексе ключ, а значение между тем может быть фактической искомой строкой (документом, вершиной) или ссылкой на строку, хранящуюся где-то в другом месте. Во втором случае место, где хранятся строки, называется *неупорядоченным файлом* (heap file), и данные там хранятся, соответственно, в неупорядоченном виде. (Это может быть файл, предназначенный только для добавления данных, или же в нем могут отслеживаться удаленные строки, чтобы позднее перезаписать их новыми данными.) Неупорядоченные файлы используются весьма часто, ведь они позволяют избежать дублирования данных в случае нескольких вторичных индексов: все индексы лишь ссылаются на местоположение в неупорядоченном файле, а фактические данные хранятся в одном месте.

При обновлении значения без изменения ключа подход с использованием неупорядоченного файла может оказаться достаточно продуктивным: запись перезаписывается на месте, конечно, при условии, что новое значение меньше старого. Если же оно больше по размеру, ситуация усложняется, так как, вероятно, придется его переместить в другое место, где достаточно свободного пространства. В этом случае придется или обновить все индексы так, чтобы они указывали на новую позицию в неупорядоченном файле, или разместить по старому адресу указатель для переадресации [5].

В некоторых случаях лишний переход от индекса к неупорядоченному файлу — слишком затратная вещь для чтения в смысле производительности, поэтому желательно хранить проиндексированную строку непосредственно в индексе. Такой вариант носит название *кластеризованного индекса* (clustered index). Например, в подсистеме хранения InnoDB СУБД MySQL первичный ключ таблицы всегда представляет собой кластеризованный индекс, а вторичные индексы ссылаются на первичный ключ (а не на позицию в неупорядоченном файле) [31]. В СУБД SQL Server можно задавать один кластеризованный индекс на таблицу [32].

Компромисс между кластеризованным индексом (хранением всех ключей в индексе) и некластеризованным (хранением в индексе только ссылок на данные) известен под названием *охватывающего индекса* (covering index) или *индекса с включенными столбцами* (index with included columns). При этом в индексе хранится только *часть* столбцов таблицы [33]. Благодаря такому обстоятельству появляется возможность отвечать на некоторые запросы с помощью только одного индекса (и тогда говорят, что индекс *охватывает* запрос) [32].

Как и при любом виде дублирования данных, кластеризованный и охватывающий индексы могут ускорить чтение, но требуют дополнительного пространства на диске и способны привести к увеличению накладных расходов при записи. Базам данных также приходится прикладывать дополнительные усилия для обеспечения транзакционности, поскольку приложения не должны сталкиваться с несогласованностью из-за дублирования.

Составные индексы

Обсуждавшиеся до сих пор индексы задавали соответствие отдельного ключа и значения. Этого недостаточно в случае запроса нескольких столбцов таблицы (или нескольких полей документа) одновременно.

Наиболее распространенный тип составных индексов — *сцепленный индекс* (concatenated index), который просто объединяет несколько полей в один ключ, присоединяя один столбец к другому (в описании индекса указывается, в каком порядке сцепляются поля). Он чем-то похож на старомодную бумажную телефонную книгу с индексом. Благодаря порядку сортировки индекс можно использовать для поиска всех людей с конкретной фамилией или всех людей с конкретным сочетанием *«фамилия-имя»*. Однако он бесполезен, если нужно найти всех людей с конкретным именем.

Многомерные индексы — более общий способ запроса нескольких столбцов сразу, особенно важный при работе с геопространственными данными. Например, у сайта поиска ресторанов может быть БД с координатами широты и долготы всех ресторанов. При просмотре пользователем ресторанов на карте сайту необходимо найти все рестораны, расположенные внутри текущей прямоугольной области на карте. Для этого необходим двумерный запрос по диапазону вот такого вида:

```
SELECT * FROM restaurants WHERE latitude  > 51.4946 AND latitude  < 51.5079
                          AND longitude > -0.1162 AND longitude < -0.1004;
```

Стандартный индекс на основе B-дерева или LSM-дерева не сможет обеспечить эффективное выполнение подобного запроса: он сумеет выдать или все рестораны в определенном диапазоне широт (но с произвольной долготой), или все рестораны в определенном диапазоне долгот (но в любой точке между Северным и Южным полюсами), но не сможет выдать и то и другое сразу.

Один из вариантов решения этого проблемы — преобразовать двумерное местоположение в одно число с помощью заполняющей пространство кривой, после чего воспользоваться обычным индексом на основе B-дерева [34]. Но чаще применяются специализированные пространственные индексы, такие как R-деревья. Например, в PostGIS геопространственные индексы реализуются в виде R-деревьев благодаря функции Generalized Search Tree (обобщенное дерево поиска) СУБД PostgreSQL [35]. У нас недостаточно места, чтобы описывать в этой книге R-деревья во всех подробностях, но им посвящено множество литературы.

Интересная идея: многомерные индексы можно использовать не только для географических координат. Например, на сайте интернет-магазина задействовать трехмерный индекс по осям координат *(красный, зеленый, синий)*, чтобы искать товары в определенном диапазоне цветов. А в базе данных метеорологических наблюдений можно применить двумерный индекс по координатам *(дата, температура)* для эффективного поиска всех наблюдений за 2013 год, в которых температура была между 25 и 30 градусами Цельсия. С одномерным индексом пришлось бы или просматривать все записи за 2013 год (независимо от температуры), после чего фильтровать их по температуре, или наоборот. Двумерный индекс способен сузить поле поиска сразу и по метке даты/времени, и по температуре. Этот метод используется в распределенном хранилище HyperDex [36].

Полнотекстовый поиск и нечеткие индексы

Все обсуждавшиеся до сих пор индексы принимали за аксиому то, что данные точны, и предоставляли возможность поиска по конкретным значениям ключей или диапазонам значений ключей с определенной сортировкой. Но они не давали искать *похожие* ключи, например слова с орфографическими ошибками. Подобные *нечеткие* (fuzzy) запросы требуют применения других методов.

Например, системы полнотекстового поиска обычно позволяют расширять поиск по слову путем включения синонимов этого слова, игнорирования грамматических вариантов слов, поиска вхождения слов рядом друг с другом в одном документе и поддерживают другие возможности, основанные на лингвистическом анализе текста. Чтобы справиться с опечатками в документах или запросах, Lucene способен искать в тексте слова в пределах определенного редакторского расстояния (редакторское расстояние, равное 1, означает добавление, удаление или замену одной буквы) [37].

Как уже упоминалось в пункте «Создание LSM-дерева из SS-таблиц» подраздела «SS-таблицы и LSM-деревья» текущего раздела, в Lucene используется подобная SS-таблице структура в качестве словаря термов. Для этой структуры необходим небольшой индекс в оперативной памяти, который бы предоставлял запросам информацию о смещении в отсортированном файле нужного им ключа. В LevelDB данный индекс в оперативной памяти представляет собой разреженную коллекцию части ключей, а в Lucene — распознающий конечный автомат на основе символов из ключей, аналогичный префиксному дереву (trie) [38]. Такой автомат можно преобразовать в *автомат Левенштейна*, поддерживающий эффективный поиск слов в пределах заданного редакторского расстояния [39].

Другие методы нечеткого поиска связаны с классификацией документов и машинным обучением. Дальнейшую информацию см. в учебниках по информационному поиску (например, 40).

Храним все в памяти

Все обсуждавшиеся ранее в этой главе структуры данных были нацелены на решение проблем, связанных с ограничениями дисковой памяти. По сравнению с оперативной памятью диски гораздо менее удобны. В случае как твердотельных накопителей, так и магнитных дисков данные на диске необходимо размещать очень обдуманно, чтобы достичь хорошей производительности операций чтения и записи. Однако мы терпим эти неудобства, ведь у дисков есть два важных достоинства: они обеспечивают сохраняемость данных (их содержимое не теряется при отключении питания) и стоимость их в пересчете на гигабайт памяти ниже, чем у RAM.

По мере удешевления RAM аргумент относительно стоимости в пересчете на гигабайт памяти постепенно слабеет. Многие наборы данных попросту не настолько велики, так что вполне допустимо держать их полностью в оперативной памяти, возможно распределенной по нескольким машинам. Этот факт привел к разработке *размещаемых в оперативной памяти БД* (in-memory databases).

Некоторые размещаемые в оперативной памяти хранилища типа «ключ — значение», такие как Memcached, используют кэш, только когда допустимы потери данных в случае перезагрузки машины. Но другие БД, размещаемые в оперативной памяти, стремятся обеспечить сохраняемость, что достигается с помощью специального аппаратного обеспечения (например, RAM, питающейся от аккумуляторов), записи на диск журналов изменений, периодической записи на диск копий состояния или путем репликации состояния оперативной памяти на другие машины.

При перезапуске размещаемой в оперативной памяти базе данных необходимо заново загрузить свое состояние с диска или из реплики по сети (если не используется специальное аппаратное обеспечение). Несмотря на запись данных на диск, она все равно остается БД, размещаемой в оперативной памяти. Так происходит потому, что диск применяется только в качестве дописываемого журнала с целью обеспечить сохраняемость, а операции чтения выполняются только из оперативной памяти. У записи на диск есть и эксплуатационные преимущества: создавать резервные копии файлов на диске легче, как и проверять их и анализировать с помощью внешних утилит.

Такие программные продукты, как VoltDB, MemSQL и Oracle TimesTen, представляют собой размещаемые в оперативной памяти БД на основе реляционной модели, и их производители утверждают, что им удалось значительно повысить производительность за счет исключения всех накладных расходов, связанных со структурами данных на диске [41, 42]. RAMCloud — размещаемое в оперативной памяти хранилище типа «ключ — значение» с открытым исходным кодом, обеспечивающее сохраняемость (благодаря использованию журналированного подхода

для данных в памяти и на диске) [43]. Redis и Couchbase обеспечивают слабую сохраняемость с помощью асинхронной записи на диск.

Парадоксально, но тот факт, что им не нужно читать с диска, не дает преимущества в производительности размещаемым в оперативной памяти БД. Даже дисковым подсистемам хранения может никогда не понадобиться читать с диска, если у вас достаточно оперативной памяти, поскольку операционная система все равно кэширует недавно прочитанные блоки диска в памяти. Скорее они работают быстрее потому, что способны избежать накладных расходов по кодированию располагаемых в памяти структур данных в форму, в которой они могли бы быть записаны на диск [44].

Помимо производительности, еще одно интересное преимущество располагаемых в памяти БД состоит в предоставлении моделей данных, которые сложно реализовать с помощью дисковых индексов. Например, Redis обеспечивает напоминающий БД интерфейс для различных структур данных, таких как очереди по приоритету и множества. В силу хранения всей информации в оперативной памяти его реализация довольно проста.

Недавние исследования показывают: архитектура располагаемых в памяти БД может быть расширена, чтобы поддержать наборы данных, превышающие размер доступной памяти, без возвращения к накладным расходам дискоцентрической архитектуры [45]. Так называемый подход *антикэширования* работает за счет перемещения наиболее давно использовавшихся данных из памяти на диск при отсутствии достаточного количества доступной памяти и загрузки их обратно в оперативную память при обращении в будущем. Это напоминает то, что операционные системы делают с виртуальной памятью и файлами подкачки, но БД может управлять памятью эффективнее, чем операционная система, за счет манипуляций на уровне отдельных записей, а не целых страниц памяти. Однако при таком подходе все равно необходимо, чтобы индексы полностью помещались в оперативной памяти (подобно примеру с Bitcask в начале данной главы).

Вероятно, когда станут шире использоваться технологии *энергонезависимой памяти* (non-volatile memory, NVM), понадобятся дальнейшие изменения в конструкции подсистем хранения [46]. В настоящее время это еще относительно новая область исследований, но она заслуживает пристального внимания.

3.2. Обработка транзакций или аналитика?

В самом начале эры обработки коммерческих данных операция записи в БД обычно соответствовала произведенной *коммерческой операции* (commercial transaction): продаже чего-либо, размещению заказа у поставщика, выплате зарплаты служащему и т. п. Хотя использование БД расширилось до сфер, в которых не происходит перехода денег из рук в руки, термин *«транзакция»* остался применительно к группе операций чтения и записи, составляющей логически единое целое.



Транзакция не обязательно должна обладать свойствами ACID (Atomicity, Consistency, Isolation, Durability — атомарность, согласованность, изоляция и сохраняемость). Обработка транзакций означает просто возможность для клиентов выполнять операции чтения и записи с низким значением задержки — в противоположность заданиям пакетной обработки, запускаемым лишь периодически (например, один раз в день). Мы обсудим свойства ACID в главе 7, а пакетную обработку — в главе 10.

И хотя БД уже применяются для множества разных видов данных — комментариев в сообщениях блогов, действий в играх, контактов в адресной книге и т. д., — основные паттерны доступа остались сходными с обработкой коммерческих операций. Приложение обычно ищет с помощью индекса небольшое количество записей по какому-либо ключу. На основе вводимых пользователем данных вставляются или обновляются записи. В силу интерактивности этих приложений такой паттерн доступа получил название *«обработка транзакций в реальном времени»* (online transaction processing, OLTP).

Однако БД все шире используются для *аналитической обработки данных* (data analytics), паттерны доступа которой совершенно другие. Обычно аналитический запрос должен просматривать огромное количество записей, читая в каждой из них только несколько столбцов и вычисляя сводные статистические показатели (например, количество, сумму или среднее значение) вместо возврата пользователю необработанных данных. Например, если данные представляют собой таблицу операций продажи, то аналитические запросы могут выглядеть следующим образом.

- Какова общая выручка каждого из магазинов в январе?
- Насколько больше бананов, по сравнению с обычным значением, было продано во время недавней рекламной кампании?
- Какую марку детского питания чаще всего покупают вместе с маркой X подгузников?

Эти запросы чаще всего написаны бизнес-аналитиками и вставлены в отчеты, которые руководство компании использует для оптимизации коммерческих решений (*бизнес-аналитика*, business intelligence). Чтобы отличать этот паттерн применения БД от обработки транзакций, его назвали *аналитической обработкой данных в реальном времени* (online analytical processing, OLAP) [47][1]. Различия OLTP и OLAP не всегда очевидны, но некоторые типичные особенности перечислены в табл. 3.1.

[1] Смысл слова online в OLAP не вполне четко определен; вероятно, речь идет не только о том, что эти запросы используются для встроенных отчетов, но и что аналитики могут задействовать OLAP-системы интерактивно для исследовательских запросов.

Таблица 3.1. Сравнительные характеристики обработки транзакций и аналитических систем

Свойство	Системы обработки транзакций (OLTP)	Аналитические системы (OLAP)
Основной паттерн чтения	Небольшое количество записей на один запрос, извлекается по ключу	Агрегирование по большому количеству записей
Основной паттерн записи	Произвольный доступ, операции записи с низким значением задержки на основе вводимых пользователем данных	Групповой импорт (ETL) или поток событий
В основном применяется	Конечными пользователями/ заказчиками, через веб-приложение	Штатным аналитиком, для поддержки принятия решений
Какие данные отражает	Актуальное состояние данных (текущий момент времени)	Историю событий, происходивших на протяжении отрезка времени
Размер набора данных	От гигабайтов до терабайтов	От терабайтов до петабайтов

Сначала как для обработки транзакций, так и для аналитических запросов использовались одни и те же базы данных. Язык SQL оказался в этом смысле весьма гибок: он работает при OLTP-запросах ничуть не хуже, чем при OLAP-запросах. Тем не менее в конце 1980-х — начале 1990-х годов возникла такая тенденция: компании прекращали задействовать OLTP-системы для целей аналитики и выполняли анализ на отдельных БД, которые назывались *складами данных* (data warehouse).

Складирование данных

У предприятия могут быть десятки различных систем обработки транзакций: системы, обслуживающие сайт с интерфейсом для пользователей, системы управления POS-терминалами (подсчетом стоимости покупок) в реальных магазинах, системы учета запасов на складах, планирования маршрутов для автомобилей, системы управления поставками, персоналом и т. д. Все эти системы сложны и требуют команды разработчиков для поддержки, так что в итоге эксплуатируются практически автономно друг от друга.

Обычно от этих OLTP-систем ожидается высокая доступность и обработка транзакций с низкой задержкой, поскольку они зачастую критичны для работы бизнеса. Администраторы БД, соответственно, старательно оберегают свои базы данных OLTP. Они обычно крайне неохотно разрешают бизнес-аналитикам выполнять произвольные аналитические запросы на этих базах, ведь эти запросы зачастую оказываются ресурсоемкими, связаны с просмотром больших частей набора данных, что может отрицательно сказаться на производительности выполняемых в этот момент транзакций.

Склад данных (data warehouse), напротив, представляет собой отдельную БД, которую аналитики могут опрашивать так, как им заблагорассудится, не влияя при этом на OLTP-операции [48]. Склад содержит предназначенную только для чтения копию данных из всех различных OLTP-систем компании. Данные извлекаются из баз OLTP (с помощью выполнения периодических дампов данных или непрерывного потока обновлений данных), преобразуются в удобный для анализа вид, очищаются и затем загружаются в склад. Процесс их помещения в склад известен под названием *«извлечение — преобразование — загрузка»* (extract — transform — load, ETL) и показан на рис. 3.8.

Склады данных сейчас имеются практически на всех крупных предприятиях, но в маленьких компаниях их и в помине нет. Вероятно, это происходит потому, что в большинстве маленьких компаний не так много различных OLTP-систем и количество данных тоже мало — достаточно невелико для того, чтобы к ним можно было осуществлять запросы в обычной базе данных SQL или даже анализировать в приложениях электронных таблиц. В большой компании необходимо приложить множество усилий для выполнения задач, которые в маленькой столь просты.

Большим преимуществом использования отдельного склада данных, а не выполнения запросов непосредственно к OLTP-системам является то, что склады можно оптимизировать в расчете на аналитические паттерны доступа. Оказывается, обсуждавшиеся в первой части этой главы алгоритмы индексации отлично работают для OLTP, но не так хороши при выдаче ответов на аналитические запросы.

В оставшейся части главы мы рассмотрим подсистемы хранения, оптимизированные для целей анализа.

Различия между базами данных OLTP и складами данных. Модель склада данных чаще всего реляционная, поскольку SQL в целом отлично подходит для аналитических запросов. Существует множество графических утилит для анализа данных, которые генерируют SQL-запросы, визуализируют результаты и обеспечивают возможность аналитикам изучать данные (с помощью таких операций, как *углубление в данные* (drill-down), а также *продольных/поперечных, плоскостных и объемных срезов* (slicing and dicing)).

На первый взгляд склады данных и реляционные базы данных OLTP выглядят похоже, поскольку и у тех и у других есть SQL-интерфейс для запросов. Однако внутреннее устройство этих систем может быть совершенно различным, ведь они оптимизированы под совсем разные паттерны запросов. Многие производители БД сейчас концентрируются на поддержке или обработке транзакций, или аналитических задач, но не обоих сразу.

Некоторые СУБД, например Microsoft SQL Server и SAP HANA, поддерживают как обработку транзакций, так и складирование данных в одном программном продукте. Но они постепенно превращаются в две отдельные подсистемы хранения и выполнения запросов, просто доступных через общий SQL-интерфейс [49–51].

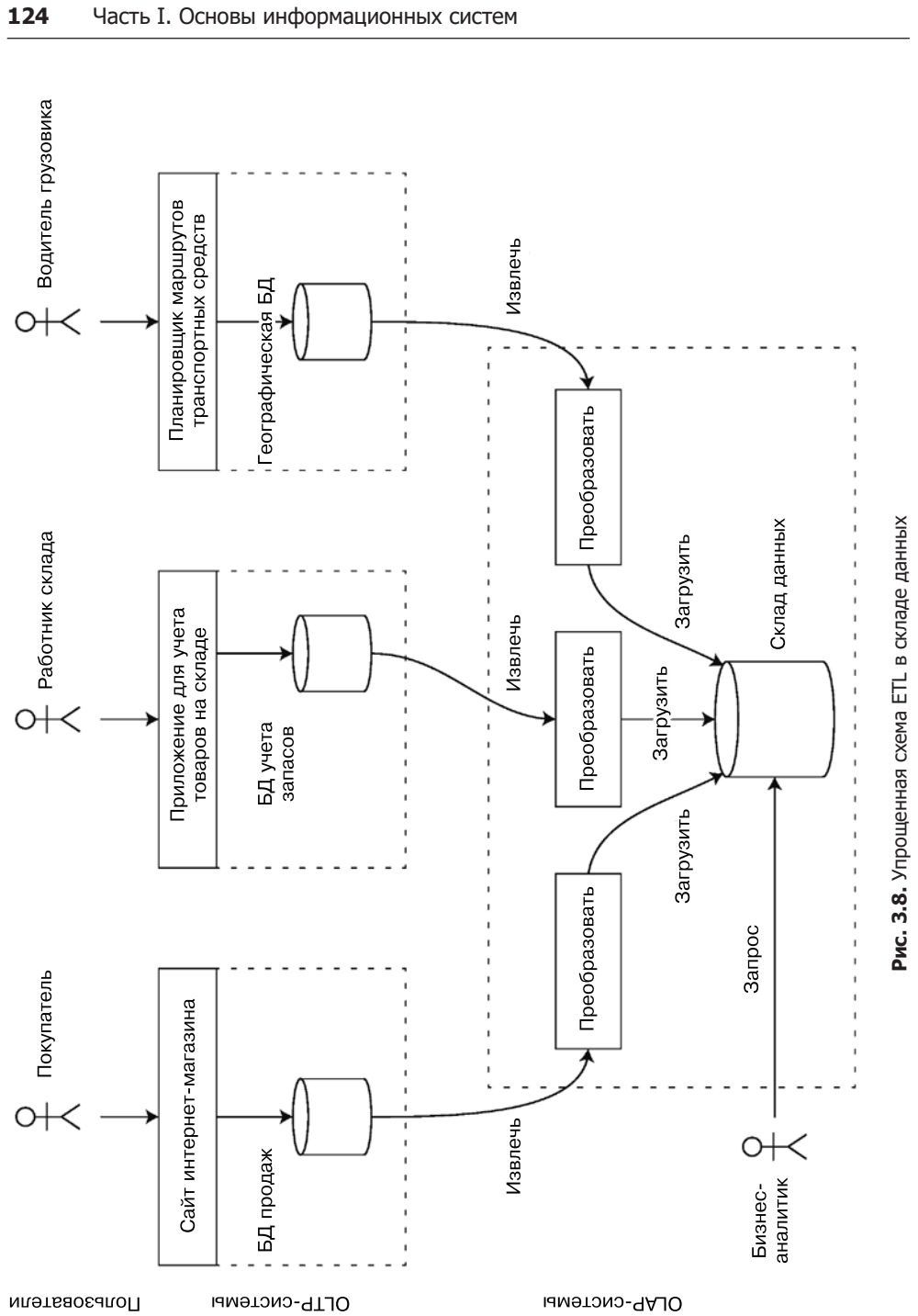


Рис. 3.8. Упрощенная схема ETL в складе данных

Такие производители складов данных, как Teradata, Vertica, SAP HANA и ParAccel, обычно продают свои системы под дорогостоящими коммерческими лицензиями. Amazon RedShift — хостируемая версия ParAccel. В последнее время появилось множество проектов SQL на базе Hadoop с открытым исходным кодом, они еще незрелые, но уже пытаются конкурировать с коммерческими системами складов данных. Среди них: Apache Hive, Spark SQL, Cloudera Impala, Facebook Presto, Apache Tajo и Apache Drill [52, 53]. Некоторые из них основаны на идеях, заимствованных у системы Dremel компании Google [54].

«Звезды» и «снежинки»: схемы для аналитики

Как показывалось в главе 2, в сфере обработки транзакций используется множество различных моделей данных в зависимости от потребностей приложения. С другой стороны, в аналитике разнообразие моделей намного меньше. Множество складов данных применяются довольно шаблонным образом, известным под названием *«схема "звезда"»* (star schema, также *моделирование с помощью измерений* (dimensional modeling) [55]).

Образец схемы, приведенный на рис. 3.9, демонстрирует склад данных, который мог бы встретиться, например, у розничного торговца бакалейно-гастрономическими товарами. В центре схемы находится так называемая *таблица фактов* (здесь она называется `fact_sales`). Каждая строка таблицы отражает событие, произошедшее в конкретный момент времени (в данном случае приобретение покупателем товара). Если бы мы анализировали не розничные продажи, а трафик сайта, то каждая строка могла бы отражать просмотр страницы или щелчок пользователя.

Обычно факты поступают в виде отдельных событий, поскольку таким образом обеспечивается максимальная гибкость дальнейшего анализа. Однако это значит, что таблица фактов может вырасти до чрезвычайно больших размеров. У крупных корпораций, таких как Apple, Walmart или eBay, в складах данных могут храниться десятки петабайт истории транзакций, и большую часть их составляют таблицы фактов [56].

Отдельные столбцы таблиц фактов представляют собой атрибуты, такие как цена, по которой был продан товар, и стоимость его закупки у поставщика (благодаря чему можно вычислить размер прибыли). Другие столбцы представляют собой внешние ключи к другим таблицам, именуемым *таблицами измерений* (dimension table). В то время как строка в таблице фактов соответствует событию, измерения соответствуют *«кто»*, *«что»*, *«где»*, *«когда»*, *«как»* и *«почему»* этого события.

Например, на рис. 3.9 одно из измерений — проданный товар. Каждая строка в таблице `dim_product` отражает один из видов продаваемых товаров, включая его единицу хранения (stock keeping unit, SKU), описание, название марки, категорию, содержание жира, размер упаковки и т. д. Все строки таблицы `fact_sales` используют внешние ключи для указания на то, какой товар был продан в данной конкретной торговой операции (для простоты, если покупатель приобретает несколько различных товаров за один раз, то они отражаются отдельными строками таблицы фактов).

Таблица dim_product

product_sk	sku	description	brand	category
30	OK4012	Bananas	Freshmax	Fresh fruit
31	KA9511	Fish food	Aquatech	Pet supplies
32	AB1234	Croissant	Dealicious	Bakery

Таблица dim_store

store_sk	state	city
1	WA	Seattle
2	CA	San Francisco
3	CA	Palo Alto

Таблица fact_sales

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	31	3	NULL	NULL	1	2.49	2.49
140102	69	5	19	NULL	3	14.99	9.99
140102	74	3	23	191	1	4.49	3.89
140102	33	8	NULL	235	4	0.99	0.99

Таблица dim_date

date_key	year	month	day	weekday	is_holiday
140101	2014	jan	1	wed	yes
140102	2014	jan	2	thu	no
140103	2014	jan	3	fri	no

Таблица dim_customer

customer_sk	name	date_of_birth
190	Alice	1979-03-29
191	Bob	1961-09-02
192	Cecil	1991-12-13

Таблица dim_promotion

promotion_sk	name	ad_type	coupon_type
18	New Year sale	Poster	NULL
19	Aquarium deal	Direct mail	Leaflet
20	Coffee & cake bundle	In-store sign	NULL

Рис. 3.9. Пример применения в складе данных схемы «звезда»

Даже дата и время часто отражаются с помощью таблиц измерений, поскольку это позволяет кодировать дополнительную информацию о датах (например, является ли день государственным праздником); это дает возможность различать в запросах продажи по выходным и в будние дни.

Название «схема “звезда”» возникло потому, что при визуализации связей таблиц таблица фактов находится посередине, окруженная таблицами измерений, а связи с этими таблицами напоминают лучи звезды.

Разновидность этого шаблона известна под названием «*схемы "снежинка"*». В ней измерения далее разбиваются на подызмерения. Например, в ней могут быть отдельные таблицы для марок и категорий товаров, а каждая строка в таблице `dim_product` может ссылаться на марку и категорию в виде внешних ключей вместо хранения их в виде строк в таблице `dim_product`. Схемы «снежинки» более нормализованы, чем схемы «звезды», но последние используются чаще, поскольку с ними удобнее работать аналитикам [55].

В типичном складе данных таблицы часто очень широкие: в таблицах фактов может быть более 100 столбцов, иногда даже несколько сотен [51]. Таблицы измерений тоже могут быть очень широкими, так как включают все метаданные, которые могут пригодиться при анализе. Например, таблица `dim_store` способна содержать подробную информацию о том, какие услуги предоставляются в каждом из магазинов, есть ли там внутри магазинная пекарня, какова площадь помещения, дату открытия магазина, дату его реконструкции, расстояние от ближайшего шоссе и т. п.

3.3. Столбцовое хранилище

Если в таблицах фактов содержатся триллионы строк и петабайты данных, эффективное хранение и выполнение запросов становится непростой задачей. Таблицы измерений обычно намного меньше (миллионы строк), так что в этом разделе мы сосредоточимся в основном на хранилищах фактов.

Хотя ширина таблиц фактов часто превышает 100 столбцов, типичный запрос к складу данных обращается только к четырем или пяти из них за раз (запросы типа `"SELECT *"` для анализа данных бывают нужны редко) [51]. Рассмотрим запрос в примере 3.1: он обращается к большому количеству строк (каждая покупка кем-либо фрукта или конфет за 2013 календарный год), но только к трем столбцам таблицы `fact_sales`: `date_key`, `product_sk` и `quantity`. Все остальные столбцы запрос игнорирует.

Пример 3.1. Выясняем, что люди более склонны покупать — свежие фрукты или конфеты, в зависимости от дня недели

```
SELECT
  dim_date.weekday, dim_product.category,
  SUM(fact_sales.quantity) AS quantity_sold
FROM fact_sales
  JOIN dim_date    ON fact_sales.date_key   = dim_date.date_key
  JOIN dim_product ON fact_sales.product_sk = dim_product.product_sk
WHERE
  dim_date.year = 2013 AND
  dim_product.category IN ('Fresh fruit', 'Candy')
GROUP BY
  dim_date.weekday, dim_product.category;
```

Как выполнить этот запрос эффективно?

В большинстве баз данных OLTP хранилище располагается *построчно*: все значения из одной строки таблицы хранятся рядом друг с другом. Документоориентированные БД устроены аналогично: весь документ обычно хранится в виде непрерывной последовательности байтов. Это можно увидеть в CSV-примере на рис. 3.1.

Чтобы выполнить такой запрос, как в примере 3.1, нам понадобятся индексы по столбцам `fact_sales.date_key` и/или `fact_sales.product_sk`, которые бы сообщали подсистеме хранения, где искать все продажи для конкретной даты или конкретного товара. Но даже после этого реализованная построчно подсистема хранения должна загрузить все строки (каждая из которых состоит из более чем 100 атрибутов) с диска в оперативную память, выполнить их синтаксический разбор и отфильтровать те, что не удовлетворяют заданным условиям. На это может уйти много времени.

Идея *столбцовых хранилищ* проста: нужно хранить рядом значения не из одной строки, а из одного *столбца*. Если каждый столбец хранится в отдельном файле, то запросу требуется только прочитать и выполнить синтаксический разбор необходимых запросов столбцов, что может сэкономить массу усилий. Эта идея проиллюстрирована на рис. 3.10.

Таблица fact_sales

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	69	4	NULL	NULL	1	13.99	13.99
140102	69	5	19	NULL	3	14.99	9.99
140102	69	5	NULL	191	1	14.99	14.99
140102	74	3	23	202	5	0.99	0.89
140103	31	2	NULL	NULL	1	2.49	2.49
140103	31	3	NULL	NULL	3	14.99	9.99
140103	31	3	21	123	1	49.99	39.99
140103	31	8	NULL	233	1	0.99	0.99

Размещение данных по столбцам:

date_key file contents:	140102, 140102, 140102, 140102, 140103, 140103, 140103, 140103
product_sk file contents:	69, 69, 69, 74, 31, 31, 31, 31
store_sk file contents:	4, 5, 5, 3, 2, 3, 3, 8
promotion_sk file contents:	NULL, 19, NULL, 23, NULL, NULL, 21, NULL
customer_sk file contents:	NULL, NULL, 191, 202, NULL, NULL, 123, 233
quantity file contents:	1, 3, 1, 5, 1, 3, 1, 1
net_price file contents:	13.99, 14.99, 14.99, 0.99, 2.49, 14.99, 49.99, 0.99
discount_price file contents:	13.99, 9.99, 14.99, 0.89, 2.49, 9.99, 39.99, 0.99

Рис. 3.10. Хранение реляционных данных по столбцам, а не по строкам



Столбцовые хранилища легче всего понять в случае реляционной модели данных, но они могут использоваться и для нереляционных данных. Например, Parquet [57], основанный на системе Dremel компании Google [54], — столбцовый формат хранения, поддерживающий документную модель данных.

Размещение данных по столбцам требует, чтобы файлы всех столбцов содержали строки в одинаковом порядке. Следовательно, при необходимости собрать воедино целую строку можно взять из всех файлов отдельных столбцов 23-й элемент и собрать их вместе для формирования 23-й строки таблицы.

Сжатие столбцов

Помимо загрузки с диска только тех столбцов, которые нужно для запроса, можно еще более снизить требования к пропускной способности диска, сжав данные. К счастью, столбцовое хранилище часто очень хорошо поддается сжатию.

Рассмотрим последовательность значений для каждого столбца на рис. 3.10: похоже, что они часто повторяются, и это хороший знак в смысле возможности сжатия. В зависимости от содержащихся в столбце данных применяются различные методы сжатия. Один из методов, особенно эффективных при работе со складами данных, — *кодирование с помощью битовой карты* (bitmap encoding), показанное на рис. 3.11.

Значения столбцов:

product_sk:	69	69	69	69	74	31	31	31	31	29	30	30	31	31	31	68	69	69

Битовая карта для всех возможных значений:

product_sk = 29:	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
product_sk = 30:	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
product_sk = 31:	0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	0	0	0
product_sk = 68:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
product_sk = 69:	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
product_sk = 74:	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Кодирование длин серий:

product_sk = 29:	9, 1	(9 zeros, 1 one, rest zeros)
product_sk = 30:	10, 2	(10 zeros, 2 ones, rest zeros)
product_sk = 31:	5, 4, 3, 3	(5 zeros, 4 ones, 3 zeros, 3 ones, rest zeros)
product_sk = 68:	15, 1	(15 zeros, 1 one, rest zeros)
product_sk = 69:	0, 4, 12, 2	(0 zeros, 4 ones, 12 zeros, 2 ones)
product_sk = 74:	4, 1	(4 zeros, 1 one, rest zeros)

Рис. 3.11. Сжатое, закодированное с помощью битовой карты представление отдельного столбца

Зачастую количество различных значений в столбце невелико по сравнению с числом строк (например, у розничного торговца могут быть миллиарды торговых операций, но только 10 000 различных товаров). Мы можем теперь превратить столбец, содержащий n различных значений, в n отдельных битовых карт: по одной карте для каждого значения, по одному биту для каждой строки. Бит равен 1, если в строке содержится это значение, 0 — если нет.

При очень малом n (например, в столбце *country* (страна) может быть только примерно 200 различных значений) такие битовые карты могут хранить по одному биту на строку. Но если n намного больше, то в большинстве битовых карт будет множество нулей (говорят, что они *разреженные*). В этом случае битовые карты можно дополнительно кодировать с помощью алгоритма кодирования длин серий, как показано внизу рис. 3.11.

Подобные битовые индексы отлично подходят для часто встречающихся в складах данных типов запросов. Например:

`WHERE product_sk IN (30, 68, 69):`

> Загрузить три битовые карты для `product_sk = 30`, `product_sk = 68` и `product_sk = 69` и вычислить поразрядное *ИЛИ* этих трех битовых карт, что можно сделать весьма эффективно.

`WHERE product_sk = 31 AND store_sk = 3:`

> Загрузить битовые карты для `product_sk = 31` и `store_sk = 3` и вычислить поразрядное *И*. Это работает, поскольку столбцы содержат строки в одинаковом порядке, вследствие чего k-й бит в битовой карте одного столбца соответствует той же строке, что и k-й бит в битовой карте другого столбца.

Существуют также другие схемы сжатия для различных видов данных, но мы не станем в них углубляться — см. их обзор в [58].



Столбцовые хранилища и семейства столбцов

В СУБД Cassandra и HBase существует понятие семейств столбцов (column-family), которое они унаследовали от модели данных Bigtable [9]. Однако называть их столбцовыми означало бы вводить себя в заблуждение: внутри каждого семейства столбцов все столбцы из строки хранятся вместе наряду с ключом строки и сжатие столбцов там не используется. Следовательно, модель данных Bigtable является в основном построчной.

Пропускная способность памяти и векторизованная обработка. Существенным узким местом для запросов складов данных, которым приходится просматривать миллионы строк, выступает пропускная способность перемещения данных из диска в память. Однако это не единственное такое место. Кроме того, заботой разработчиков аналитических БД являются эффективное использование скорости передачи данных из оперативной памяти в кэш процессора, избегание разными способами

ошибочного прогнозирования ветвей и «пузырьков» в конвейере обработки инструкций CPU, а также применение векторных инструкций (single-instruction-multi-data, SIMD) современных процессоров [59, 60].

Помимо снижения объемов загружаемых с диска данных, столбцовые схемы хранилищ также могут послужить для более эффективного использования циклов CPU. Например, подсистема обработки запросов может взять порцию данных, которая бы хорошо помещалась в L1-кэше процессора, и проходить по ней в сплошном цикле (то есть без вызовов функций). Процессор способен выполнять такой цикл намного быстрее, чем код, содержащий множество вызовов функций и условий для каждой обрабатываемой записи. Сжатие столбцов позволяет поместить в том же объеме L1-кэша больше строк для одного столбца. Для работы напрямую с такими порциями сжатых столбцовых данных можно задействовать такие операторы, как описанные выше побитовые *ИЛИ* и *И*. Этот метод известен под названием *векторизованной обработки* [58, 49].

Порядок сортировки в столбцовом хранилище

В столбцовом хранилище порядок хранения строк может не иметь значения. Проще всего хранить их в порядке их вставки, ведь при этом вставка новой строки будет означать просто дописывание в конец всех файлов столбцов. Однако можно и навязать какой-либо порядок аналогично тому, как мы поступили с SS-таблицами ранее, и использовать его в качестве механизма индексации.

Обратите внимание: сортировать каждый столбец отдельно смысла нет, поскольку тогда не будет известно, какие элементы столбцов относятся к одной строке. Мы сможем воссоздавать строки только потому, что знаем: k-й элемент одного столбца и k-й элемент другого соответствуют той же строке.

Вместо этого необходимо сортировать сразу целые строки, несмотря на то что данные хранятся по столбцам. Администратор БД может выбрать столбцы, по которым требуется сортировать таблицу, основываясь на знаниях о том, какие запросы выполняются чаще всего. Например, если запросы часто производятся по диапазонам дат, таким как «прошлый месяц», то имеет смысл сделать первым ключом сортировки столбец `date_key`. Тогда оптимизатор запросов сможет просматривать только строки, относящиеся к прошлому месяцу, что будет намного быстрее, чем просматривать все строки.

Второй столбец может определять порядок сортировки строк, значение которых в первом столбце одинаково. Например, если на рис. 3.10 первый ключ сортировки — `date_key`, то имеет смысл сделать вторым ключом `product_sk`, чтобы все продажи одного товара в один день группировались в хранилище. Это поможет при выполнении запросов, для которых необходимо группировать или отфильтровывать продажи по товару в пределах определенного диапазона дат.

Еще одно преимущество сортировки — то, что она способна помочь при сжатии столбцов. Если в основном столбце сортировки количество различных значений

невелико, то после сортировки в нем появятся длинные последовательности повторяющихся одинаковых значений в строке. Простое кодирование длин серий, аналогичное тому, которое мы использовали для битовых карт на рис. 3.11, может сжать такой столбец до нескольких килобайтов — даже при наличии в таблице миллиардов строк.

Сжатие лучше всего работает для первого ключа сортировки. Второй и третий ключи будут сильнее перемешаны, а следовательно, в них не будет таких длинных серий повторяющихся значений. Расположенные далее в списке сортировки столбцы окажутся, по сути, в случайном порядке, так что вряд ли будут сжаты очень хорошо. Но и сортировка первых нескольких столбцов приносит немалые плоды.

Несколько различных порядков сортировки. Удачное развитие этой идеи было предложено в C-Store и использовано в коммерческом складе данных Vertica [61, 62]. Для различных запросов лучше подходят разчные порядки сортировки, так почему бы не хранить одни и те же данные отсортированными *несколькими различными* способами? Все равно ведь приходится реплицировать данные по нескольким машинам во избежание их потери в случае сбоя одной из них. Вполне допустимо хранить эти избыточные данные отсортированными различными способами, чтобы при обработке запроса можно было использовать ту их версию, которая лучше всего подходит для конкретного запроса.

Наличие нескольких порядков сортировки в столбцовом хранилище слегка напоминает ряд вторичных индексов в построчном хранилище. Но важное различие состоит в том, что в построчном хранилище каждая строка хранится в одном месте (в неупорядоченном файле или кластеризованном индексе) и вторичные индексы лишь содержат указатели на соответствующие строки. В столбцовом хранилище же обычно нет никаких указателей на данные, только содержащие значения столбцы.

Запись в столбцовое хранилище

Описанные оптимизации имеют смысл в складах данных, поскольку большая часть нагрузки состоит из выполняемых аналитиками больших запросов, предназначенных только для чтения данных. Столбцовое хранение, сжатие и сортировка служат для ускорения выполнения этих запросов. Однако у них есть недостаток в виде усложнения операций записи.

Подход с обновлением данных на месте, используемый B-деревьями, невозможен в случае сжатых столбцов. При необходимости вставить строки в середине отсортированной таблицы, вероятнее всего, придется переписать все файлы столбцов. Вставка должна обновлять все столбцы согласованным образом, ведь строки определяются по их позиции в столбце.

К счастью, ранее мы уже видели хорошее решение этой проблемы: LSM-деревья. Все записывается сначала в хранилище в оперативной памяти, где данные добавляются в отсортированную структуру и подготавливаются к записи на диск.

Не имеет значения, является хранилище в оперативной памяти столбцовым или построчным. При накоплении достаточного количества записываемых данных они объединяются с файлами столбцов на диске и записываются блоками в новые файлы. По сути, именно так и функционирует Vertica [62].

Запросы должны просматривать как столбцовые данные на диске, так и свежие данные в оперативной памяти и объединять их. Однако оптимизатор запросов скрывает этот нюанс от пользователя. С точки зрения аналитика, изменяемые с помощью вставок, обновлений или удалений данные сразу же отражаются в последующих запросах.

Агрегирование: кубы данных и материализованные представления

Не все склады данных являются столбцовыми хранилищами: используются и обычные построчные БД, а также несколько других архитектур. Однако столбцовые хранилища работают намного быстрее при произвольных аналитических запросах, так что их популярность быстро растет [51, 63].

Другой нюанс складов данных, который стоит вскользь упомянуть: *материализованные сводные показатели*. Как уже обсуждалось ранее, запросы к складам часто включают функции агрегирования, такие как `COUNT`, `SUM`, `AVG`, `MIN` или `MAX` в языке SQL. Если во множестве различных запросов используются одни и те же функции агрегирования, то было бы расточительством каждый раз «перемалывать» необработанные данные. Почему бы не кэшировать часть сводных показателей, применяемых в запросах чаще всего?

Один из способов создания подобного кэша — *материализованное представление* (materialized view). В реляционной модели данных оно часто описывается аналогично обычному (виртуальному) представлению: напоминающий таблицу объект, чье содержимое является результатом какого-то запроса. Различие состоит в том, что материализованное представление является фактической копией результатов запроса, записанной на диск, в то время как виртуальное представление — просто сокращенная форма записи для написания запросов. При чтении из виртуального представления движок SQL динамически разворачивает его в лежащий в его основе запрос и затем выполняет этот развернутый запрос.

При изменении основополагающих данных необходимо обновить материализованное представление, поскольку оно представляет собой денормализованную копию этих данных. БД может выполнять обновления автоматически, но подобные манипуляции увеличивают стоимость операций записи, так что материализованные представления редко используются в базах данных OLTP. В складах данных же, где основная нагрузка состоит из операций чтения, применять их имеет смысл (улучшают ли они на самом деле производительность операций чтения, зависит от конкретного случая).

Распространенный особый случай материализованного представления — *куб данных* (data cube), или *OLAP-куб* (OLAP cube) [64]. Он представляет собой сетку сводных показателей, сгруппированных по различным измерениям. На рис. 3.12 показан его пример.

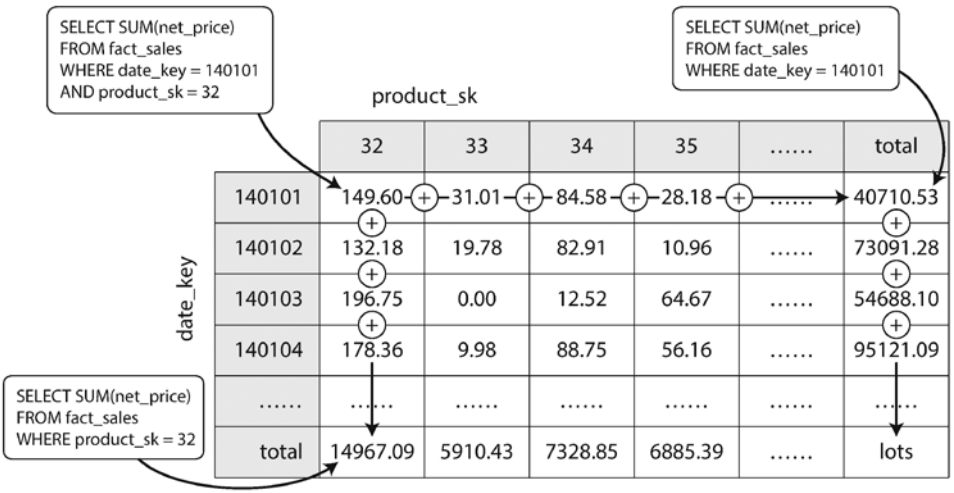


Рис. 3.12. Два измерения куба данных, агрегирующих данные путем суммирования

Представьте, что у каждого факта имеются внешние ключи только к двум таблицам измерений — на рис. 3.12 это *дата* и *товар*. Можно нарисовать двумерную таблицу: с товарами по одной оси координат и датами — по другой. Каждая ячейка содержит сводный показатель (например, `SUM`) атрибута (например, `net_price`) всех фактов с таким сочетанием даты и товара. Затем можно применить ту же агрегирующую функцию вдоль каждой строки или столбца и получить итоги, сокращенные на одно измерение (продажи по товарам независимо от даты или продажи по датам независимо от товара).

Вообще говоря, у фактов часто бывает больше двух измерений. На рис. 3.9 есть пять измерений: дата, товар, магазин, рекламная кампания и покупатель. Представить пятимерный гиперкуб намного сложнее, но идея остается той же самой: каждая ячейка содержит продажи для соответствующего сочетания даты, товара, магазина, рекламной кампании и покупателя. Эти значения можно затем последовательно группировать по каждому из измерений.

Преимущество материализованных кубов состоит в том, что определенные запросы будут выполняться очень быстро, поскольку данные для них были, по сути, заранее вычислены. Например, если вам нужно узнать общий объем продаж за вчера по каждому магазину, то достаточно просто посмотреть на итоги по соответствующему измерению — нет никакой необходимости просматривать миллионы строк.

Недостаток же заключается в том, что кубы данных не имеют такой же гибкости, как запросы к исходным данным. Например, тут нельзя подсчитать, какая доля продаж приходится на товары, стоящие менее $100, поскольку цена не является одним из измерений. Большинство складов данных поэтому стараются хранить как можно больше исходных данных и использовать сводные показатели, такие как кубы данных, только в качестве ускорителя производительности для определенных запросов.

3.4. Резюме

В этой главе мы попытались разобраться в том, как БД хранят и извлекают данные. Что же происходит при сохранении данных в базе и что делает база, когда вы позднее запрашиваете данные?

На высоком уровне абстракции мы видим, что подсистемы хранения делятся на две широкие категории: оптимизированные для обработки транзакций (OLTP) и оптимизированные для аналитики (OLAP). Между паттернами доступа в этих сценариях использования есть большие различия.

- OLTP-системы обычно нацелены на работу с пользователями, и это означает огромное потенциальное количество запросов. Чтобы справиться с такой нагрузкой, приложения обычно затрагивают в каждом запросе только небольшое число строк. Программы запрашивают записи с помощью определенного ключа, а подсистема хранения задействует индекс для поиска данных с соответствующим ключом. Узким местом здесь обычно становится время перехода к нужной позиции на диске.
- Склады данных и подобные им аналитические системы менее широко известны, поскольку они в основном применяются бизнес-аналитиками, а не конечными пользователями. Склады обрабатывают намного меньшее количество запросов, чем OLTP-системы, но все запросы обычно очень ресурсоемки и требуют просмотра миллионов строк за короткое время. Узким место здесь обычно становится пропускная способность диска (а не время перехода к нужной позиции на нем). Все большую популярность для этой разновидности задач приобретают столбцовые хранилища.

С точки зрения OLTP мы рассмотрели два различных подхода к подсистемам хранения.

- Журналированный подход, при котором допускается только дописывание данных в файлы и удаление устаревших файлов, а не обновление записанного файла. Сюда относятся: Bitcask, SS-таблицы, LSM-деревья, LevelDB, HBase, Lucene и другие.
- Подход с обновлением на месте, при котором диск рассматривается как набор страниц заданного размера, допускающих перезапись. Крупнейший представитель этой философии — B-деревья, используемые во всех основных реляционных базах данных, а также и во многих нереляционных.

Журналированные подсистемы хранения — относительно недавнее изобретение. Их основная идея состоит в систематическом преобразовании произвольной записи на диск в последовательную, что обеспечивает более высокую пропускную способность по записи вследствие характеристик производительности жестких дисков и твердотельных накопителей.

Заканчивая обзор OLTP-подхода, мы вкратце рассмотрели некоторые более сложные индексные структуры, а также базы, оптимизированные для хранения всех данных в оперативной памяти.

Далее мы перешли от рассмотрения внутреннего устройства подсистем хранения к высокоуровневой архитектуре типичного склада данных. Это позволило проиллюстрировать, почему аналитические задачи настолько отличаются от OLTP: когда для запросов необходим последовательный просмотр большого количества строк, индексы не играют особой роли. Вместо этого становится гораздо важнее кодировать данные очень компактно, чтобы минимизировать объем читаемых с диска данных. Мы обсудили, как столбцовые хранилища помогают достичь поставленной цели.

Вооружившись знаниями о внутреннем устройстве подсистем хранения, вы, как разработчик приложений, можете гораздо лучше разобраться в том, какой инструмент больше подойдет для вашего конкретного приложения. Если вам понадобится настроить параметры БД, то эти знания позволят понять, что повлечет изменение параметра в большую или меньшую сторону.

Хотя эта глава и не могла сделать из вас эксперта по настройке какой-либо конкретной подсистемы хранения, я надеюсь, наших идей и словаря будет достаточно для того, чтобы вы смогли разобраться в документации выбранной вами базы данных.

3.5. Библиография

1. *Aho A. V., Hopcroft J. E., Ullman J. D.* Data Structures and Algorithms. — Addison-Wesley, 1983.
2. *Cormen T. H., Leiserson C. E., Rivest R. L., Stein C.* Introduction to Algorithms, 3rd edition. — MIT Press, 2009.
3. *Sheehy J., Smith D.* Bitcask: A Log-Structured Hash Table for Fast Key/Value Data // Basho Technologies, April 2010 [Электронный ресурс]. — Режим доступа: http://basho.com/wp-content/uploads/2015/05/bitcask-intro.pdf.
4. *Li Y., He B., Yang R. J., et al.* Tree Indexing on Solid State Drives // Proceedings of the VLDB Endowment, volume 3, number 1, pages 1195–1206, September 2010 [Электронный ресурс]. — Режим доступа: http://www.vldb.org/pvldb/vldb2010/papers/R106.pdf.
5. *Graefe G.* Modern B-Tree Techniques // Foundations and Trends in Databases, volume 3, number 4, pages 203–402, August 2011 [Электронный ресурс]. — Режим доступа: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.219.7269&rep=rep1&type=pdf.

6. *Dean J., Ghemawat S.* LevelDB Implementation Notes [Электронный ресурс]. — Режим доступа: https://github.com/google/leveldb/blob/master/doc/impl.html.
7. *Borthakur D.* The History of RocksDB. November 24, 2013 [Электронный ресурс]. — Режим доступа: http://rocksdb.blogspot.com.by/.
8. *Bertozzi M.* Apache HBase I/O — HFile. June, 29 2012 [Электронный ресурс]. — Режим доступа: http://blog.cloudera.com/blog/2012/06/hbase-io-hfile-input-output/.
9. *Chang F., Dean J., Ghemawat S., et al.* Bigtable: A Distributed Storage System for Structured Data // 7th USENIX Symposium on Operating System Design and Implementation (OSDI), November 2006 [Электронный ресурс]. — Режим доступа: https://research.google.com/archive/bigtable.html.
10. *O'Neil P., Cheng E., Gawlick D., O'Neil E.* The LogStructured Merge-Tree (LSM-Tree) // Acta Informatica, volume 33, number 4, pages 351–385, June 1996 [Электронный ресурс]. — Режим доступа: https://www.cs.umb.edu/~poneil/lsmtree.pdf.
11. *Rosenblum M., Ousterhout J. K.* The Design and Implementation of a Log-Structured File System // ACM Transactions on Computer Systems, volume 10, number 1, pages 26–52, February 1992 [Электронный ресурс]. — Режим доступа: http://research.cs.wisc.edu/areas/os/Qual/papers/lfs.pdf.
12. *Grand A.* What Is in a Lucene Index? // Lucene/Solr Revolution, November 14, 2013 [Электронный ресурс]. — Режим доступа: https://www.slideshare.net/lucene-revolution/what-is-inaluceneagrandfinal.
13. *Kandepet D.* Hacking Lucene — The Index Format. October 1, 2011 [Электронный ресурс]. — Режим доступа: http://hackerlabs.github.io/blog/2011/10/01/hacking-lucene-the-index-format/index.html.
14. *McCandless M.* Visualizing Lucene's Segment Merges. February 11, 2011 [Электронный ресурс]. — Режим доступа: http://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html.
15. *Bloom B. H.* Space/Time Trade-offs in Hash Coding with Allowable Errors // Communications of the ACM, volume 13, number 7, pages 422–426, July 1970 [Электронный ресурс]. — Режим доступа: http://www.cs.upc.edu/~diaz/p422-bloom.pdf.
16. Operating Cassandra: Compaction // Apache Cassandra Documentation v4.0, 2016 [Электронный ресурс]. — Режим доступа: https://cassandra.apache.org/doc/latest/operating/compaction.html.
17. *Bayer R., McCreight E. M.* Organization and Maintenance of Large Ordered Indices // Boeing Scientific Research Laboratories, Mathematical and Information Sciences Laboratory, report no. 20, July 1970 [Электронный ресурс]. — Режим доступа: http://www.dtic.mil/cgi-bin/GetTRDoc?AD=AD0712079.
18. *Comer D.* The Ubiquitous B-Tree // ACM Computing Surveys, volume 11, number 2, pages 121–137, June 1979 [Электронный ресурс]. — Режим доступа: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.96.6637&rep=rep1&type=pdf.
19. *Goossaert E.* Coding for SSDs. February 12, 2014 [Электронный ресурс]. — Режим доступа: http://codecapsule.com/2014/02/12/coding-for-ssds-part-1-introduction-and-table-of-contents/.

20. *Mohan C., Levine F.* ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging // ACM International Conference on Management of Data (SIGMOD), June 1992 [Электронный ресурс]. — Режим доступа: http://www.ics.uci.edu/~cs223/papers/p371-mohan.pdf.
21. *Chu H.* LDAP at Lightning Speed // Build Stuff '14, November 2014 [Электронный ресурс]. — Режим доступа: https://buildstuff14.sched.com/event/08a1a368e272eb599a52e08b4c3c779d.
22. *Kuszmaul B. C.* A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees. April 22, 2014 [Электронный ресурс]. — Режим доступа: https://insideanalysis.com/.
23. *Athanassoulis M., Kester M. S., Maas L. M., et al.* Designing Access Methods: The RUM Conjecture // 19th International Conference on Extending Database Technology (EDBT), March 2016 [Электронный ресурс]. — Режим доступа: http://openproceedings.org/2016/conf/edbt/paper-12.pdf.
24. *Zaitsev P.* Innodb Double Write. August 4, 2006 [Электронный ресурс]. — Режим доступа: https://www.percona.com/blog/2006/08/04/innodb-double-write/.
25. *Vondra T.* On the Impact of Full-Page Writes. November 23, 2016 [Электронный ресурс]. — Режим доступа: https://blog.2ndquadrant.com/on-the-impact-of-full-page-writes/.
26. *Callaghan M.* The Advantages of an LSM vs a B-Tree. January 19, 2016 [Электронный ресурс]. — Режим доступа: http://www.codemesh.io/codemesh/mark-callaghan.
27. *Callaghan M.* Choosing Between Efficiency and Performance with RocksDB // Code Mesh, November4, 2016 [Электронный ресурс]. — Режим доступа: http://www.codemesh.io/codemesh/mark-callaghan.
28. *Mutsuzaki M.* MySQL vs. LevelDB // github.com, August 2011 [Электронный ресурс]. — Режим доступа: https://github.com/m1ch1/mapkeeper/wiki/MySQL-vs.-LevelDB.
29. *Coverston B., Ellis J., et al.* CASSANDRA-1608: Redesigned Compaction. July 2011 [Электронный ресурс]. — Режим доступа: https://issues.apache.org/jira/browse/CASSANDRA-1608.
30. *Canadi I., Dong S., Callaghan M.* RocksDB Tuning Guide. 2016 [Электронный ресурс]. — Режим доступа: https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.
31. MySQL 5.7 Reference Manual. Oracle, 2014 [Электронный ресурс]. — Режим доступа: https://dev.mysql.com/doc/refman/5.7/en/.
32. Books Online for SQL Server 2012. Microsoft, 2012 [Электронный ресурс]. — Режим доступа: https://docs.microsoft.com/en-us/sql/sql-server/sql-server-technical-documentation.
33. *Webb J.* Using Covering Indexes to Improve Query Performance. 29 September 2008 [Электронный ресурс]. — Режим доступа: https://www.red-gate.com/simple-talk/sql/learn-sql-server/using-covering-indexes-to-improve-query-performance/.

34. *Ramsak F., Markl V., Fenk R., et al.* Integrating the UB-Tree into a Database System Kernel // 26th International Conference on Very Large Data Bases (VLDB), September 2000 [Электронный ресурс]. — Режим доступа: http://www.vldb.org/conf/2000/P263.pdf.

35. The PostGIS Development Group: PostGIS 2.1.2dev Manual. 2014 [Электронный ресурс]. — Режим доступа: http://postgis.net/docs/manual-2.1/.

36. *Escriva R., Wong B., Sirer E. G.* HyperDex: A Distributed, Searchable Key-Value Store // ACM SIGCOMM Conference, August 2012 [Электронный ресурс]. — Режим доступа: http://www.cs.princeton.edu/courses/archive/fall13/cos518/papers/hyperdex.pdf.

37. *McCandless M.* Lucene's FuzzyQuery Is 100 Times Faster in 4.0. March 24, 2011 [Электронный ресурс]. — Режим доступа: http://blog.mikemccandless.com/2011/03/lucenes-fuzzyquery-is-100-times-faster.html.

38. *Heinz S., Zobel J., Williams H. E.* Burst Tries: A Fast, Efficient Data Structure for String Keys // ACM Transactions on Information Systems, volume 20, number 2, pages 192–223, April 2002 [Электронный ресурс]. — Режим доступа: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.3499.

39. *Schulz K. U., Mihov S.* Fast String Correction with Levenshtein Automata // International Journal on Document Analysis and Recognition, volume 5, number 1, pages 67–85, November 2002 [Электронный ресурс]. — Режим доступа: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.652.

40. *Manning C. D., Raghavan P., Schütze H.* Introduction to Information Retrieval. Cambridge University Press, 2008.

41. *Stonebraker M., Madden S., Abadi D. J., et al.* The End of an Architectural Era (It's Time for a Complete Rewrite) // 33rd International Conference on Very Large Data Bases (VLDB), September 2007 [Электронный ресурс]. — Режим доступа: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.3697&rep=rep1&type=pdf.

42. VoltDB Technical Overview White Paper // VoltDB, 2014 [Электронный ресурс]. — Режим доступа: https://www.voltdb.com/resources/.

43. *Rumble S. M., Kejriwal A., Ousterhout J. K.* Log-Structured Memory for DRAM-Based Storage // 12th USENIX Conference on File and Storage Technologies (FAST), February 2014 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/system/files/conference/fast14/fast14-paper_rumble.pdf.

44. *Harizopoulos S., Abadi D. J., Madden S., Stonebraker M.* OLTP Through the Looking Glass, and What We Found There // ACM International Conference on Management of Data (SIGMOD), June 2008 [Электронный ресурс]. — Режим доступа: http://hstore.cs.brown.edu/papers/hstore-lookingglass.pdf.

45. *DeBrabant J., Pavlo A., Tu S., et al.* Anti-Caching: A New Approach to Database Management System Architecture // Proceedings of the VLDB Endowment, volume 6, number 14, pages 1942–1953, September 2013 [Электронный ресурс]. — Режим доступа: http://www.vldb.org/pvldb/vol6/p1942-debrabant.pdf.

46. *Arulraj J., Pavlo A., Dulloor S. R.* Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems // ACM International Conference on Management of Data (SIGMOD), June 2015 [Электронный ресурс]. — Режим доступа: http://www.pdl.cmu.edu/PDL-FTP/NVM/storage.pdf.

47. *Codd E. F., Codd S. B., Salley C. T.* Providing OLAP to User-Analysts: An IT Mandate // E. F. Codd Associates, 1993 [Электронный ресурс]. — Режим доступа: http://www.minet.uni-jena.de/dbis/lehre/ss2005/sem_dwh/lit/Cod93.pdf.

48. *Chaudhuri S., Dayal U.* An Overview of Data Warehousing and OLAP Technology // ACM SIGMOD Record, volume 26, number 1, pages 65–74, March 1997 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/sigrecord.pdf.

49. *Larson P.-Å., Clinciu C., Fraser C., et al.* Enhancements to SQL Server Column Stores // ACM International Conference on Management of Data (SIGMOD), June 2013 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/enhancements-to-sql-server-column-stores/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F193599%2Fapollo3%2520-%2520sigmod%25202013%2520-%2520final.pdf.

50. *Färber F., May N., Lehner W., et al.* The SAP HANA Database — An Architecture Overview // IEEE Data Engineering Bulletin, volume 35, number 1, pages 28–33, March 2012 [Электронный ресурс]. — Режим доступа: http://sites.computer.org/debull/A12mar/hana.pdf.

51. *Stonebraker M.* The Traditional RDBMS Wisdom Is (Almost Certainly) All Wrong // presentation at EPFL, May 2013 [Электронный ресурс]. — Режим доступа: http://slideshot.epfl.ch/talks/166.

52. *Abadi D. J.* Classifying the SQL-on-Hadoop Solutions. October 2, 2013 [Электронный ресурс]. — Режим доступа: https://web.archive.org/web/20150318031121/http://hadapt.com/blog/2013/10/02/classifying-the-sql-on-hadoop-solutions/.

53. *Kornacker M., Behm A., Bittorf V., et al.* Impala: A Modern, Open-Source SQL Engine for Hadoop // 7th Biennial Conference on Innovative Data Systems Research (CIDR), January 2015 [Электронный ресурс]. — Режим доступа: http://pandis.net/resources/cidr15impala.pdf.

54. *Melnik S., Gubarev A., Long J. J., et al.* Dremel: Interactive Analysis of Web-Scale Datasets // 36th International Conference on Very Large Data Bases (VLDB), pages 330–339, September 2010 [Электронный ресурс]. — Режим доступа: https://research.google.com/pubs/pub36632.html.

55. *Kimball R., Ross M.* The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling, 3rd edition. John Wiley & Sons, July 2013.

56. *Harris D.* Why Apple, eBay, and Walmart Have Some of the Biggest Data Warehouses You've Ever Seen. March 27, 2013 [Электронный ресурс]. — Режим доступа: https://gigaom.com/2013/03/27/why-apple-ebay-and-walmart-have-some-of-the-biggest-data-warehouses-youve-ever-seen/.

57. *Dem J. L.* Dremel Made Simple with Parquet. September 11, 2013 [Электронный ресурс]. — Режим доступа: https://blog.twitter.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet.html.

58. *Abadi D. J., Boncz P., Harizopoulos S., et al.* The Design and Implementation of Modern Column-Oriented Database Systems // Foundations and Trends in Databases, volume 5, number 3, pages 197–280, December 2013 [Электронный ресурс]. — Режим доступа: http://cs-www.cs.yale.edu/homes/dna/papers/abadi-column-stores.pdf.

59. *Boncz P., Zukowski M., Nes N.* MonetDB/X100: Hyper- Pipelining Query Execution // 2nd Biennial Conference on Innovative Data Systems Research (CIDR), January 2005 [Электронный ресурс]. — Режим доступа: http://www.cidrdb.org/cidr2005/papers/P19.pdf.

60. *Zhou J., Ross K. A.* Implementing Database Operations Using SIMD Instructions // ACM International Conference on Management of Data (SIGMOD), pages 145–156, June 2002 [Электронный ресурс]. — Режим доступа: http://www1.cs.columbia.edu/~kar/pubsk/simd.pdf.

61. *Stonebraker M., Abadi D. J., Batkin A., et al.* C-Store: A Column- oriented DBMS // 31st International Conference on Very Large Data Bases (VLDB), pages 553–564, September 2005 [Электронный ресурс]. — Режим доступа: http://www.vldb2005.org/program/paper/thu/p553-stonebraker.pdf.

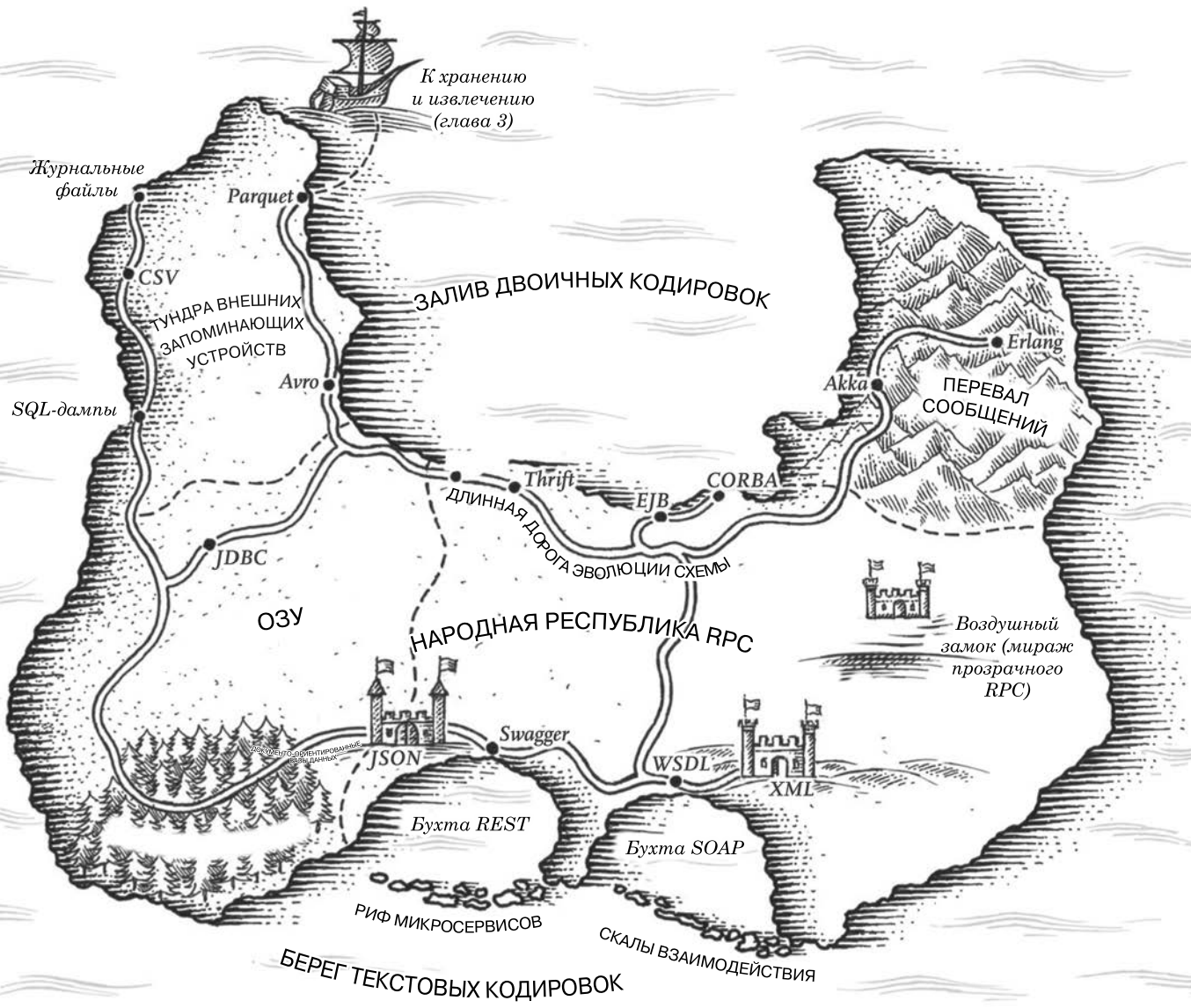
62. *Lamb A., Fuller M., Varadarajan R., et al.* The Vertica Analytic Database: C-Store 7 Years Later // Proceedings of the VLDB Endowment, volume 5, number 12, pages 1790–1801, August 2012 [Электронный ресурс]. — Режим доступа: http://vldb.org/pvldb/vol5/p1790_andrewlamb_vldb2012.pdf.

63. *Dem J. L., Li N.* Efficient Data Storage for Analytics with Apache Parquet 2.0 // Hadoop Summit, San Jose, June 2014 [Электронный ресурс]. — Режим доступа: https://www.slideshare.net/julienledem/th-210pledem.

64. *Gray J., Chaudhuri S., Bosworth A., et al.* Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals // Data Mining and Knowledge Discovery, volume 1, number 1, pages 29–53, March 2007 [Электронный ресурс]. — Режим доступа: https://arxiv.org/ftp/cs/papers/0701/0701155.pdf.

65. *Ахо А. В., Хопкрофт Д. Э., Ульман Д. Д.* Структуры данных и алгоритмы. — М.: Вильямс, 2016.

66. *Кормен Т. Х., Лейзерсон Ч., Ривест Р., Штайн К.*. Введение в алгоритмы. — М.: Вильямс, 2013. — 1328 с.


К хранению и извлечению (глава 3)
Журнальные файлы
Parquet
CSV
ТУНДРА ВНЕШНИХ ЗАПОМИНАЮЩИХ УСТРОЙСТВ
ЗАЛИВ ДВОИЧНЫХ КОДИРОВОК
Avro
SQL-дампы
Erlang
Akka
ПЕРЕВАЛ СООБЩЕНИЙ
Thrift
EJB
CORBA
ДЛИННАЯ ДОРОГА ЭВОЛЮЦИИ СХЕМЫ
JDBC
ОЗУ
НАРОДНАЯ РЕСПУБЛИКА RPC
Воздушный замок (мираж прозрачного RPC)
JSON
Swagger
WSDL
XML
Документо-ориентированные базы данных
Бухта REST
Бухта SOAP
РИФ МИКРОСЕРВИСОВ
СКАЛЫ ВЗАИМОДЕЙСТВИЯ
БЕРЕГ ТЕКСТОВЫХ КОДИРОВОК

4 Кодирование и эволюция

Все сущее движется, и ничто не остается на месте.

Гераклит Эфесский, в изложении Платона (диалог «Кратил»)

Приложения неизбежно меняются с течением времени. По мере выхода новых программных продуктов, более глубокого понимания требований пользователя или изменения обстоятельств добавляются новые возможности и меняются старые. В главе 1 мы познакомились с понятием *«возможность развития»* (evolvability): желательно стараться делать системы высокоадаптивными (см. подраздел «Возможность развития: облегчаем внесение изменений» раздела 1.4).

В большинстве случаев вносимые в приложение изменения требуют также изменений хранимых им данных: вероятно, понадобится захватывать новое поле или тип записи или представить старые данные по-новому.

Представленные в главе 2 модели данных по-разному решают эти проблемы. Реляционные БД, как правило, принимают за аксиому, что все данные в базе соответствуют единой схеме: хотя последняя и способна меняться (с помощью миграций схемы, то есть операторов `ALTER`), в любой момент времени действует одна схема. Напротив, БД типа «схема при чтении» (бессхемные) не навязывают схему данных, так что такая база может содержать смесь более старых и более новых форматов данных, записанных в разное время (см. пункт «Гибкость схемы в документной модели» подраздела «Реляционные и документоориентированные базы данных сегодня» раздела 2.1).

При изменении формата данных или схемы часто приходится вносить в код приложения соответствующие правки (например, при добавлении нового поля в запись

код приложения начинает читать и записывать это поле). Однако в большом приложении изменения кода не могут происходить по мановению волшебной палочки.

- В случае приложений на стороне сервера может понадобиться выполнить *плавающее обновление* (rolling upgrade), также называемое *поэтапным развертыванием* (staged rollout). При нем новая версия развертывается всего на нескольких узлах за раз и проверяется, работает ли она без проблем; так постепенно обходятся все узлы. Благодаря этому новые версии развертываются без простоя сервиса, что способствует более частым выпускам обновлений и повышает возможности развития.
- В случае приложений на стороне клиента разработчик отдан на милость пользователей, которые могут решить не устанавливать обновления в течение какого-либо времени.

Все описанное выше говорит об одном: старые и новые версии кода, а равно старые и новые форматы, потенциально могут сосуществовать в системе в один и тот же момент времени. Чтобы система могла работать без проблем, необходимо поддерживать совместимость в обоих направлениях:

- *обратная совместимость* — более новый код способен читать данные, записанные более старым;
- *прямая совместимость* — более старый код способен читать данные, записанные более новым.

Добиться обратной совместимости обычно несложно: будучи создателем более нового кода, вы знаете формат данных, записанных более старым, и можете явным образом обработать их (при необходимости просто сохранив старый код для их чтения). Добиться прямой совместимости иногда сложнее, поскольку для этого более старый код должен игнорировать добавляемые более новой версией данные.

В настоящей главе мы рассмотрим несколько форматов кодирования данных, включая JSON, XML, Protocol Buffers, Thrift и Avro. В частности, изучим, что они делают при изменении схемы, а также их поддержку систем, в которых вынуждены сосуществовать старые и новые данные и код. Затем обсудим, как эти форматы используются для данных и обмена сообщениями: в веб-сервисах, при передаче состояния представления (representational state transfer, REST), удаленных вызовах процедур (RPC) и в системах передачи сообщений, таких как акторы (actors) и очереди сообщений.

4.1. Форматы кодирования данных

Программы обычно имеют дело с данными в (как минимум) двух представлениях.

1. В памяти данные хранятся в объектах, структурах, списках, массивах, хеш-таблицах, деревьях и т. д. Эти структуры данных оптимизированы так, чтобы CPU мог эффективно обращаться к ним и манипулировать ими (обычно с помощью указателей).

2. При необходимости записать данные в файлы или переслать их по сети необходимо кодировать их в некую самостоятельную последовательность байтов (например, документ в формате JSON). Поскольку указатели для других процессов никакого смысла не имеют, такое представление в виде последовательности байтов выглядит совсем не так, как обычные структуры данных в памяти[1].

Следовательно, нам нужен способ преобразования между этими двумя представлениями. Преобразование из представления в памяти в последовательность байтов называется *кодированием* (encoding), или *сериализацией* (serialization), или *маршалингом* (marshalling), а обратная ему операция — *декодированием* (decoding), или *парсингом* (parsing), или *десериализацией* (deserialization), или *демаршалингом* (demarshalling)[2].



Терминологический конфликт

К сожалению, термин «сериализация» используется также в контексте транзакций (см. главу 7) с совершенно другим смыслом. Чтобы избежать применения нескольких значений для одного слова, мы остановимся в этой книге на термине «кодирование» (encoding), хотя «сериализация», возможно, более распространенный термин.

В силу распространенности данной задачи существует множество различных библиотек и форматов кодирования. Приведу их краткий обзор.

Форматы, ориентированные на конкретные языки

В множестве языков программирования есть встроенная поддержка кодирования объектов из памяти в последовательности байтов. Например, в языке Java есть пакет `java.io.Serializable` [1], в языке Ruby — Marshal [2], в языке Python — pickle [3] и т. д. Существует также множество сторонних библиотек, например библиотека Kryo для языка Java [4].

Эти библиотеки кодирования очень удобны, поскольку позволяют сохранять и возвращать в исходный вид объекты из памяти с помощью минимального количества кода. Однако у них есть несколько серьезных проблем.

- Кодирование зачастую привязано к конкретному языку программирования, и чтение полученных данных из программы на другом языке представляет собой очень непростую задачу. Храня или передавая данные в подобном формате, вы

[1] За исключением некоторых особых случаев, например определенных проецируемых в память файлов или работы непосредственно со сжатыми данными (как описано в подразделе «Сжатие столбцов» раздела 3.3).

[2] Обратите внимание: кодирование (encoding) не имеет никакого отношения к шифрованию (encryption). Мы не обсуждаем последнее в этой книге.

надолго связываете себя с текущим языком программирования, препятствуя интеграции ваших систем с системами других компаний (которые используют другие языки программирования).

- Для возвращения данных в форму тех же объектных типов процесс декодирования должен иметь возможность создавать экземпляры произвольных классов. Это часто приводит к проблемам с безопасностью [5]: если злоумышленник способен заставить приложение декодировать произвольную последовательность байтов, то сможет создавать экземпляры произвольных классов, что, в свою очередь, позволит ему делать очень неприятные вещи, например удаленно выполнять произвольный код [6, 7].
- Контроль версий в этих библиотеках зачастую вопрос второстепенный: раз их основное предназначение — быстрое и удобное кодирование данных, «неудобные» проблемы прямой и обратной совместимости в них часто игнорируются.
- Эффективность (процессорное время, занимаемое кодированием и декодированием, а также размер получившейся структуры) тоже часто считается в них вопросом второстепенным. Например, встроенная сериализация языка Java печально известна своей низкой производительностью и занимающими большой объем памяти структурами [8].

Вследствие этих причин использование встроенных средств кодирования языка программирования для любых целей, кроме очень кратковременных, обычно будет плохой идеей.

JSON, XML и двоичные типы данных

Если говорить о стандартизованных кодированиях, которые могут читаться и записываться многими языками программирования, то очевидными претендентами являются JSON и XML. Они довольно известны, широко поддерживаются, и их практически столь же активно недолюбливают. XML часто критикуют за его «многословность» и излишнюю усложненность [9]. JSON обязан своей популярностью в основном встроенной поддержке в браузерах (благодаря тому что он является подмножеством Java) и относительной простоте по сравнению с XML. CSV — еще один популярный и независимый от языка программирования формат, хотя и обладающий меньшими возможностями.

JSON, XML и CSV — текстовые форматы, а значит, в какой-то мере удобочитаемые для людей (хотя их синтаксис — частая тема дискуссий). Помимо кажущихся проблем с синтаксисом, у них есть и более тонкие проблемы.

- Кодирование чисел приводит к множеству неоднозначностей. В XML и CSV невозможно различить число и строку, состоящую из цифр (разве что с по-

мощью ссылки на внешнюю схему). JSON различает числа и строки, но не целочисленные значения и значения с плавающей точкой и не позволяет задать их точность.

Существует также проблема относительно больших чисел. Например, целые числа больше 2^{53} невозможно точно представить в виде числа с плавающей точкой двойной точности в смысле стандарта IEEE 754, так что подобные числа становятся неточными при синтаксическом разборе в языках, использующих числа с плавающей точкой (например, JavaScript). Числа, превышающие 2^{53}, встречаются в Twitter, где применяются 64-битные числа для идентификации твитов. Возвращаемый API Twitter JSON включает идентификатор твита два раза, один в качестве JSON-числа и второй — десятичной строки, чтобы обойти проблему с некорректным разбором чисел приложениями на языке JavaScript [10].

- В JSON и XML хорошо поддерживаются строки символов в кодировке Unicode (то есть доступный для чтения человеком текст), но не двоичные строки (последовательности байтов без кодирования символов). Такие строки — полезная возможность, вследствие чего разработчики обходят это ограничение, кодируя двоичные данные как текст с помощью Base64. Далее используется схема, чтобы было понятно: значения должны интерпретироваться как кодировавшиеся с применением Base64. Это работает, но выглядит как «костыль» и увеличивает объем данных на 33 %.
- Как в XML [11], так и в JSON [12] есть дополнительная поддержка схемы. Эти языки схем обладают немалыми возможностями, а потому весьма сложны в изучении и использовании. XML-схемы применяются довольно широко, но многие основанные на JSON утилиты не слишком озабочиваются обращением к схемам. Поскольку правильная интерпретация данных (например, чисел и двоичных строк) зависит от содержащейся в схеме информации, не задействующим схемы XML/JSON приложениям приходится «зашивать» в код соответствующую логику кодирования/декодирования.
- У формата CSV нет никакой схемы, вследствие чего значение каждого столбца и строки приходится определять самому приложению. Если при изменении приложения добавляется новая строка или столбец, то приходится обрабатывать это вручную. CSV, кроме того, весьма расплывчатый формат (что произойдет, например, случись значению содержать запятую или символ новой строки?). Хотя его правила экранирования были формально специфицированы [13], далеко не все синтаксические анализаторы реализуют их правильно.

Несмотря на все эти изъяны, JSON, XML и CSV достаточно хороши для многих целей. Вполне вероятно, что они сохранят свою популярность, особенно в качестве форматов обмена данными (то есть для целей отправки данных от одного предприятия другому). В таких случаях главное — договориться, какой формат

используется; зачастую не имеет значения, насколько он красив или эффективен. Но сложность, состоящая в том, чтобы заставить различные предприятия договориться *хотя бы о чем-то*, часто перевешивает все остальные вопросы.

Двоичное кодирование. Для используемых только внутри организации данных нет такой острой необходимости применять формат кодирования, который бы стал «наименьшим общим знаменателем». Например, можно выбрать более сжатый формат или более быстрый при синтаксическом разборе. Для маленького набора данных выгоды пренебрежимо малы, но когда речь заходит о терабайтах, выбор формата данных имеет большое значение.

JSON — менее «многословный» формат, чем XML, но оба используют немало места, по сравнению с двоичными форматами. Это наблюдение привело к разработке большого количества двоичных кодирований для JSON (упомяну лишь несколько: MessagePack, BSON, BJSON, UBJSON, BISON и Smile) и для XML (например, WBXML и Fast Infoset). Эти форматы применяются во многих некрупных сегментах рынка, но ни один из них не задействован столь широко, как текстовые версии JSON и XML.

Некоторые из этих форматов расширяют множество типов данных (например, различая целые числа и числа с плавающей точкой или добавляя поддержку двоичных строк), но в остальном оставляют модель данных JSON/XML неизменной. В частности, поскольку они не задают какой-либо схемы, в закодированных данных должны содержаться все имена полей объекта. То есть в двоичной версии JSON-документа в примере 4.1 где-то должны содержаться строки `userName`, `favoriteNumber` и `interests`.

Пример 4.1. Пример записи, которую мы в этой главе будем кодировать в различные форматы

```
{
    "userName": "Martin",
    "favoriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```

Рассмотрим пример MessagePack — двоичного формата кодирования JSON. На рис. 4.1 показана последовательность байтов, получаемая в результате кодирования JSON-документа из примера 4.1 с помощью MessagePack [14]. Первые несколько байтов выглядят следующим образом.

1. Первый байт, `0x83`, указывает, что за ним следует объект (старшие четыре бита равны `0x80`) с тремя полями (младшие четыре бита равны `0x03`). Если вам интересно, что будет в случае объекта с более чем 15 полями (ведь такое количество полей не помещается в четыре бита), то для него предусмотрен другой индикатор типа, в котором количество полей кодируется двумя или четырьмя байтами.

2. Второй байт, `0xa8`, указывает, что за ним следует строка (старшие четыре бита равны `0xa0`) длиной восемь байт (младшие четыре бита равны `0x08`).
3. Следующие восемь байт — имя поля `userName` в кодировке ASCII. Поскольку его длина была задана ранее, нет необходимости в каком-либо маркере, указывающем позицию окончания строки (как нет нужды и в экранировании).
4. Следующие семь байт кодируют строковое значение из шести символов `Martin` с помощью префикса `0xa6` и т. д.

Длина закодированных таким двоичным образом данных — 66 байт, что лишь немногим более 81 байта, занимаемого при текстовом кодировании JSON (с удалением пробелов). Все виды двоичного кодирования JSON в этом смысле схожи. Трудно сказать, стоит ли подобное незначительное уменьшение занимаемого места (и, вероятно, ускорение синтаксического разбора) потери удобочитаемости для людей.

Ниже мы увидим, как можно добиться гораздо большего и закодировать ту же запись с помощью всего 32 байт.

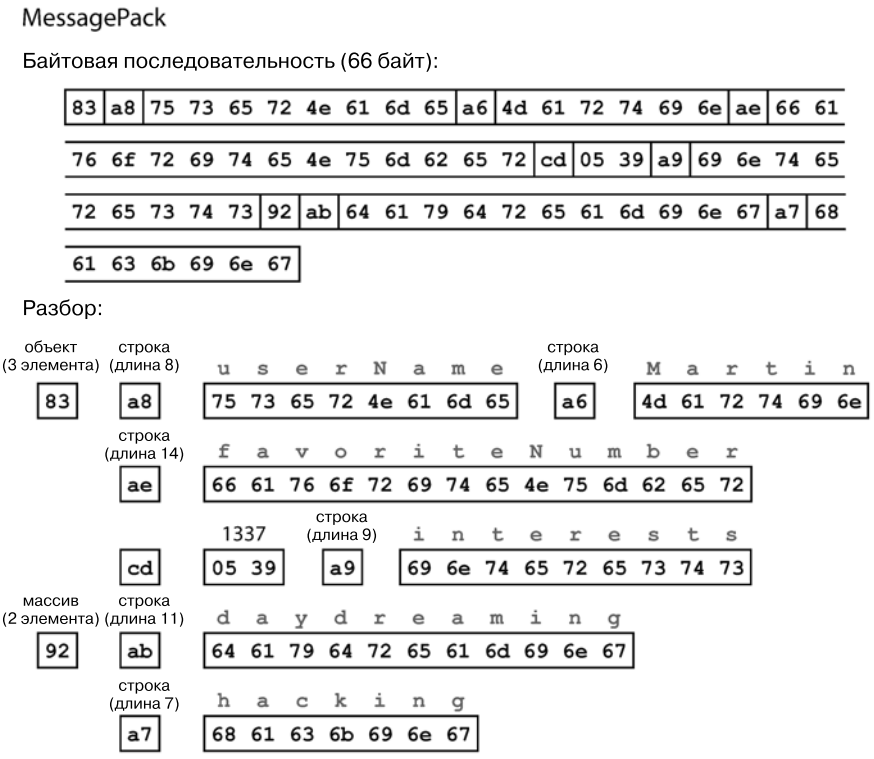


Рис. 4.1. Пример записи (см. пример 4.1) в кодировке MessagePack

Thrift и Protocol Buffers

Apache Thrift [15] и Protocol Buffers (protobuf) [16] — библиотеки двоичного кодирования, основанные на одном принципе. Protocol Buffers была разработана компанией Google, Thrift — компанией Facebook, и исходный код обеих был открыт в 2007–2008 годах [17].

Как Thrift, так и Protocol Buffers требуют наличия схемы для любых кодируемых данных. Чтобы кодировать данные из примера 4.1 в библиотеке Thrift, можно изложить схему на языке описания интерфейсов библиотеки Thrift (interface definition language, IDL) следующим образом:

```
struct Person {
  1: required string       userName,
  2: optional i64          favoriteNumber,
  3: optional list<string> interests
}
```

Эквивалентное описание схемы для библиотеки Protocol Buffers выглядит очень похоже:

```
message Person {
    required string user_name       = 1;
    optional int64  favorite_number = 2;
    repeated string interests       = 3;
}
```

Как Thrift, так и Protocol Buffers включают утилиту генерации кода, получающую на входе описание схемы, подобное вышеприведенным, и генерирует классы, реализующие указанную схему на различных языках программирования [18]. А код приложения затем может вызывать этот сгенерированный код для кодирования/декодирования записей схемы.

Как выглядят данные, закодированные на основе этой схемы? В библиотеке Thrift имеется два разных двоичных формата кодирования[1], что часто вызывает путаницу. Они называются BinaryProtocol и CompactProtocol соответственно. Рассмотрим сначала BinaryProtocol. Пример 4.1 в этом формате занимает 59 байт, как показано на рис. 4.2 [19].

Аналогично рис. 4.1, здесь у каждого поля есть сигнатура типа (указывающая, является ли он строкой, целым числом, списком и т. п.) и при необходимости указание длины (длина строки, количество элементов в списке). Встречающиеся в данных

[1] На самом деле их даже три: BinaryProtocol, CompactProtocol и DenseProtocol, хотя DenseProtocol поддерживается только реализацией на языке C++, так что его нельзя считать межъязыковым [18]. Помимо них, в Thrift есть также два различных формата кодирования на основе JSON [19]. Вот здорово!

строки (`"Martin"`, `"daydreaming"`, `"hacking"`) также преобразуются в кодировку ASCII (или, точнее, UTF-8), как и ранее.

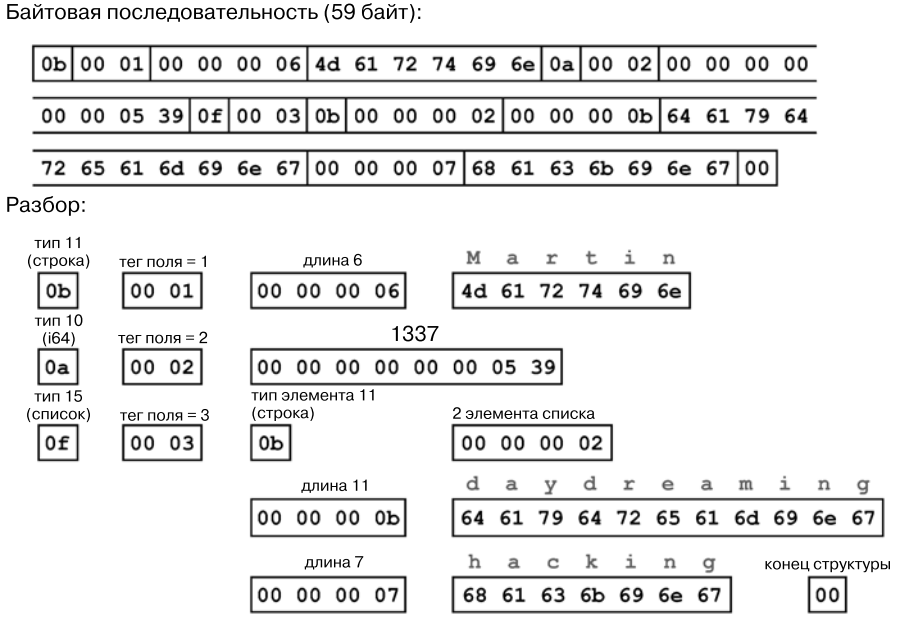


Рис. 4.2. Наш пример записи, закодированный с помощью BinaryProtocol библиотеки Thrift

Важное отличие от рис. 4.1 — отсутствие названий полей (`userName`, `favoriteNumber`, `interests`). Вместо них кодированные данные содержат *теги полей* (field tags), представляющие собой числа (`1`, `2` и `3`). Эти числа указываются в описании схемы. Теги полей — нечто вроде псевдонимов для полей: сжатый способ указать на поле, о котором мы говорим, не используя его название.

Формат кодирования CompactProtocol библиотеки Thrift семантически эквивалентен BinaryProtocol, но, как можно видеть на рис. 4.3, упаковывает ту же информацию всего лишь в 34 байта. Это достигается благодаря упаковке типа поля и номера тега в один байт, а также с помощью целых чисел переменной длины. Вместо того чтобы использовать для числа 1337 полных восемь байтов, оно кодируется двумя байтами, причем старший бит каждого из этих байт указывает, есть ли еще байты. Это значит, что числа от –64 до 63 кодируются одним байтом, от –8192 до 8191 — двумя и т. д. Большие числа требуют большего числа байтов.

Наконец, библиотека Protocol Buffers (в которой есть только один двоичный формат кодирования) кодирует те же данные так, как показано на рис. 4.4. Упаковка битов происходит немного иначе, но в остальном все очень похоже на CompactProtocol библиотеки Thrift. Protocol Buffers умещает ту же запись в 33 байта.

Байтовая последовательность (34 байта):

```
18 06 4d 61 72 74 69 6e 16 f2 14 19 28 0b 64 61 79 64 72 65
61 6d 69 6e 67 07 68 61 63 6b 69 6e 67 00
```

Разбор:

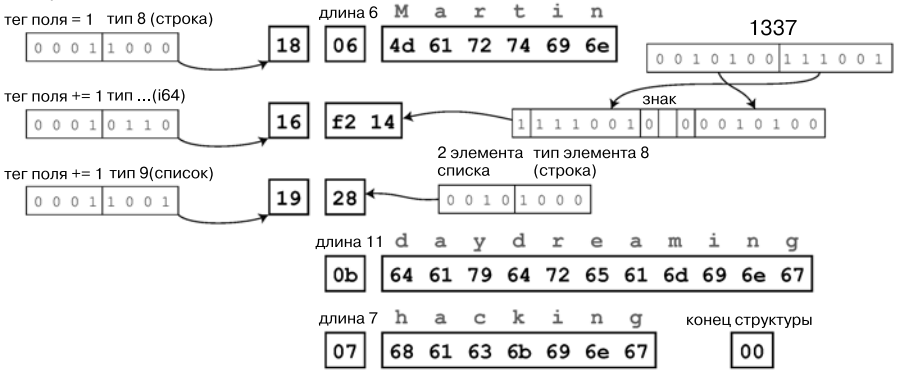


Рис. 4.3. Наш пример записи, закодированный с помощью CompactProtocol библиотеки Thrift

Байтовая последовательность (33 байта):

```
0a 06 4d 61 72 74 69 6e 10 b9 0a 1a 0b 64 61 79 64 72 65 61
6d 69 6e 67 1a 07 68 61 63 6b 69 6e 67
```

Разбор:

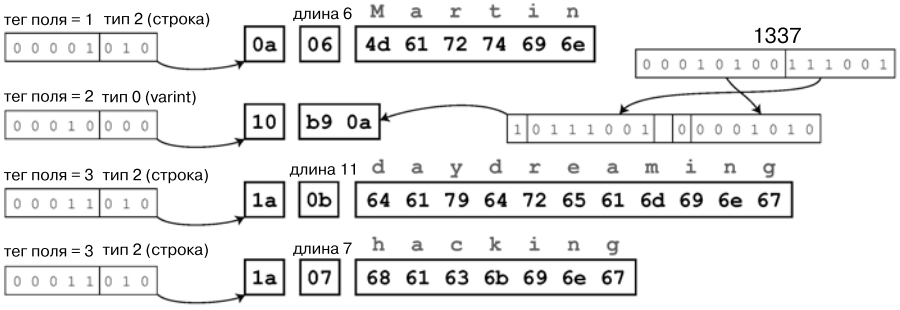


Рис. 4.4. Наш пример записи, закодированный с помощью Protocol Buffers

Отмечу один нюанс: в показанных выше схемах каждое из полей было помечено или как `required` (обязательное), или как `optional` (необязательное), но для кодирования полей это никакого значения не имеет (в двоичных данных ничто

не указывает, было ли поле обязательным). Разница проста: пометка `required` активизирует проверку во время выполнения, которая завершается неудачно, если значение поля не задано; это может быть полезно для поиска ошибок.

Теги полей и эволюция схемы

Я уже упоминал, что схемы неизбежно меняются с течением времени. Это называется *эволюцией схемы*. Как же библиотеки Thrift и Protocol Buffers справляются с изменениями схемы, не теряя при этом прямой и обратной совместимости?

Как видно из примеров, закодированная запись представляет собой просто конкатенацию закодированных полей. Каждое поле определяется своим номером тега (номера `1`, `2`, `3` в примерах схем) и помечено типом данных (например, строка или целое число). Если значение поля не задано, то оно просто не включается в закодированную запись. Вследствие этого очевидно, что теги полей очень важны для осмысленности закодированных данных. Можно поменять название поля в схеме, ведь закодированные данные никогда не ссылаются на названия полей, но нельзя поменять тег поля, так как из-за этого все существующие закодированные данные превратятся в неправильные.

В схему можно добавлять новые поля, конечно, если не забывать задавать новый номер тега для каждого из них. Старый код (которому ничего не известно о добавленных новых номерах тегов), пытающийся прочитать записанные новым кодом данные, включая новое поле с непонятным ему номером тега, может просто этот номер игнорировать. Благодаря сигнатуре типа данных синтаксический анализатор способен определить, сколько байтов ему нужно пропустить. Этим обеспечивается прямая совместимость: старый код может читать записи нового кода.

Что же насчет обратной совместимости? При наличии у всех полей уникальных номеров тегов новый код всегда сможет прочитать старые данные, ведь смысл номеров тегов не поменялся. Единственный нюанс: при добавлении нового поля нельзя сделать его обязательным. Если пришлось добавить поле и сделать его обязательным, то проверка окажется неудачной в случае чтения новым кодом старых данных, ведь старый код не запишет это добавленное новое поле. Следовательно, для поддержания обратной совместимости каждое добавленное после начального развертывания схемы поле должно быть необязательным или содержать значение по умолчанию.

Удаление поля ничем не отличается от его добавления, разве что обратная и прямая совместимости меняются местами. То есть можно удалять только необязательные поля (обязательное поле нельзя удалять) и нельзя использовать один и тот же номер тега повторно (поскольку где-то могут еще храниться данные, содержащие старый номер тега, а это поле новый код должен игнорировать).

Типы данных и эволюция схемы

А как насчет изменения типа данных поля? Это возможно (см. подробности в документации), но существует риск потери точности полей или их усечения. Например, мы меняем 32-битное целое число на такое же 64-битное. Новый код с легкостью может читать данные, записанные старым, поскольку синтаксический анализатор может заполнить все недостающие биты нулями. Однако при чтении старым кодом данных, записанных новым, в старом все еще используется 32-битная переменная для хранения значения. Если декодированное 64-битное значение не помещается в 32 бита, то оно будет усечено.

Любопытная особенность формата Protocol Buffers: в нем нет типов данных для списка или массива, но вместо этого есть маркер `repeated` для полей (еще одна опция наряду с `required` и `optional`). Как вы можете видеть на рис. 4.4, кодирование поля с маркером `repeated` полностью соответствует названию: те же самые теги полей просто повторяются в записи несколько раз. Одно из удобств этого состоит в том, что можно поменять поле `optional` (с одним значением) на `repeated` (с несколькими значениями). Читающий старые данные новый код увидит список из нуля или одного элемента (в зависимости от того, присутствует ли поле), читающий новые данные старый код увидит только последний элемент.

В библиотеке Thrift есть специализированный тип данных для списка, параметризуемый типом данных элементов списка. При этом невозможна такая эволюция из поля с одним значением в поле с несколькими, как в Protocol Buffers, но есть преимущество в виде поддержки вложенных списков.

Avro

Apache Avro [20] — еще один двоичный формат, немного отличающийся от Protocol Buffers или Thrift. Начало ему было положено в 2009 году, в виде субпроекта Hadoop, поскольку Thrift плохо подходил для сценариев использования Hadoop [21].

Avro тоже применяет схему для задания структуры кодируемых данных. В нем есть два языка описания схем: один (Avro IDL) предназначен для редактирования людьми, а второй (основанный на формате JSON) лучше подходит для считывания компьютерами.

Наш пример схемы, записанный на Avro IDL, будет выглядеть следующим образом:

```
record Person {
    string               userName;
    union { null, long } favoriteNumber = null;
    array<string>        interests;
}
```

Соответствующее JSON-представление этой схемы:

```
{
    "type": "record",
    "name": "Person",
    "fields": [
        {"name": "userName",       "type": "string"},
        {"name": "favoriteNumber", "type": ["null", "long"], "default": null},
        {"name": "interests",      "type": {"type": "array", "items": "string"}}
    ]
}
```

Прежде всего обратите внимание: в схеме нет номеров тегов. При кодировании нашего примера записи (пример 4.1) с помощью этой схемы длина данных, полученных путем двоичного кодирования Avro, составит 32 байта — самое компактное кодирование из всех, рассмотренных нами до сих пор. Разбор закодированной последовательности байтов показан на рис. 4.5.

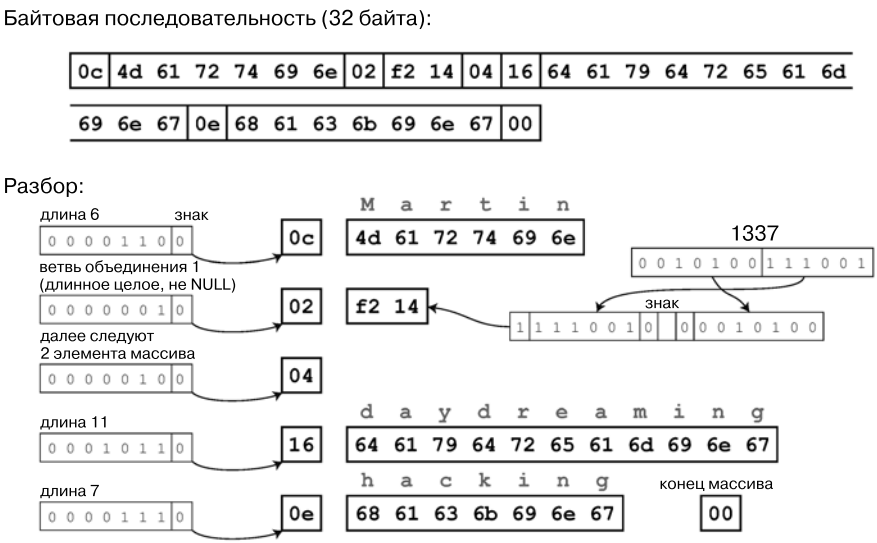


Рис. 4.5. Наш пример записи, закодированный с помощью Avro

При внимательном взгляде на последовательность байтов видно: в ней нет ничего, что бы идентифицировало поля или их типы данных. Закодированные данные состоят просто из сцепленных воедино значений, строка — просто из префикса с длиной, за которым следуют байты в кодировке UTF-8, но ничто в закодированных данных не указывает, что это строка. На ее месте могло бы быть целое число или вообще нечто совершенно иное. Целые числа кодируются с помощью кодирования со значениями переменной длины (подобно CompactProtocol библиотеки Thrift).

Для синтаксического разбора двоичных данных необходимо пройти по полям в том порядке, в котором они встречаются в схеме, и получить из схемы тип данных каждого из полей. Следовательно, правильно декодировать двоичные данные можно только тогда, когда читающий данные код использует *ту же самую* схему, что и записавший их код. Любые различия схем между записывающим и читающим кодом будет означать неправильное декодирование данных.

Итак, как же Avro обеспечивает эволюцию схемы?

Схема для чтения и схема для записи

Если работающему с Avro приложению необходимо кодировать какие-либо данные (чтобы записать их в файл или БД, отправить по сети и т. п.), то оно декодирует их на основе той схемы, которая ему доступна, — например, такая схема может быть скомпилирована в качестве части приложения. Это называется *схемой для записи* (writer's schema).

Если приложению необходимо декодировать какие-либо данные (прочитать их из файла или БД, получить их по сети и т. п.), оно ожидает, что те будут соответствовать определенной схеме, известной под названием *схемы для чтения* (reader's schema). Именно она лежит в основе кода приложения: тот, например, мог быть сгенерирован на основе этой схемы во время процесса компоновки приложения.

Основная идея Avro: схема для чтения и схема для записи *не обязательно должны совпадать* — достаточно, чтобы они были совместимы. При декодировании (чтении) данных библиотека разрешает конфликты, сопоставляя схему для чтения и схему для записи и преобразовывая данные из одной в другую. Спецификации Avro [20] четко определяют метод работы этого разрешения конфликтов (показано на рис. 4.6).

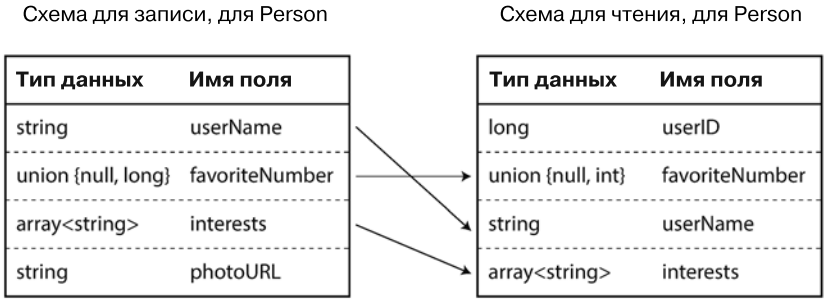


Рис. 4.6. Модуль чтения Avro разрешает конфликты между схемой для записи и схемой для чтения

Например, различный порядок в этих двух схемах не является проблемой, ведь при разрешении схемы поля сопоставляются по имени поля. Если читающий данные код натыкается на поле, которое есть в схеме для записи, но отсутствует в схеме для чтения, то игнорирует его. Если читающий данные код ожидает некое поле, но поле с таким именем отсутствует в схеме для записи, то оно заполняется указанным в схеме для чтения значением по умолчанию.

Правила эволюции схемы

При работе с Avro прямая совместимость означает возможность сосуществования новой версии схемы для записи и старой для чтения. С другой стороны, обратная совместимость означает, что может быть новая версия схемы для чтения и старая — схемы для записи.

Для поддержки совместимости иногда достаточно лишь добавить или удалить поле со значением по умолчанию. (Таковым для поля `favoriteNumber` нашей схемы является `null`.) Например, мы добавили поле со значением по умолчанию, так что это новое поле есть в новой схеме, но отсутствует в старой. В момент чтения с помощью новой схемы записи, которая была записана с применением старой, отсутствующее поле заполняется значением по умолчанию.

При необходимости добавить поле, у которого нет значения по умолчанию, новый код не сможет читать записанное ранее, что приведет к нарушению обратной совместимости. Если вам пришлось удалить поле без значения по умолчанию, то старый код чтения не сможет читать данные, записанные новым; это повлечет нарушение прямой совместимости.

В некоторых языках программирования `null` — допустимое значение по умолчанию для любой переменной, но не в Avro: чтобы поле могло быть `null`, необходимо воспользоваться *типом объединения* (union). Например, выражение `union { null, long, string } field;` означает: `field` может быть числом, или строкой, или иметь значение `null`. В качестве значения по умолчанию `null` применяется только в том случае, если является одной из ветвей объединения[1]. Такой способ записи занимает несколько больше места, чем в случае, когда все способно принимать неопределенное значение по умолчанию, однако явное указание на то, что может быть `null`, а что — нет, позволяет предотвращать ошибки [22].

Соответственно, в Avro нет таких маркеров `optional` и `required`, как в Protocol Buffers и Thrift (вместо этого здесь есть тип объединения и значения по умолчанию).

---

[1] Точнее говоря, тип значения по умолчанию должен соответствовать первой ветви объединения, хотя это ограничение именно Avro, а не общее свойство типов объединений.

Изменение типа данных поля вполне вероятно, если Avro может преобразовать этот тип. Изменить название поля тоже допустимо, но не так просто: схема для чтения может включать псевдонимы для названий полей ради возможности сравнения старых полей из схемы для записи с псевдонимами. Это значит, что существует обратная совместимость изменения названия поля, но не прямая. Аналогично существует обратная совместимость добавления ветви в тип объединения, но не прямая.

Но что же такое схема для записи?

До сих пор мы обходили стороной важный вопрос: откуда читающий данные код знает, с помощью какой схемы для записи кодировался каждый конкретный элемент данных? Нельзя же включать в каждую запись всю схему, поскольку та, вполне возможно, намного превышает по размеру закодированные данные, что сводит на нет всю экономию от двоичного кодирования.

Ответ зависит от контекста использования Avro. Рассмотрим несколько примеров.

- *Большой файл с множеством записей.* Avro часто используется — особенно в контексте Hadoop — для хранения больших файлов, содержащих миллионы записей, кодированных на основе одной и той же схемы (мы обсудим такие ситуации в главе 10). В таком случае записывающий данный файл код вполне может вставить один раз схему для записи в начало файла. Для этой цели Avro специфицирует формат файлов (объектных файлов-контейнеров).
- *База данных с отдельными записями.* В БД различные записи могут быть записаны в разные моменты времени с помощью различных схем для записи — нельзя предполагать, что схема у всех записей будет одна и та же. Простейшее решение — добавить номер версии в начале каждой закодированной записи и хранить в БД список версий схемы. Читающий код может получить запись, извлечь из нее номер версии, после чего получить из базы соответствующую этому номеру версии схему для записи (подобным образом работает, например, Espresso [23]).
- *Отправка записей по сети.* Когда два процесса обмениваются сообщениями через двунаправленное сетевое соединение, они могут согласовать версию схемы при установке соединения и использовать ее затем на протяжении всего времени жизни этого соединения. Подобным образом работает протокол Avro RPC (см. подраздел «Поток данных через сервисы: REST и RPC» раздела 4.2).

База данных версий схем пригодится в любом случае, ведь она может служить в качестве документации и позволяет проверять совместимость схем [24]. В качестве номера версии подойдет или просто увеличивающееся целое число, или хеш схемы.

Динамически генерируемые схемы

Одно из преимуществ подхода Avro, по сравнению с Protocol Buffers и Thrift, — схема не содержит никаких номеров тегов. Но почему это важно? Что плохого в хранении в схеме пары чисел?

Различие в том, что Avro больше благоприятствует *динамически генерируемым* схемам. Например, у вас имеется реляционная база данных и необходимо выполнить дамп ее содержимого в файл, причем хотелось бы задействовать двоичный формат во избежание вышеупомянутых проблем с текстовыми форматами (JSON, CSV, SQL). Используя Avro, можно довольно легко из реляционной схемы сгенерировать схему Avro (в JSON-представлении, которое вы видели выше) и кодировать содержимое БД с помощью этой схемы, сбросив его полностью в объектный файл-контейнер [25]. Для каждой таблицы БД генерируется схема записей, а каждый столбец становится полем в этих записях. Названию столбца в базе ставится в соответствие название поля в Avro.

Теперь, если схема базы данных поменяется (например, в таблице один столбец будет добавлен, а один — удален), то можно будет просто сгенерировать новую схему Avro на основе обновленной схемы БД и экспортировать данные в этой новой схеме. На процесс экспорта данных не должны влиять изменения схемы — достаточно просто выполнять при каждом его запуске преобразование схемы. Все, кто будет читать новые файлы данных, увидят, что поля записей изменились, но поскольку они идентифицируются по именам, обновленную схему для записи можно будет все равно сопоставить со старой для чтения.

Напротив, при использовании для этих целей Protocol Buffers или Thrift теги полей придется, вероятно, присваивать вручную: при каждом изменении схемы базы данных администратору придется вручную обновлять соответствие названий столбцов БД тегам полей (процесс можно автоматизировать, но генератор схем должен будет работать очень аккуратно, чтобы не присвоить уже применяемые теги полей). Подобные динамически генерируемые схемы попросту не рассматривались в качестве цели разработки Thrift или Protocol Buffers, в отличие от Avro.

Генерация кода и языки программирования с динамической типизацией

Форматы Thrift и Protocol Buffers основываются на генерации кода: после описания схемы можно сгенерировать код, реализующий ее на выбранном языке программирования. Это удобно для языков программирования со статической типизацией, таких как Java, C++ или C#, поскольку позволяет использовать для декодированных данных эффективные структуры данных в оперативной памяти, а также

проверять типы и автодополнения в IDE при написании программ, обращающихся к структурам данных.

В языках программирования с динамической типизацией, таких как JavaScript, Ruby или Python, смысла в генерации кода практически нет, поскольку типы во время компиляции не проверяются. К генерации кода в этих языках чаще всего относятся неодобрительно, поскольку стараются избежать явного шага компиляции. Более того, в случае динамически сгенерированной схемы (наподобие сгенерированной из таблицы БД схемы Avro) генерация кода становится лишним препятствием на пути к данным.

Avro предоставляет возможность генерации (по желанию) кода для языков программирования со статической типизацией, но он столь же успешно применим и без всякой генерации. При наличии объектного файла-контейнера (включающего схему для записи) можно просто открыть его с помощью библиотеки Avro и посмотреть на данные, аналогично просмотру файла в формате JSON. Этот файл — *самоописываемый*, в том смысле, что включает все необходимые метаданные.

Это свойство особенно полезно в сочетании с динамически типизированными языками обработки данных, такими как Apache Pig [26]. В языке Pig можно просто открывать файлы Avro, начинать их анализировать и записывать в выходные файлы в формате Avro производные наборы данных, даже не задумываясь о схемах.

Достоинства схем

Как мы видели, Protocol Buffers, Thrift и Avro применяют схемы для описания форматов двоичного кодирования. Их языки описания схем куда проще, чем XML Schema или JSON Schema, поддерживающие намного более подробные правила проверки (например, «строковое значение этого поля должно соответствовать следующему регулярному выражению» или «целочисленное значение этого поля должно находиться между 0 и 100»). А раз Protocol Buffers, Thrift и Avro проще в реализации и использовании, они расширились до поддержки достаточно широкого спектра языков программирования.

Идеи, на которых основаны эти форматы кодирования, совсем не новы. Например, у них много общего с ASN.1 — языком описания схем, впервые стандартизированным в 1984-м [27]. Он использовался для описания различных сетевых протоколов, а его двоичное кодирование (DER) до сих пор применяется для кодирования SSL-сертификатов (X.509), например [28]. ASN.1 поддерживает эволюцию схемы с помощью номеров тегов, подобно Protocol Buffers и Thrift [29]. Однако он очень сложен и плохо документирован, так что для новых приложений это не лучший вариант.

Многие информационные системы также реализуют какое-либо проприетарное двоичное кодирование для своих данных. Например, у большинства реляционных БД есть сетевой протокол, по которому можно отправлять запросы к базе и получать на них ответы. Такой протокол обычно свой для каждой конкретной СУБД, и производители БД предоставляют драйверы (например, на основе API ODBC или JDBC), декодирующие получаемые по сетевому протоколу базы ответы в структуры, располагаемые в оперативной памяти.

Итак, хотя текстовые форматы данных, такие как JSON, XML и CSV, широко распространены, форматы двоичного кодирования, основанные на схемах, — тоже вполне приемлемое решение. У них есть несколько удобных свойств.

- Они могут быть намного компактнее различных вариантов «двоичного JSON», поскольку позволяют не включать названия полей в закодированные данные.
- Схема — важный вид документации, и поскольку она необходима для декодирования, вы всегда можете быть уверены в ее актуальности (в то время как поддерживаемая вручную документация легко может оказаться оторванной от действительности).
- База данных схем дает возможность проверять прямую и обратную совместимость изменений схемы до развертывания.
- Пользователям языков программирования со статической типизацией окажется полезна возможность генерировать код на основе схемы, позволяющая проверять типы во время компиляции.

Подведем итог: эволюция схемы предоставляет такую же гибкость, что и бессхемные (типа «схема при чтении») базы данных JSON (см. пункт «Гибкость схемы в документной модели» подраздела «Реляционные и документоориентированные базы данных сегодня» раздела 2.1), обеспечивая при этом лучший инструментарий и эффективнее защищая данные.

4.2. Режимы движения данных

В начале главы упоминалось, что при каждой отправке данных другому процессу, с которым у вас нет совместно используемой памяти (например, при отправке данных по сети или записи их в файл), необходимо кодировать их в байтовую последовательность. Далее мы обсудили множество различных методов кодирования для достижения этой цели.

Мы рассмотрели вопросы обратной и прямой совместимости, играющие важную роль при обеспечении возможности эволюции схемы (они упрощают изменения, обновляя различные части системы независимо, а не все вместе). Совместимость — связь между процессом, кодирующим данные, и другим процессом, декодирующим их.

Это довольно абстрактная идея, и существует множество путей движения данных из одного процесса в другой. Кто занимается кодированием данных, и кто — декодированием? В оставшейся части главы мы рассмотрим некоторые из наиболее распространенных способов движения данных между процессами:

- через БД (см. подраздел «Поток данных через БД» текущего раздела);
- с помощью вызовов сервисов (см. подраздел «Поток данных через сервисы: REST и RPC» текущего раздела);
- путем передачи асинхронных сообщений (см. подраздел «Поток данных передачи сообщений» текущего раздела).

Поток данных через БД

В БД записывающий в базу процесс кодирует данные, а читающий из нее — декодирует. Это может быть один процесс, обращающийся к базе, то есть читающий процесс представляет собой просто более позднюю версию того же процесса — при этом сохранение чего-либо в базе можно рассматривать как *отправку сообщения будущему самому себе*.

Обратная совместимость здесь совершенно необходима, в противном случае «будущий» вы не сможете декодировать то, что записали ранее.

В целом часто случается одновременное обращение к базе ряда разных процессов. Они могут быть несколькими различными приложениями или сервисами либо несколькими экземплярами одного сервиса (выполняемыми параллельно в целях лучшего масштабирования или отказоустойчивости). В любом случае в среде, в которой приложение изменяется, существует вероятность того, что код части обращающихся к базе данных процессов будет более новым, а части — более старым. Такое возможно, например, из-за того, что новая версия в этот момент развертывается с помощью плавающего обновления, вследствие чего часть экземпляров уже обновлена, а часть — нет.

То есть значение может быть записано *более новой* версией кода, а затем прочитано все еще работающей *более старой* версией. Следовательно, прямая совместимость для баз данных тоже необходима.

Однако есть еще одна загвоздка. Допустим, вы добавили в схему какой-либо записи поле, и новый код записал в базу данных значение этого поля. Позднее старая версия кода (не знающая о существовании нового поля) читает ту запись, изменяет ее и записывает обратно. В таком случае желательно, чтобы старый код не трогал новое поле, независимо от невозможности его интерпретировать.

Обсуждавшиеся ранее форматы кодирования поддерживают подобное предохранение неопознанных полей, но иногда приходится позаботиться об этом на уровне

приложения, как показано на рис. 4.7. Например, если декодировать значение из базы данных в объекты модели в приложении, а затем повторно их кодировать, то неопознанное поле может потеряться в процессе преобразования. Решить эту проблему несложно, нужно только не забывать о ее существовании.

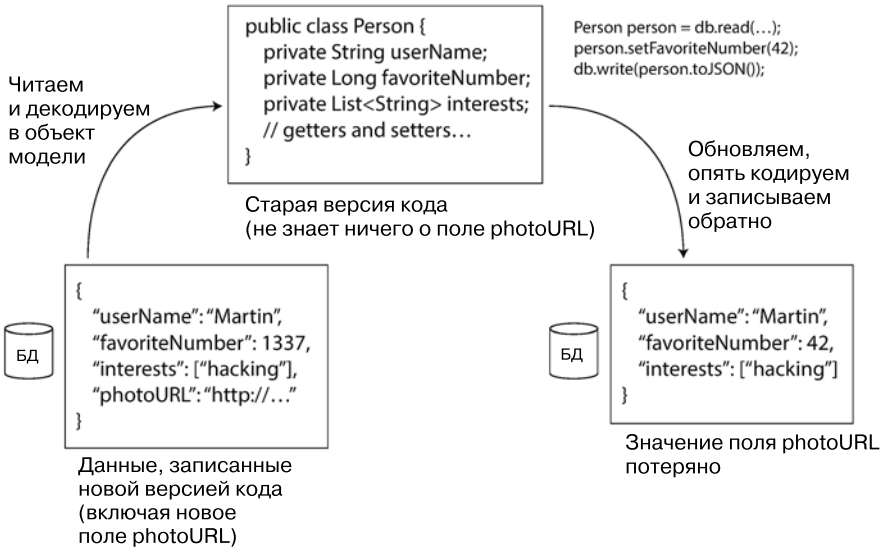


Рис. 4.7. Если не быть аккуратными, то можно потерять данные при обновлении старой версией приложения данных, ранее записанных более новой его версией

В разные моменты времени записываются различные значения

В целом база данных разрешает обновлять любое значение в любой момент времени. Это значит, что в одной базе могут быть значения, записанные пять миллисекунд назад и пять лет назад.

При развертывании новой версии приложения (серверного приложения по крайней мере) можно полностью заменить старую версию новой за несколько минут. Применительно к содержимому БД так сказать нельзя: пятилетние данные останутся на своем месте, в исходном формате кодирования, если вы с тех пор не перезаписали их явным образом. Это наблюдение иногда формулируется в виде *«время жизни данных превышает время жизни кода»*.

Перезапись (миграция) данных в новую схему, безусловно, возможна, но в случае большого набора данных это ресурсоемкая операция, так что большинство БД по мере сил избегают ее выполнения. В большинстве реляционных БД разрешены простые изменения схемы, такие как добавление нового столбца со значением `null`

по умолчанию, без перезаписи существующих данных[1]. При чтении старой строки база заполняет значениями `null` все столбцы, отсутствующие в закодированных данных с диска. Документоориентированная БД Espresso соцсети LinkedIn применяет для хранения формат Avro, благодаря чему может пользоваться правилами эволюции схемы Avro [23].

Эволюция схемы, таким образом, позволяет базе данных быть представленной в таком виде, как будто она вся закодирована с помощью единой схемы, несмотря на то что в хранилище могут присутствовать записи, закодированные на основе различных версий этой схемы.

Архивное хранение

Вероятно, время от времени вы делаете копии состояния своей БД, например, в качестве резервных копий или для дальнейшей загрузки в склад (см. подраздел «Складирование данных» раздела 3.2). В этом случае дампы данных обычно кодируются на основе наиболее новой схемы, даже если первоначальное кодирование в базе-источнике содержало смесь различных версий схем, относящихся к разным моментам времени. Поскольку вы все равно копируете данные, можно по крайней мере закодировать эту копию единообразно.

Так как дамп данных записывается за один подход, а значит, является неизменяемым, для него хорошо подходят такие форматы, как объектные файлы-контейнеры Avro. Кроме того, это хорошая возможность закодировать данные в удобный для анализа столбцовый формат, например, Parquet (см. раздел 3.3).

В главе 10 мы обсудим более подробно использование данных при архивном хранении.

Поток данных через сервисы: REST и RPC

Существует несколько различных способов организации взаимодействия процессов по сети. Наиболее часто используемый вариант включает два вида ролей: *клиенты* и *серверы*. Серверы предоставляют видимый по сети API, а клиенты могут к ним подключаться и выполнять запросы к этому API. Предоставляемый сервером API называется *сервисом*.

Интернет функционирует следующим образом: клиенты (браузеры) выполняют запросы к веб-серверам — запросы типа `GET` для скачивания HTML, CSS, JavaScript, изображений и т. п. и запросы типа `POST` для отправки данных на сервер. API со-

[1] За исключением СУБД MySQL, которая часто перезаписывает целые таблицы даже в отсутствие строгой необходимости этого, как упоминалось в пункте «Гибкость схемы в документной модели» подраздела «Реляционные и документоориентированные базы данных сегодня» раздела 2.1.

стоит из типового набора протоколов и форматов данных (HTTP, URLs, SSL/TLS, HTML и т. д.). А раз браузеры, веб-серверы и создатели сайтов пришли к общему соглашению относительно этих стандартов, для обращения к любому сайту можно использовать любой браузер (по крайней мере теоретически!).

Браузеры не единственный тип клиентов. Например, запущенное на мобильном устройстве или настольном компьютере нативное приложение тоже способно выполнять запросы к серверу, а работающее в браузере клиентское JavaScript-приложение может задействовать объект XMLHttpRequest, чтобы стать HTTP-клиентом (эта технология называется *Ajax* [30]). В подобном случае ответ сервера часто представляет собой не отображаемый человеку HTML, а данные в формате кодирования, удобном для дальнейшей обработки кодом клиентского приложения (например, JSON). Хотя в качестве транспортного протокола может использоваться HTTP, реализованный поверх него API зависит от конкретного приложения, так что детали этого API должны быть согласованы между клиентом и сервером.

Более того, сервер сам может служить клиентом для другого сервера (например, типичный веб-сервер приложений играет роль клиента по отношению к базе данных). Такой подход часто используется для декомпозиции больших приложений на небольшие сервисы по областям функциональности, чтобы сервисы выполняли запросы друг к другу в случае потребности в какой-либо функциональности или данных. Подобный метод создания приложений традиционно назывался *сервис-ориентированной архитектурой* (service-oriented architecture, SOA), а в последнее время подвергся пересмотру и получил новое название: *микросервисная архитектура* (microservices architecture) [31, 32].

В некоторых аспектах сервисы напоминают БД: они обычно предоставляют клиентам возможность отправлять и запрашивать данные. Однако в то время, как базы допускают произвольные запросы с помощью языков запросов, обсуждавшихся в главе 2, сервисы предоставляют API, свой для каждого приложения, допускающий только предопределенные бизнес-логикой (кодом приложения) сервиса входные и выходные данные [33]. Это ограничение обеспечивает некую степень инкапсуляции: сервисы получают возможность в деталях ограничивать то, что разрешено и не разрешено делать клиентам.

Главная цель сервис-ориентированной и микросервисной архитектур — облегчить изменение и поддержку приложения путем обеспечения независимого развертывания и развития сервисов. Например, каждый из сервисов должен относиться к сфере ответственности одной команды разработчиков, которая должна быть в состоянии часто выпускать новые его версии, не нуждаясь в согласовании действий с другими командами. Другими словами, нужно быть готовыми к тому, что старые и новые версии серверов и клиентов будут работать параллельно, а значит, используемые серверами и клиентами форматы кодирования данных должны быть совместимы между разными версиями API сервиса — именно то, о чем мы и говорили в этой главе.

Веб-сервисы

Если в качестве базового протокола для связи с сервисом используется HTTP, то такой сервис называется *веб-сервисом*. Вероятно, это не совсем правильное наименование, поскольку веб-сервисы применяются не только для работы в Интернете, но и в нескольких других контекстах. Рассмотрим примеры.

1. Запущенное на пользовательском устройстве клиентское приложение (например, нативное приложение на мобильном устройстве или JavaScript-приложение, задействующее Ajax) выполняет запросы к сервису по протоколу HTTP. Эти запросы обычно передаются через общедоступный Интернет.
2. Сервисы, относящиеся к одной организации, зачастую расположенные в одном ЦОДе, выполняют запросы друг к другу в рамках сервис-ориентированной/микросервисной архитектуры (поддерживающее подобные сценарии использования программное обеспечение иногда называется *промежуточным ПО*).
3. Сервисы, относящиеся к разным организациям, выполняют запросы друг к другу обычно через Интернет. Применяется для обмена данными между серверными системами разных организаций. Эта категория включает предоставляемые онлайн-сервисами общедоступные API, например системы обработки платежей по кредитным картам или OAuth для совместного доступа к пользовательским данным.

Существует два популярных подхода к проектированию веб-сервисов: *REST* и *SOAP*. Эти подходы практически противоположны и часто становятся объектом жарких дискуссий среди их сторонников[1].

REST не протокол, а скорее подход к проектированию, основанный на принципах HTTP [34, 35]. Он делает акцент на простых форматах данных, применении URL для идентификации ресурсов и использовании возможностей HTTP для управления кэшем, аутентификации и согласования типа контента. REST становится все более популярным по сравнению с SOAP, по крайней мере в контексте интеграции сервисов между организациями [36], и часто ассоциируется с микросервисами [31]. API, спроектированный в соответствии с принципами REST, называют *воплощающим REST* (RESTful).

Напротив, SOAP — основанный на формате XML протокол для выполнения запросов к сетевым API[2]. Будучи применяемым чаще всего по HTTP, он стремится к независимости от последнего и избегает использования большинства его возможностей. Вместо этого к нему прилагается масса разнообразных сопутствующих

---

[1] Даже внутри каждого из соответствующих лагерей есть множество споров. Например, часто вызывает дискуссии принцип HATEOAS (hypermedia as the engine of application state — «гипермедиа — движущая сила состояния приложения») [35].

[2] Несмотря на схожесть аббревиатур, SOAP отнюдь не одно из требований SOA. SOAP — отдельная технология, в то время как SOA — общий подход к созданию систем.

стандартов (*фреймворк веб-сервисов*, известный под названием *WS-**), которые добавляют в него различные возможности [37].

API SOAP веб-сервиса описывается с помощью основанного на XML языка, именуемого языком описания веб-сервисов (web services description language, WSDL). Последний позволяет генерировать код, так что клиент может обращаться к удаленным сервисам путем локальных классов и вызовов методов (кодируемых в XML-сообщениях и декодируемых снова фреймворком). Это подходит для языков программирования со статической типизацией, но менее удобно в языках с динамической (см. пункт «Генерация кода и языки программирования с динамической типизацией» подраздела «Avro» раздела 4.1).

Поскольку WSDL не предназначен для чтения людьми, а SOAP-сообщения часто слишком сложны для того, чтобы их можно было сформировать вручную, пользователи SOAP сильно зависят от поддержки утилит, генерации кода и различных IDE [38]. Для пользователей тех языков программирования, которые не поддерживаются производителями реализаций SOAP, интеграция с сервисами SOAP представляет непростую задачу.

Хотя SOAP и многочисленные его расширения вроде бы стандартизированы, взаимодействие между его реализациями от различных производителей часто приводит к проблемам [39]. В силу этих причин, хотя SOAP и используется во многих крупных организациях, в большинстве малых фирм он вышел из моды.

Воплощающие REST API имеют предрасположенность к более простым подходам, включающим обычно генерацию меньшего количества кода и использование автоматизированных утилит. Для описания воплощающих REST API и создания документации можно применить такой формат описания, как OpenAPI, известный также под названием Swagger [40].

Проблемы с удаленными вызовами процедур (RPC)

Веб-сервисы — просто новейшее воплощение длинной череды технологий выполнения запросов к API по сети, многие из которых наделали в свое время немало шума, но имели серьезные проблемы. Enterprise JavaBeans (EJB) и удаленные вызовы методов языка Java (remote method invocation, RMI) ограничены Java. Распределенная компонентная объектная модель (distributed component object model, DCOM) ограничивается платформами компании Microsoft. Общая архитектура брокера объектных запросов (common object request broker architecture, CORBA) излишне сложна и не обеспечивает прямой или обратной совместимости [41].

Все они основаны на понятии удаленного вызова процедуры (*remote procedure call*, RPC), появившемся еще в 1970-х годах [42]. Основная идея модели RPC состоит в том, что выполнение запроса к удаленному сетевому сервису должно выглядеть так же, как и вызов функции или метода на обычном языке программирования, в пределах одного процесса (эта абстракция называется *независимостью от расположения* (location transparency)). Хотя на первый взгляд RPC представляется

удобным, у этого подхода есть фундаментальные недостатки [43, 44]. Сетевой запрос сильно отличается от локального вызова функции.

- Локальному вызову функции присуща предсказуемость, он или завершается успешно, или нет, в зависимости только от контролируемых вами параметров. Сетевой запрос непредсказуем: запрос или ответ на него способен потеряться вследствие сетевого сбоя, удаленная машина может работать медленно или быть недоступной, и подобные проблемы — вне вашего контроля. Проблемы сети встречаются очень часто, их нужно стараться предугадывать, например, путем повтора отправки неудавшихся запросов.
- Локальный вызов функции или возвращает результат, или генерирует исключение, или вообще ничего не возвращает (поскольку, например, попадает в бесконечный цикл либо вследствие фатального сбоя процесса). У сетевого запроса есть еще один вероятный исход: может произойти возврат, но без результата вследствие *превышения времени ожидания*. В этом случае вы просто не будете знать, что произошло: если не получили ответ от удаленного сервиса, то никак не можете знать, был ли доставлен и выполнен запрос (мы обсудим этот вопрос подробнее в главе 8).
- Повторяя отправку неудавшегося сетевого запроса, вы должны учитывать возможность того, что ваши запросы на самом деле доставляются и выполняются, а теряются лишь ответы. В этом случае повтор отправки приведет к повторному выполнению действия, если только вы не встроите в протокол механизм дедупликации *(идемпотентности)*. У локальных вызовов функций этой проблемы нет (мы обсудим идемпотентность подробнее в главе 11).
- Каждый вызов локальной функции занимает примерно одно и то же время. Сетевые запросы выполняются намного медленнее вызовов функций, причем задержка варьируется гораздо сильнее: в удачное время он может выполняться менее чем за миллисекунду, но если сеть или удаленный сервис перегружены, то выполнение того же самого запроса может занять многие секунды.
- При вызове локальной функции ей можно успешно передать ссылки (указатели) на объекты в локальной памяти. При выполнении сетевого запроса все эти параметры приходится кодировать в байтовые последовательности, которые можно отправить по сети. В случае параметров, относящихся к простым типам данных (например, строк или чисел), никаких проблем нет, но для бо́льших объектов все резко усложняется.
- Клиент и сервис могут быть реализованы на разных языках программирования, так что фреймворку RPC придется преобразовывать типы данных одного языка программирования в типы данных другого. Это может закончиться плохо, ведь типы далеко не всех языков одинаковы — вспомним хотя бы проблемы JavaScript с числами, превышающими 2^{53} (см. подраздел «JSON, XML и двоичные типы данных» раздела 4.1). В случае одного процесса, написанного на одном языке программирования, эта проблема отсутствует.

Все приведенные факторы означают одно: нет смысла пытаться сделать удаленный сервис похожим на локальный объект языка программирования, поскольку это нечто принципиально иное. Подход REST, в частности, тем и привлекателен, что не пытается скрывать сетевую природу протокола (хотя это не удерживает некоторых от создания RPC-библиотек на основе REST).

Текущие тенденции RPC

Несмотря на все эти проблемы, RPC отнюдь не уходит со сцены. На перечисленных в данной главе форматах кодирования были основаны различные фреймворки RPC: например, в Thrift и Avro есть встроенная поддержка RPC, gRPC — реализация RPC, использующая Protocol Buffers, Finagle тоже применяет Thrift, а Rest.li задействует JSON через протокол HTTP.

Новое поколение фреймворков RPC более явным образом декларирует различие между удаленным запросом и локальным вызовом функции. Например, Finagle и Rest.li используют *фьючеры* (futures)/*промисы* (promises), инкапсулирующие потенциально сбойные асинхронные действия. Фьючеры также упрощают ситуации, при которых приходится выполнять параллельно запросы к нескольким сервисам с последующим объединением их результатов [45]. gRPC поддерживает *потоки данных* (streams), где вызов состоит не просто из одного запроса и одного ответа, а из последовательности производимых в течение некоторого времени запросов и ответов [46].

Часть представленных фреймворков также включают *обнаружение сервисов* (service discovery) — то есть позволяют клиентам искать, на каких IP-адресах и портах работает конкретный сервис. Мы вернемся к этому вопросу в разделе 6.5.

Пользовательские протоколы RPC с двоичным форматом кодирования иногда могут оказаться более производительными, чем универсальные, такие как JSON поверх REST. Однако у воплощающих REST API есть другие важные преимущества: они хорошо подходят для экспериментирования и отладки (к ним можно выполнять запросы просто через браузер или утилиту командной строки `curl`, без какой-либо генерации кода или установки программного обеспечения), их поддерживают все основные языки программирования и платформы. Кроме того, для них существует обширная экосистема вспомогательных утилит (серверов, кэшей, балансировщиков нагрузки, прокси, брандмауэров, утилит мониторинга, отладки, утилит для тестирования и др.).

Благодаря вышеперечисленному REST оказывается господствующим стилем общедоступных API. RPC-фреймворки сосредоточены в основном на запросах, выполняемых между сервисами, принадлежащими одной организации, обычно в пределах одного ЦОДа.

Кодирование и эволюция данных в случае RPC

Для возможности развития важно, что RPC-клиенты и серверы могут модифицироваться и развертываться независимо друг от друга. По сравнению с данными, проходящими через БД (как описывалось в предыдущем разделе), относительно потока данных через сервисы можно принять упрощающее допущение: разумно допустить, что сначала будут обновляться серверы, а потом клиенты. Следовательно, необходима только обратная совместимость по запросам и прямая совместимость по ответам.

Свойства обратной и прямой совместимости RPC-схема наследует от используемого ею формата кодирования.

- Эволюция RPC для Thrift, gRPC (Protocol Buffers) и Avro возможна по правилам совместимости соответствующего формата кодирования.
- В протоколе SOAP запросы и ответы описываются с помощью XML-схем. Эволюция этих схем возможна, но есть некоторые нюансы [47].
- Воплощающие REST API для ответов чаще всего используют формат JSON (без формального описания схемы), а для запросов — либо JSON, либо кодированные URI или формой параметры запроса. Для поддержки совместимости обычно рассматривают такие меры, как добавление необязательных параметров запроса и добавление новых полей в объекты ответов.

Обеспечение совместимости сервисов усложняется тем, что RPC часто применяется для взаимодействия между предприятиями, поэтому поставщики сервисов часто не могут никак повлиять на клиентов и заставить их обновиться. Следовательно, необходимо поддерживать совместимость в течение длительного времени, вероятно, до бесконечности. Если оказываются нужны нарушающие совместимость изменения, то поставщику сервиса часто приходится в конце концов поддерживать одновременно множество версий API сервиса.

Не существует каких-либо соглашений о том, как должен функционировать контроль версий API (то есть как клиент должен указывать нужную ему версию API [48]). В случае воплощающих REST API чаще всего указывается номер версии в URL или в HTTP-заголовке `Accept`. Сервисы, использующие для идентификации клиентов API-ключи, могут хранить запрашиваемую клиентом версию API на сервере и позволять обновление этой версии с помощью отдельного административного интерфейса [49].

Поток данных передачи сообщений

Мы рассмотрели различные способы кодирования потоков данных между процессами. До сих пор мы обсуждали REST и RPC (один процесс отправляет запрос к другому процессу по сети и ожидает от него ответ как можно скорее) и БД

(в которых один процесс записывает закодированные данные, а другой читает их когда-нибудь потом).

В этом последнем подразделе мы вкратце рассмотрим *системы асинхронной передачи сообщений* (asynchronous message-passing system) — нечто среднее между RPC и базами данных. Они схожи с RPC в том, что запрос от клиента (обычно называемый *сообщением* (message)) доставляется другому процессу без особой задержки. Они напоминают БД тем, что сообщение не отправляется непосредственно через сетевое соединение, а проходит через посредника, именуемого *брокером сообщений* (message broker), или *очередью сообщений* (message queue), или *промежуточным ПО, ориентированным на обработку сообщений* (message-oriented middleware), который временно хранит сообщение.

Использование брокера сообщений имеет несколько преимуществ по сравнению с RPC:

- он служит в качестве буфера в случае недоступности или перегруженности получателя, а следовательно, повышает надежность системы;
- он может автоматически отправлять сообщения повторно сбойным процессам, тем самым предотвращая потерю сообщений;
- благодаря ему отправителю не требуется знать IP-адрес и номера порта получателя (что особенно удобно в облачной среде, где виртуальные машины часто сменяют друг друга);
- он обеспечивает возможность отправки одного сообщения нескольким получателям;
- он логически расцепляет отправителя с получателем (отправитель лишь публикует сообщения, его не волнует, кто их получит).

Однако, в отличие от RPC, описанное взаимодействие — чаще всего одностороннее: отправитель обычно не ждет ответа на свои сообщения. Процессы могут отправлять ответы, но обычно задействуют отдельные каналы. Это *асинхронный* паттерн обмена сообщениями: отправитель не ждет доставки сообщения, а просто отправляет его и забывает о нем.

Брокеры сообщений

Ранее на рынке брокеров сообщений преобладало коммерческое корпоративное ПО от компаний TIBCO, IBM (WebSphere) и webMethods. Позднее начали приобретать популярность такие реализации с открытым исходным кодом, как RabbitMQ, ActiveMQ, HornetQ, NATS и Apache Kafka. Мы сравним их более подробно в главе 11.

Нюансы модели доставки зависят от реализации и конфигурации, но в целом брокеры сообщений используются следующим образом: один процесс отправляет сообщение, предназначенное для определенной *очереди* (queue) или *дискуссии*

(topic), а брокер обеспечивает доставку данного сообщения одному или нескольким *потребителям* (consumers) или *подписчикам* (subscribers) этой очереди или дискуссии. В одной дискуссии может быть много инициаторов (producers) и много потребителей сообщений.

Дискуссия обеспечивает лишь односторонний поток данных. Однако потребитель способен сам опубликовать сообщения в другой дискуссии (так что можно будет их связать в цепочку, как мы увидим в главе 11) или в очередь ответов, потребляемых отправителем первоначального сообщения (это позволяет формировать поток запросов/ответов аналогично RPC).

Брокеры сообщений обычно не навязывают какой-либо конкретной модели данных — сообщение представляет собой просто байтовую последовательность с метаданными, так что можно использовать любой формат кодирования. Если ваш формат кодирования как прямо, так и обратно совместим, то у вас появляется возможность совершенно свободно менять издателей и потребителей сообщений независимо друг от друга и развертывать их в любом порядке.

Если потребитель публикует сообщение снова в другой дискуссии, то следует соблюдать осторожность и сохранять неизвестные поля, чтобы избежать описанных выше в контексте баз данных проблем (рис. 4.7).

Распределенные акторные фреймворки

Акторная модель (actor model) — модель программирования для создания конкурентного доступа в пределах одного процесса. Вместо того чтобы иметь дело непосредственно с потоками выполнения (и сопутствующими проблемами состояния гонки, блокировок и взаимных блокировок), логика инкапсулируется в *акторах*. Каждый актор обычно соответствует одному клиенту или сущности, у него может быть свое локальное состояние (не разделяемое ни с какими другими акторами), и он взаимодействует с другими акторами путем отправки и получения асинхронных сообщений. Доставка сообщений не гарантируется: при определенных сценариях ошибок сообщения могут теряться. А поскольку каждый актор обрабатывает одновременно только одно сообщение, нет необходимости заботиться о потоках выполнения, а выполнение каждого актора может планироваться фреймворком независимо от других.

В *распределенных акторных фреймворках* эта модель программирования используется для масштабирования приложения на несколько узлов. При этом применяется тот же самый механизм передачи сообщений, вне зависимости от того, на одном узле расположены отправитель и получатель или на разных. Если они расположены на разных узлах, то сообщение явным образом кодируется в байтовую последовательность, пересылается по сети и декодируется на другой стороне.

Независимость от расположения лучше развита в акторной модели, чем RPC, поскольку эта модель изначально допускает возможность потери сообщений даже

в пределах одного процесса. Хотя задержка передачи сообщений по сети, вероятно, выше, чем в пределах одного процесса, при использовании акторной модели различия между локальным и удаленным обменом сообщениями не столь велики.

Распределенные акторные фреймворки, по сути, объединяют в одном фреймворке брокер сообщений и акторную модель программирования. Однако при необходимости выполнять плавающие обновления основанного на акторах приложения все равно приходится заботиться о прямой и обратной совместимости, так как сообщение может быть отправлено с узла, на котором запущена новая версия, на узел, где запущена старая, и наоборот.

Три широко используемых распределенных акторных фреймворка кодируют сообщения следующим образом.

- Фреймворк *Akka* применяет по умолчанию встроенную сериализацию языка Java, не обеспечивающую прямой или обратной совместимости. Однако вы можете заменить ее чем-то вроде Protocol Buffers и таким образом получить возможность выполнять плавающие обновления [50].
- Фреймворк *Orleans* задействует пользовательский формат кодирования данных, не поддерживающий развертывания плавающих обновлений; для развертывания новой версии приложения придется настроить новый кластер, перенаправить трафик со старого кластера на новый и отключить старый [51, 52]. Как и в случе с Akka, есть возможность применять пользовательские плагины для сериализации.
- Во фреймворке *Erlang OTP* менять схемы записей на удивление непросто (несмотря на наличие в системе множества возможностей, предназначенных для обеспечения высокой доступности). Плавающие обновления возможны, но их нужно очень тщательно планировать [53]. Вероятно, их выполнение облегчит экспериментальный новый тип данных `maps` (напоминающая JSON структура, появившаяся в Erlang R17 в 2014 году) [54].

4.3. Резюме

В настоящей главе мы рассмотрели несколько способов преобразования структур данных в байты в сети или на диске. Мы увидели, что нюансы этих форматов кодирования влияют не только на их эффективность, но и, что важнее, на архитектуру использующих их приложений и возможности их развертывания.

В частности, для многих сервисов необходима поддержка плавающих обновлений с постепенным развертыванием новой версии сервиса лишь на нескольких узлах за один раз, а не на всех узлах одновременно. Плавающие обновления дают возможность выпускать новые версии сервисов без вынужденного бездействия системы (тем самым поощряя частые небольшие выпуски, а не редко выполняемые большие) и снижают рискованность развертывания (благодаря обнаружению и откату

сбойных выпусков до того, как они затронут большое количество пользователей). Эти качества чрезвычайно благотворно влияют на *возможность развития* — простоту внесения изменений в приложение.

В случае применения плавающих обновлений или по другим причинам нам приходится допускать возможность того, что на разных узлах работают различные версии кода нашего приложения. Следовательно, важно закодировать все движущиеся по системе данные с поддержкой обратной совместимости (новый код мог читать старые данные) и прямой совместимости (старый мог читать новые).

Мы обсудили несколько форматов кодирования данных и их свойства совместимости.

- Ориентированные на конкретный язык программирования форматы кодирования ограничены этим языком и зачастую не могут обеспечить обратной и прямой совместимости.
- Текстовые форматы, подобные JSON, XML и CSV, широко распространены, и их совместимость зависит от способа их использования. В них есть необязательные дополнительные языки описания схем, иногда полезные, а иногда лишь мешающие. В этих языках типы данных иногда довольно расплывчаты, так что лучше быть осторожными в них с такими вещами, как числа и двоичные строки.
- Основанные на двоичных схемах форматы, например, Thrift, Protocol Buffers и Avro, позволяют выполнять сжатое и эффективное кодирование с четко определенной семантикой обратной и прямой совместимости. Их схемы могут оказаться полезны в целях документирования и генерации кода в языках программирования со статической типизацией. Однако у них есть недостаток: данные необходимо декодировать, чтобы люди смогли их читать.

Мы также обсудили несколько режимов движения потоков данных, проиллюстрировав различные сценарии, в которых формат кодирования данных играет важную роль:

- базы данных, где записывающий в базу процесс кодирует данные, а читающий из нее — декодирует;
- RPC и воплощающие REST API, в которых клиент кодирует запрос, сервер декодирует запрос и кодирует ответ, и наконец, клиент декодирует ответ;
- асинхронная передача сообщений (с помощью брокеров сообщений или акторов), где узлы взаимодействуют путем отправки друг другу сообщений, кодируемых отправителем и декодируемых получателем.

Подытожим: при соблюдении некоторой осторожности вполне можно добиться обратной/прямой совместимости и выполнять плавающие обновления. Пусть же эволюция ваших приложений будет быстрой, а развертывания — частыми.

4.4. Библиография

1. Java Object Serialization Specification. 2010 [Электронный ресурс]. — Режим доступа: http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html.
2. Ruby 2.2.0 API Documentation. Dec 2014 [Электронный ресурс]. — Режим доступа: http://ruby-doc.org/core-2.2.0/.
3. The Python 3.4.3 Standard Library Reference Manual. February 2015 [Электронный ресурс]. — Режим доступа: https://docs.python.org/3/library/pickle.html.
4. EsotericSoftware/kryo. October 2014 [Электронный ресурс]. — Режим доступа: https://github.com/EsotericSoftware/kryo.
5. CWE-502: Deserialization of Untrusted Data // Common Weakness Enumeration, July 30, 2014 [Электронный ресурс]. — Режим доступа: http://cwe.mitre.org/data/definitions/502.html.
6. *Breen S.* What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability. November 6, 2015 [Электронный ресурс]. — Режим доступа: https://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/.
7. *McKenzie P.* What the Rails Security Issue Means for Your Startup. January 31, 2013 [Электронный ресурс]. — Режим доступа: http://www.kalzumeus.com/2013/01/31/what-the-rails-security-issue-means-for-your-startup/.
8. *Smith E.* jvm-serializers wiki. November 2014 [Электронный ресурс]. — Режим доступа: https://github.com/eishay/jvm-serializers/wiki.
9. XML Is a Poor Copy of S-Expressions [Электронный ресурс]. — Режим доступа: http://wiki.c2.com/?XmlIsaPoorCopyOfEssExpressions.
10. *Harris M.* Snowflake: An Update and Some Very Important Information // email to Twitter Development Talk mailing list, October 19, 2010 [Электронный ресурс]. — Режим доступа: https://groups.google.com/forum/#!topic/twitter-development-talk/ahbvo3VTIYI.
11. *Gao S., Sperberg-McQueen C. M., Thompson H. S.* XML Schema 1.1 // W3C Recommendation, May 2001 [Электронный ресурс]. — Режим доступа: https://www.w3.org/XML/Schema.
12. *Galiegue F., Zyp K., Court G.* JSON Schema // IETF InternetDraft, February 2013 [Электронный ресурс]. — Режим доступа: http://json-schema.org/.
13. *Shafranovich Y.* RFC 4180: Common Format and MIME Type for Comma-Separated Values (CSV) Files. October 2005 [Электронный ресурс]. — Режим доступа: https://tools.ietf.org/html/rfc4180.
14. MessagePack Specification [Электронный ресурс]. — Режим доступа: http://msgpack.org/.

15. *Slee M., Agarwal A., Kwiatkowski M.* Thrift: Scalable CrossLanguage Services Implementation // Facebook technical report, April 2007 [Электронный ресурс]. — Режим доступа: http://thrift.apache.org/static/files/thrift-20070401.pdf.
16. Protocol Buffers Developer Guide // Google, Inc. [Электронный ресурс]. — Режим доступа: https://developers.google.com/protocol-buffers/docs/overview.
17. *Anishchenko I.* Thrift vs Protocol Buffers vs Avro — Biased Comparison. September 17, 2012 [Электронный ресурс]. — Режим доступа: https://www.slideshare.net/IgorAnishchenko/pb-vs-thrift-vs-avro.
18. A Matrix of the Features Each Individual Language Library Supports [Электронный ресурс]. — Режим доступа: https://wiki.apache.org/thrift/LibraryFeatures.
19. *Kleppmann M.* Schema Evolution in Avro, Protocol Buffers and Thrift. December 5, 2012 [Электронный ресурс]. — Режим доступа: http://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html.
20. Apache Avro 1.7.7 Documentation. July 2014 [Электронный ресурс]. — Режим доступа: http://avro.apache.org/docs/1.7.7/.
21. *Cutting D., Walters C., Kellerman J., et al.* [PROPOSAL] New Subproject: Avro // email thread on hadoop-general mailing list, April 2009 [Электронный ресурс]. — Режим доступа: http://mail-archives.apache.org/mod_mbox/hadoop-general/200904.mbox/%3C49D53694.1050906@apache.org%3E.
22. *Hoare T.* Null References: The Billion Dollar Mistake // QCon London, March 2009 [Электронный ресурс]. — Режим доступа: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare.
23. *Auradkar A., Quiggle T.* Introducing Espresso — LinkedIn's Hot New Distributed Document Store. January 21, 2015 [Электронный ресурс]. — Режим доступа: https://engineering.linkedin.com/espresso/introducing-espresso-linkedins-hot-new-distributed-document-store.
24. *Kreps J.* Putting Apache Kafka to Use: A Practical Guide to Building a Stream Data Platform (Part 2). February 25, 2015 [Электронный ресурс]. — Режим доступа: https://www.confluent.io/blog/stream-data-platform-2/.
25. *Shapira G.* The Problem of Managing Schemas. November 4, 2014 [Электронный ресурс]. — Режим доступа: https://www.oreilly.com/ideas/the-problem-of-managing-schemas.
26. Apache Pig 0.14.0 Documentation. November 2014 [Электронный ресурс]. — Режим доступа: http://pig.apache.org/docs/r0.14.0/.
27. *Larmouth J.* ASN.1 Complete. Morgan Kaufmann, 1999. http://www.oss.com/asn1/resources/books-whitepapers-pubs/larmouth-asn1-book.pdf.
28. *Housley R., Ford W., Polk T., Solo D.* RFC 2459: Internet X.509 Public Key Infrastructure: Certificate and CRL Profile // IETF Network Working Group, Standards Track, January 1999 [Электронный ресурс]. — Режим доступа: https://www.ietf.org/rfc/rfc2459.txt.

29. *Walkin L.* Question: Extensibility and Dropping Fields. September 21, 2010 [Электронный ресурс]. — Режим доступа: http://lionet.info/asn1c/blog/2010/09/21/question-extensibility-removing-fields/.
30. *Garrett J. J.* Ajax: A New Approach to Web Applications. February 18, 2005 [Электронный ресурс]. — Режим доступа: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/.
31. *Newman S.* Building Microservices. — O'Reilly Media, 2015.
32. *Richardson C.* Microservices: Decomposing Applications for Deployability and Scalability // infoq.com, May 25, 2014 [Электронный ресурс]. — Режим доступа: https://www.infoq.com/articles/microservices-intro.
33. *Helland P.* Data on the Outside Versus Data on the Inside // 2nd Biennial Conference on Innovative Data Systems Research (CIDR), January 2005 [Электронный ресурс]. — Режим доступа: http://cidrdb.org/cidr2005/papers/P12.pdf.
34. *Fielding R. T.* Architectural Styles and the Design of Network-Based Software Architectures // PhD Thesis, University of California, Irvine, 2000 [Электронный ресурс]. — Режим доступа: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.
35. *Fielding R. T.* REST APIs Must Be Hypertext-Driven. October 20 2008 [Электронный ресурс]. — Режим доступа: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.
36. REST in Peace, SOAP. October 15, 2010 [Электронный ресурс]. — Режим доступа: http://royal.pingdom.com/2010/10/15/rest-in-peace-soap/.
37. Web Services Standards as of Q1 2007. February 2007 [Электронный ресурс]. — Режим доступа: https://www.innoq.com/resources/ws-standards-poster/.
38. *Lacey P.* The S Stands for Simple. November 15, 2006 [Электронный ресурс]. — Режим доступа: http://harmful.cat-v.org/software/xml/soap/simple.
39. *Tilkov S.* Interview: Pete Lacey Criticizes Web Services. December 12, 2006 [Электронный ресурс]. — Режим доступа: https://www.infoq.com/articles/pete-lacey-ws-criticism.
40. OpenAPI Specification (fka Swagger RESTful API Documentation Specification) Version 2.0. September 8, 2014 [Электронный ресурс]. — Режим доступа: https://swagger.io/specification/.
41. *Henning M.* The Rise and Fall of CORBA // ACM Queue, volume 4, number 5, pages 28–34, June 2006 [Электронный ресурс]. — Режим доступа: http://queue.acm.org/detail.cfm?id=1142044.
42. *Birrell A. D., Nelson B. J.* Implementing Remote Procedure Calls // ACM Transactions on Computer Systems (TOCS), volume 2, number 1, pages 39–59, February 1984 [Электронный ресурс]. — Режим доступа: http://www.cs.princeton.edu/courses/archive/fall03/cs518/papers/rpc.pdf.

43. *Waldo J., Wyant G., Wollrath A., Kendall S.* A Note on Distributed Computing // Sun Microsystems Laboratories, Inc., Technical Report TR-94-29, November 1994 [Электронный ресурс]. — Режим доступа: http://m.mirror.facebook.net/kde/devel/smli_tr-94-29.pdf.
44. *Vinoski S.* Convenience over Correctness // IEEE Internet Computing, volume 12, number 4, pages 89–92, July 2008 [Электронный ресурс]. — Режим доступа: http://steve.vinoski.net/pdf/IEEE-Convenience_Over_Correctness.pdf.
45. *Eriksen M.* Your Server as a Function // 7th Workshop on Programming Languages and Operating Systems (PLOS), November 2013 [Электронный ресурс]. — Режим доступа: https://monkey.org/~marius/funsrv.pdf.
46. grpc-common Documentation // Google, Inc., February 2015 [Электронный ресурс]. — Режим доступа: https://github.com/grpc/grpc-experiments.
47. *Narayan A., Singh I.* Designing and Versioning Compatible Web Services // IBMMarch 28, 2007 [Электронный ресурс]. — Режим доступа: https://www.ibm.com/developerworks/aboutdw/page-not-available/.
48. *Hunt T.* Your API Versioning Is Wrong, Which Is Why I Decided to Do It 3 Different Wrong Ways. February 10, 2014 [Электронный ресурс]. — Режим доступа: https://www.troyhunt.com/your-api-versioning-is-wrong-which-is/.
49. API Upgrades // Stripe, Inc., April 2015 [Электронный ресурс]. — Режим доступа: https://stripe.com/docs/upgrades.
50. *Bonér J.* Upgrade in an Akka Cluster // email to akka-user mailing list, August 28, 2013 [Электронный ресурс]. — Режим доступа: http://grokbase.com/t/gg/akka-user/138wd8j9e3/upgrade-in-an-akka-cluster.
51. *Bernstein P. A., Bykov S., Geller A., et al.* Orleans: Distributed Virtual Actors for Programmability and Scalability // Microsoft Research Technical Report MSR-TR-2014-41, March 2014 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F210931%2Forleans-msr-tr-2014-41.pdf.
52. Microsoft Project Orleans Documentation // Microsoft Research, 2015 [Электронный ресурс]. — Режим доступа: http://dotnet.github.io/orleans/.
53. *Mercer D., Hinde S., Chen Y., O'Keefe R. A.* beginner: Updating Data Structures // email thread on erlang-questions mailing list, October 29, 2007 [Электронный ресурс]. — Режим доступа: http://erlang.org/pipermail/erlang-questions/2007-October/030318.html.
54. *Hebert F.* Postscript: Maps. April 9, 2014 [Электронный ресурс]. — Режим доступа: http://learnyousomeerlang.com/maps.

Часть II
Распределенные данные

> Для успеха технологии главное — реальные факты, а не «пиар», потому что природу обмануть невозможно.
>
> *Ричард Фейнман.*
> *Отчет комиссии Роджерса[1] (1986)*

В части I этой книги мы обсуждали вопросы систем данных, актуальные при хранении данных на отдельной машине. Теперь, в части II, мы поднимемся на уровень выше и зададимся вопросом: что произойдет, если в хранении и извлечении данных будут участвовать несколько машин?

Существует множество поводов для распределения базы данных по нескольким машинам.

- *Масштабируемость*. Если объем данных, нагрузка по чтению или записи перерастают возможности одной машины, то можно распределить эту нагрузку на несколько компьютеров.
- *Отказоустойчивость/высокая доступность*. Если приложение должно продолжать работать даже в случае сбоя одной из машин (или нескольких машин, или

[1] Комиссия под руководством бывшего госсекретаря США У. П. Роджерса, расследовавшая катастрофу шаттла «Челленджер». — *Примеч. пер.*

сети, или даже всего ЦОДа), то можно использовать избыточные компьютеры. При отказе одного из них выполнение задач делегируется другому.

- *Задержка*. При наличии пользователей по всему миру необходимы серверы в разных точках земного шара, чтобы каждый пользователь обслуживался ЦОДом, географически расположенным максимально близко от него. При этом пользователям не нужно будет ждать, пока сетевые пакеты обойдут половину земного шара.

Масштабирование в расчете на более высокую нагрузку. Если вам всего лишь нужно масштабировать свою систему в расчете на более высокую нагрузку, то простейший способ — купить более мощную машину. Иногда этот подход называют *вертикальным масштабированием* (vertical scaling, scaling up). Можно объединить много процессоров, чипов памяти и жестких дисков под управлением одной операционной системы, а быстрые соединения между ними позволят любому из процессоров обращаться к любой части памяти или диска. В подобной *архитектуре с разделяемой памятью* (shared-memory architecture) можно рассматривать все компоненты как единую машину [1][1].

Главная проблема подхода с разделяемой памятью состоит в том, что стоимость растет быстрее, чем линейно: машина с вдвое большим количеством CPU, вдвое большим количеством памяти и двойным объемом дисков стоит отнюдь не вдвое дороже. Кроме того, вследствие узких мест вдвое более мощная машина не обязательно сможет справиться с двойной нагрузкой.

Архитектура с разделяемой памятью обеспечивает ограниченную отказоустойчивость — в высокопроизводительных машинах есть возможность горячей замены компонентов (дисков, модулей памяти и даже CPU без отключения машины), — но она строго ограничена одной географической точкой.

Другой подход: *архитектура с разделяемым дисковым накопителем* (shared-disk architecture), при которой применяется несколько машин с отдельными CPU и оперативной памятью, но данные хранятся в массиве дисков, совместно используемых всеми машинами, подключенными с помощью быстродействующей сети[2]. Такая архитектура применяется при складировании данных, но конку-

---

[1] В большой машине, хотя все процессоры и могут обращаться ко всем частям памяти, некоторые банки памяти ближе к одним процессорам, чем к другим. Это называется неоднородным доступом к памяти (nonuniform memory access, NUMA [1]). Чтобы эффективно использовать такую архитектуру, обработку необходимо разбить на составные части, чтобы все CPU обращались преимущественно к находящейся поблизости памяти. Следовательно, даже при работе на одной машине не обойтись без секционирования.

[2] Сетевое хранилище данных (network attached storage, NAS) или сеть хранения данных (storage area network, SAN).

ренция и накладные расходы на блокировки ограничивают масштабируемость этого подхода [2].

Архитектуры без разделения ресурсов. Напротив, *архитектуры без разделения ресурсов* (shared-nothing architectures) [3], известные под названием *горизонтального масштабирования* (horizontal scaling, scaling out), приобрели немалую популярность. При этом подходе каждый компьютер или виртуальная машина, на которой работает база данных, называется *узлом* (node). Все узлы используют свои CPU, память и диски независимо друг от друга. Согласование узлов выполняется на уровне программного обеспечения с помощью обычной сети.

Для систем без разделения ресурсов не требуется никакого специального аппаратного обеспечения, так что можно применять машины, имеющие наилучшее соотношение «цена/производительность». При желании можно распределить данные по многим географическим регионам и таким образом снизить задержку для пользователей и потенциально сделать систему устойчивой к потере целого ЦОДа. Благодаря облачному развертыванию виртуальных машин вашей компании не нужно работать в масштабах Google: даже маленькие компании могут позволить себе межрегиональную распределенную архитектуру.

В части II мы сосредоточимся на архитектурах без разделения ресурсов — не потому, что это наилучший вариант для всех сценариев использования, а скорее из-за того, что они требуют от разработчика приложения наибольшей осторожности. В случае распределенных по нескольким узлам данных необходимо осознавать ограничения и компромиссы подобных распределенных систем — БД не может, словно по волшебству, избавить вас от этих нюансов.

Хотя у распределенных архитектур без разделения ресурсов есть много достоинств, обычно они приводят к усложнению приложений, а также иногда ограничивают выразительность используемых моделей данных. В некоторых случаях простая однопоточная программа может продемонстрировать намного лучшую производительность, чем кластер с сотней ядер CPU [4]. С другой стороны, системы без разделения ресурсов могут быть высокопроизводительными. В следующих нескольких главах мы подробно рассмотрим возникающие при распределении данных вопросы.

Репликация или секционирование? Существует два распространенных способа распределения данных по нескольким узлам.

- *Репликация*. Копии одних и тех же данных хранятся в нескольких различных узлах, вероятно располагающихся в разных местах. Репликация обеспечивает избыточность: в случае недоступности части узлов данные можно выдать из остальных. Репликация также подходит для повышения производительности. Мы обсудим ее в главе 5.

- *Секционирование*. Разбиение большой базы данных на небольшие подмножества, называемые *секциями* (partitions), в результате чего разным узлам можно поставить в соответствие различные секции (это называется *«шардинг»*). Мы обсудим секционирование в главе 6.

Это отдельные механизмы, но они часто сопутствуют друг другу, как показано на рис. II.1.

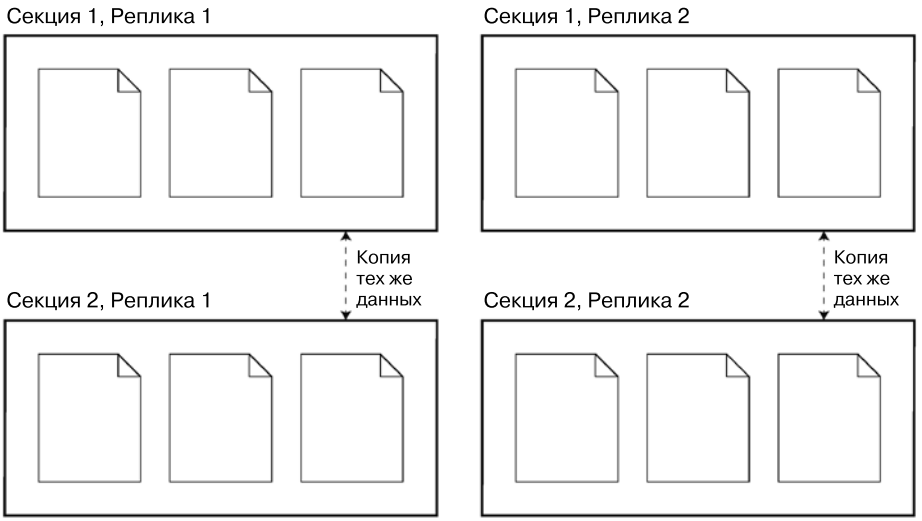


Рис. II.1. База данных, разбитая на две секции, по две реплики на секцию

Разобравшись в этих концепциях, мы сможем обсудить непростые компромиссы, необходимые для распределенных систем. В главе 7 рассмотрим транзакции, так как они помогут вам понять все многочисленные потенциальные проблемы информационных систем и возможные пути их решения. Мы завершим данную часть книги обсуждением фундаментальных ограничений распределенных систем в главах 8 и 9.

Далее, в части III, мы изучим способы создания нескольких (вероятно, распределенных) хранилищ данных и интеграции их в большую систему ради удовлетворения потребностей сложного приложения. Но сначала поговорим о распределенных данных.

Библиография

1. *Drepper U.* What Every Programmer Should Know About Memory // November 21, 2007 [Электронный ресурс]. — Режим доступа: https://www.akkadia.org/drepper/cpumemory.pdf.

2. *Stopford B.* Shared Nothing vs. Shared Disk Architectures: An Independent View. November 24, 2009 [Электронный ресурс]. — Режим доступа: http://www.benstopford.com/2009/11/24/understanding-the-shared-nothing-architecture/.
3. *Stonebraker M.* The Case for Shared Nothing // IEEE Database Engineering Bulletin, volume 9, number 1, pages 4–9, March 1986 [Электронный ресурс]. — Режим доступа: http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf.
4. *McSherry F., Isard M., Murray D. G.* Scalability! But at What COST? // 15th USENIX Workshop on Hot Topics in Operating Systems (HotOS), May 2015 [Электронный ресурс]. — Режим доступа: http://www.frankmcsherry.org/assets/COST.pdf.


К согласованности и консенсусу (глава 9)
ЛОГИЧЕСКОЕ ПОБЕРЕЖЬЕ
RethinkDB
MongoDB
MySQL
ЛЕС МОДЕЛЕЙ СОГЛАСОВАННОСТИ
РЕПЛИКАЦИЯ С ОДНИМ ВЕДУЩИМ УЗЛОМ
Oracle
PostgreSQL
ФИЗИЧЕСКОЕ ПОБЕРЕЖЬЕ
SQL Server
Кратер восстановления после отказа
ПОСЛЕДОВАТЕЛЬНОСТЬ ЗАДЕРЖКИ РЕПЛИКАЦИИ
БУХТА ПРИЧИННО-СЛЕДСТВЕННЫХ СВЯЗЕЙ
Etherpad
Google Docs
ПИКИ РАЗРЕШЕНИЯ КОНФЛИКТОВ
РЕПЛИКАЦИЯ С НЕСКОЛЬКИМИ ВЕДУЩИМИ УЗЛАМИ
Гавань Кворум
CouchDB
Синхронизация календаря
CRDTs
Riak
РЕПЛИКАЦИЯ БЕЗ ВЕДУЩЕГО УЗЛА
БУЛЬВАР СОГЛАСОВАННОСТИ В КОНЕЧНОМ СЧЕТЕ
Voldemort
Cassandra
К журнализированному хранилищу (глава 3)

5 Репликация

> Главное различие между вещью, которая может сломаться, и той, которая не может, состоит в том, что, когда вещь, которая может сломаться, все-таки сломается, до нее невозможно будет добраться и починить.
>
> *Дуглас Адамс. В основном безвредна (1992)*

Репликация (replication) означает хранение копий одних и тех же данных на нескольких машинах, соединенных с помощью сети. Как обсуждалось во введении к части II, существует несколько возможных причин репликации данных:

- ради хранения данных географически близко к пользователям (и сокращения, таким образом, задержек);
- чтобы система могла продолжать работать при отказе некоторых ее частей (и повышения, таким образом, доступности);
- для горизонтального масштабирования количества машин, обслуживающих запросы на чтение (и повышения, таким образом, пропускной способности по чтению).

В настоящей главе мы предполагаем: наш набор данных настолько мал, что можно хранить копию всего набора на каждой из машин. В главе 6 мы откажемся от этого допущения и обсудим *секционирование (шардинг)* наборов данных, которые слишком велики для размещения на отдельной машине. В последующих главах обсудим также различные виды сбоев, которые могут возникнуть в реплицируемых информационных системах, и что с ними делать.

Если реплицируемые данные не меняются с течением времени, то репликация не представляет сложности: просто нужно однократно скопировать их на каждый узел и все. Основные сложности репликации заключаются в том, что делать

с *изменениями* реплицированных данных. Именно этому и посвящена настоящая глава. Мы обсудим три популярных алгоритма репликации изменений между узлами: репликацию с *одним ведущим узлом* (single-leader), с *несколькими ведущими узлами* (multi-leader) и *без ведущего узла* (leaderless). Практически все распределенные базы данных используют один из этих подходов. У каждого из них есть свои плюсы и минусы, которые мы сейчас подробно рассмотрим.

В случае репликации нужно учитывать множество альтернатив: например, использовать синхронную или асинхронную репликацию или что делать со сбойными репликами. Зачастую в базах данных есть для этого конфигурационные настройки, и хотя конкретные детали зависят от базы, общие принципы одинаковы для множества различных реализаций. В этой главе мы обсудим последствия выбора различных альтернатив.

Репликация баз данных — давний предмет изучения, ее принципы не сильно изменились с 1970-х годов, когда их только начали изучать [1], поскольку фундаментальные ограничения сетей остались теми же самыми. Однако за пределами исследовательских лабораторий разработчики продолжали неявно предполагать, что база состоит только из одного узла. Широкое распространение распределенные БД получили лишь недавно. А поскольку многим разработчикам приложений эта сфера мало знакома, такие вопросы, как *конечная согласованность* (eventual consistency), часто толкуются неправильно. В разделе 5.2 мы поговорим более конкретно о конечной согласованности и обсудим такие понятия, как гарантированные *чтение своих записей* (read-your-writes) и *монотонное чтение* (monotonic reads).

5.1. Ведущие и ведомые узлы

Узлы, в которых хранятся копии БД, называются *репликами*. При наличии множества реплик неизбежно возникает вопрос: как обеспечить присутствие всех данных во всех репликах?

Каждая операция записи в базу должна учитываться каждой репликой, иначе нельзя гарантировать, что реплики содержат одни и те же данные. Наиболее распространенное решение этой проблемы называется *репликацией с ведущим узлом* (leader-based replication)[1]. Она показана на рис. 5.1. Схема ее работы следующая.

1. Одна из реплик назначается *ведущим*[2] (leader) узлом. Клиенты, желающие записать данные в базу, должны отправить свои запросы ведущему узлу, который сначала записывает новые данные в свое локальное хранилище.

---

[1] Известно также под названиями «репликации типа “активный/пассивный”» (active/passive replication) или «репликации типа “главный-подчиненный”» (master-slave replication).

[2] Также называется главным (master) или основным (primary) узлом.

2. Другие реплики называются *ведомыми* (followers) узлами[1]. Всякий раз, когда ведущий узел записывает в свое хранилище новые данные, он также отправляет информацию об изменениях данных всем ведомым узлам в качестве части *журнала репликации* (replication log) или *потока изменений* (change stream). Все ведомые узлы получают журнал от ведущего и обновляют соответствующим образом свою локальную копию БД, применяя все операции записи в порядке их обработки ведущим узлом.
3. Когда клиенту требуется прочитать данные из базы, он может выполнить запрос или к ведущему узлу, или к любому из ведомых. Однако запросы на запись разрешено отправлять только ведущему (ведомые с точки зрения клиента предназначены только для чтения).

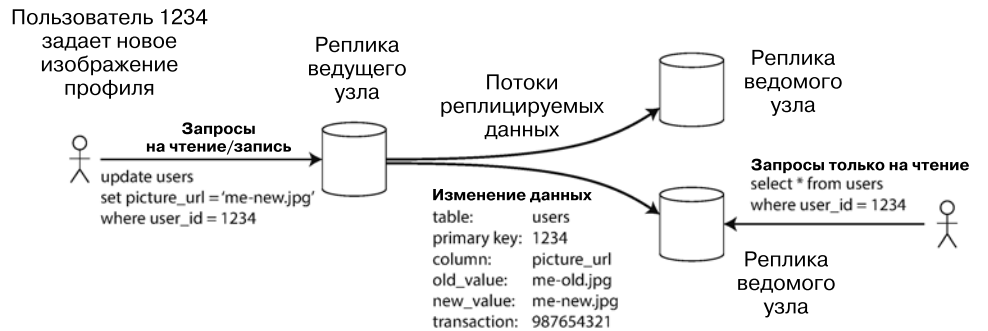


Рис. 5.1. Репликация с ведущим узлом («главный — подчиненный»)

Такой режим репликации — встроенная возможность многих реляционных баз данных, например PostgreSQL (начиная с версии 9.0), MySQL, Oracle Data Guard [2], AlwaysOn Availability Groups СУБД SQL Server [3]. Кроме того, он используется и в некоторых нереляционных БД, включая MongoDB, RethinkDB и Espresso [4]. Наконец, репликация с ведущим узлом применяется не только в базах: распределенные брокеры сообщений, такие как Kafka [5] и высокодоступные очереди сообщений RabbitMQ [6], тоже его применяют. А равно и отдельные сетевые файловые системы и реплицируемые блочные устройства, например DRBD.

---

[1] Другие названия: реплики чтения (read replicas), подчиненные узлы (slaves), вспомогательные узлы (secondaries) или горячий резерв (hot standbys). Разные люди вкладывают разный смысл в понятия горячих, теплых и холодных резервных серверов. В PostgreSQL, например, горячим резервом называются реплики, принимающие запросы на чтение от клиентов, а теплым резервом — обрабатывающие поступающие от ведущего узла изменения, но не запросы от клиентов. Для целей данной книги эти различия несущественны.

Синхронная и асинхронная репликация

Важный фактор работы реплицируемой системы — *синхронно* или *асинхронно* выполняется репликация (в реляционных базах данных этот параметр часто настраиваемый; в коде других систем он обычно жестко «зашит»).

Обратимся еще раз к рис. 5.1, на котором пользователь сайта обновляет свое изображение профиля. В какой-то момент времени клиент отправляет запрос на обновление ведущему узлу, и тот вскоре его получает. Затем ведущий узел пересылает изменения данных ведомым узлам и в конце концов уведомляет клиента об успешном выполнении обновления.

На рис. 5.2 показаны взаимодействия между разными компонентами системы: клиентом пользователя, ведущим узлом и двумя ведомыми узлами. Время растет слева направо. Запрос и ответ показаны в виде жирных стрелок.

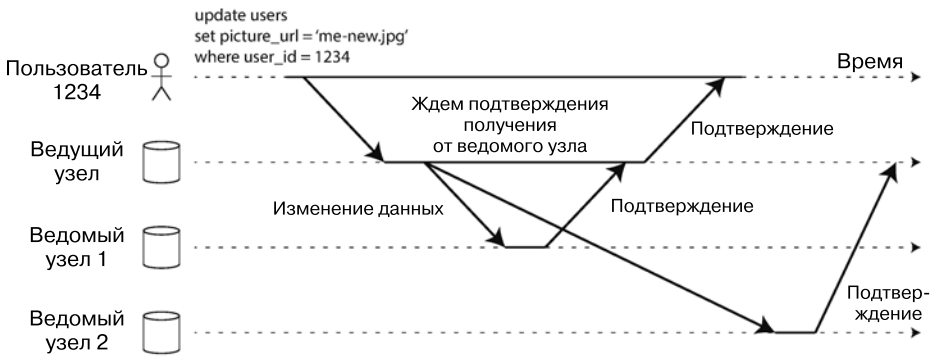


Рис. 5.2. Репликация с ведущим узлом и с одним синхронным и одним асинхронным ведомыми узлами

На примере из рис. 5.2 репликация на ведомый узел 1 *синхронна*: ведущий узел ждет до тех пор, пока ведомый узел 1 не подтвердит получение операции записи, прежде чем сообщить пользователю об успехе и сделать результаты записи видимыми другим клиентам. Репликация на ведомый узел 2 *асинхронна*: ведущий узел отправляет сообщение, но не ждет ответа от ведомого.

На схеме видна существенная задержка перед обработкой сообщения ведомым узлом 2. Обычно репликация выполняется довольно быстро: большинство СУБД вносят изменения на ведомые узлы менее чем за секунду. Однако гарантий относительно длительности этого процесса нет. Бывают обстоятельства, когда ведомые узлы запаздывают, по сравнению с ведущим, на несколько минут или более: например, когда ведомый восстанавливается после отказа, когда система работает практически на пределе возможностей или когда при связи между узлами возникают проблемы в работе сети.

Преимущество синхронной репликации: копия данных на ведомом узле гарантированно актуальна и согласуется с ведущим узлом. В случае внезапного сбоя последнего можно быть уверенным, что данные по-прежнему доступны на ведомом узле. Недостаток же состоит в следующем: если синхронный ведомый узел не отвечает (из-за его сбоя, или сбоя сети, или по любой другой причине), то операцию записи завершить не удастся. Ведущему узлу придется блокировать все операции записи и ждать до тех пор, пока синхронная реплика не станет снова доступна.

Поэтому делать все ведомые узлы синхронными неразумно: перебой в обслуживании одного любого узла приведет к замедлению работы системы вплоть до полной остановки. На практике активизация в СУБД синхронной репликации обычно означает, что *один* из ведомых узлов — синхронный, а остальные — асинхронны. В случае замедления или недоступности синхронного ведомого узла в него превращается один из асинхронных ведомых узлов. Это гарантирует наличие актуальной копии данных по крайней мере на двух узлах: ведущем и одном синхронном ведомом. Такая конфигурация иногда называется *полусинхронной* (semi-synchronous) [7].

Зачастую репликация с ведущим узлом делается полностью асинхронной. В этом случае при фатальном сбое ведущего узла все еще не реплицированные на ведомые узлы операции записи теряются. Это значит, что сохраняемость записи не гарантируется, даже если клиент получил ее подтверждение. Однако преимуществом полностью асинхронной конфигурации является возможность ведущего узла выполнять операции записи даже в случае запаздывания всех ведомых.

Ослабление сохраняемости может показаться плохим компромиссом, но асинхронная репликация тем не менее широко используется, особенно при наличии множества ведомых узлов или их географической разбросанности. Мы вернемся к этому вопросу в разделе 5.2.

Исследования вопроса репликации

Для асинхронных реплицируемых систем потеря данных в случае сбоя ведущего узла может оказаться серьезной проблемой, поэтому исследователи продолжают изучать те методы репликации, которые позволяют не терять данные и сохранять хорошую производительность и доступность. Например, *цепная репликация* (chain replication) [8, 9] представляет собой вариант синхронной репликации, успешно реализованный в нескольких системах, например в Microsoft Azure Storage [10, 11].

Существует тесная связь между согласованностью репликации и *консенсусом* (согласованием значения между несколькими узлами), и мы изучим эту теорию подробнее в главе 9. В текущей же главе мы сосредоточимся на более простых формах репликации, чаще всего используемых на практике в БД.

Создание новых ведомых узлов

Время от времени приходится создавать новые ведомые узлы — с целью увеличить количество реплик или заменить сбойные узлы. Как же гарантировать, что новый ведомый узел будет содержать точную копию данных ведущего?

Простого копирования данных с одного узла на другой обычно недостаточно: клиенты постоянно пишут данные в базу и данные постоянно меняются, так что при обычном копировании файлов разные части БД будут скопированы в разные моменты времени. Результат окажется совершенно бессмысленным.

Обеспечить согласованность файлов на диске можно путем блокировки базы данных (сделав ее недоступной для операций записи), но это идет вразрез с нашей целью: обеспечить высокую доступность. К счастью, создать ведомый узел обычно можно без простоя системы. По сути, такой процесс выглядит следующим образом.

1. Сделать согласованный снимок состояния БД ведущего узла на определенный момент времени — по возможности без блокировки всей базы. В большинстве баз такая возможность есть, так как она нужна и для резервного копирования. В некоторых случаях понадобятся сторонние утилиты, например *innobackupex* для СУБД MySQL [12].
2. Скопировать снимок состояния на новый ведомый узел.
3. Ведомый узел подключается к ведущему и запрашивает все изменения данных, произошедшие с момента создания снимка. Для этого нужно, чтобы снимок состояния соотносился с определенной позицией в журнале репликации ведущего узла. Сама позиция называется по-разному: в PostgreSQL — *регистрационным номером транзакции в журнале* (log sequence number), в MySQL — *координатами в бинарном журнале* (binlog coordinates).
4. Когда ведомый узел завершил обработку изменений данных, произошедших с момента снимка состояния, говорят, что он *наверстал упущенное*. После этого он может продолжать обрабатывать поступающие от ведущего узла изменения данных.

На практике создание и настройка ведомого узла очень зависит от используемой базы данных. В ряде систем этот процесс полностью автоматизирован, а в других — представляет собой запутанную многошаговую последовательность действий, которую должен вручную выполнить администратор.

Перебои в обслуживании узлов

Любой узел в системе способен приостановить работу: из-за сбоя или запланированного технического обслуживания (например, перезагрузки машины для установки исправления уязвимости ядра системы). Возможность перезагружать отдельные узлы без простоя системы — огромное преимущество в смысле удобства эксплуатации и обслуживания. Следовательно, наша цель состоит в том, чтобы

система продолжала работать, несмотря на отказы отдельных узлов, а перебои в обслуживании узлов влияли на ее работу как можно меньше.

Как же добиться высокой доступности с помощью репликации с одним ведущим узлом?

Отказ ведомого узла: наверстывающее восстановление

Каждый ведомый узел хранит на своем жестком диске журнал полученных от ведущего изменений данных. В случае сбоя и перезагрузки ведомого узла или временного прекращения работы участка сети между ведущим и ведомым узлами последний может легко возобновить работу: из своего журнала он знает, какая транзакция была обработана перед сбоем. Следовательно, ведомый узел способен подключиться к ведущему и запросить все изменения данных, имевшие место за то время, пока он был недоступен.

Отказ ведущего узла: восстановление после отказа

Справиться с отказом ведущего узла сложнее: необходимо «повысить в звании» один из ведомых до ведущего, настроить клиенты на отправку записей новому ведущему, а другие ведомые должны начать получать изменения данных от нового ведущего. Этот процесс называется *восстановлением после отказа* (failover).

Восстановление после отказа может выполняться вручную (администратор получает оповещение об отказе ведущего узла и принимает соответствующие меры по созданию нового ведущего) или автоматически. Автоматически процесс восстановления после отказа обычно состоит из следующих шагов.

1. *Установить отказ ведущего узла.* Многое может потенциально пойти не так: фатальные сбои, перебои питания, сетевые проблемы и т. д. Не существует надежного способа определить, что именно пошло не так. Поэтому во многих системах для данной цели используется превышение времени ожидания: узлы постоянно обмениваются сообщениями друг с другом, и если один из них не отвечает в течение определенного времени (скажем, 30 секунд), то считается, что он не работает (это не относится к случаю преднамеренного отключения ведущего узла для запланированного техобслуживания).
2. *Выбрать новый ведущий узел.* Здесь можно задействовать процесс «выборов» (ведущий узел выбирается в соответствии с большинством оставшихся реплик), или же этот ведущий назначает предварительно выбранный *узел-контроллер* (controller node). Оптимальным кандидатом на роль нового ведущего узла обычно является реплика с наиболее свежими изменениями данных, полученными от старого (в целях минимизации потерь данных). Согласование нового ведущего между всеми узлами — задача консенсуса, которую мы обсудим подробнее в главе 9.

3. *Настроить систему на использование нового ведущего узла*. Клиенты должны начать отправлять запросы на запись новому ведущему узлу (мы обсудим это в разделе 6.5). Если старый ведущий узел возобновляет работу, может оказаться, что он продолжает считать себя ведущим и не осознает решение остальных реплик считать его недееспособным. Система должна обеспечить превращение старого ведущего узла в ведомый и признание им нового ведущего.

Восстановление после отказа битком набито потенциальными проблемами.

- При использовании асинхронной репликации новый ведущий узел мог не получить все сообщения о записи от старого из-за отказа последнего. Что произойдет с этими записями, если старый ведущий узел попытается присоединиться к кластеру после того, как уже был выбран новый? Новый ведущий мог тем временем получить информацию о конфликтующих операциях записи. Наиболее частое решение: попросту отбросить нереплицированные записи старого ведущего узла, что, однако, похоронит надежды клиентов на сохраняемость данных.
- Отбрасывание записей представляет особенную опасность при необходимости согласовывать содержимое БД с внешними системами хранения. Например, однажды на GitHub [13] ведомый узел с неактуальными данными был «повышен в звании» до ведущего. База задействовала автоматически увеличиваемый счетчик для присвоения первичных ключей новым строкам, но вследствие отставания счетчика нового ведущего узла от старого он начал применять часть первичных ключей, уже выделенных старым ведущим узлом. Эти первичные ключи также использовались в хранилище Redis; повторное применение первичных ключей привело к несогласованности между MySQL и Redis, что, в свою очередь, привело к раскрытию определенных закрытых данных не тем пользователям.
- При определенных сценариях сбоев (см. главу 8) может оказаться так, что два узла будут одновременно считать себя ведущими. Такая ситуация называется «разделение вычислительных мощностей», и она опасна: если оба ведущих узла принимают информацию об операциях записи и не существует процесса разрешения конфликтов (см. раздел 5.3), то возникает большая вероятность потери или повреждения данных. Во многих системах в качестве предохранителя присутствует механизм отключения одного из них в случае обнаружения двух ведущих узлов[1]. Однако если не спроектировать этот механизм очень тщательно, то оба узла могут в итоге перестать работать [14].
- Сколько следует ждать, прежде чем объявить ведущий узел недоступным? Долгое ожидание означает более длительное время восстановления в случае отказа ведущего узла. Однако слишком короткое время ожидания приведет

[1] Этот подход известен под названием «ограждение» или, говоря более выразительным языком, «убийство лишнего узла» (shoot the other node in the head, STONITH). Мы обсудим ограждение подробнее в пункте «Ведущий узел и блокировки» подраздела «Истина определяется большинством» раздела 8.4.

к ненужным восстановлениям после мнимых отказов. Например, временный пик нагрузки может привести к увеличению времени реакции узла сверх заданного времени ожидания или сбой сети — повлечь задержку пакетов. А если система и так страдает от высокой нагрузки или проблем с сетью, то лишнее восстановление после отказа, вероятно, только ухудшит ситуацию.

Легких решений описанных проблем не существует. Поэтому часть обслуживающего персонала предпочитает выполнять восстановления после отказов в ручном режиме, даже если программное обеспечение поддерживает возможность автоматического.

Эти вопросы — отказы узлов, ненадежные сети и компромиссы, связанные с согласованностью реплик, надежностью, доступностью и временем задержки, — по сути, фундаментальные проблемы распределенных систем. В главах 8 и 9 мы обсудим их подробнее.

Реализация журналов репликации

Что же находится «за кулисами» репликации с ведущим узлом? На практике используется несколько различных методов репликации. Рассмотрим их все вкратце.

Операторная репликация

В простейшем случае ведущий узел записывает в журнал каждый выполняемый запрос на запись *(оператор)* и отправляет данный журнал выполнения операторов ведомым узлам. В случае реляционной БД это значит, что каждый оператор `INSERT`, `UPDATE` или `DELETE` пересылается ведомым узлам, и каждый ведомый узел производит синтаксический разбор и выполнение этого оператора SQL так, как если бы он был получен от клиента.

Хотя описанное может показаться разумным подходом к репликации, существует много сценариев, при которых он не будет работать.

- Все операторы, вызывающие недетерминированные функции (например, функцию `NOW()` для получения текущего времени или `RAND()` — для случайного числа), вероятно, будут генерировать разные значения для каждой реплики.
- Если операторы используют столбец с автоматически увеличиваемым значением или зависят от существующих данных из базы (например, `UPDATE … WHERE <какое-то условие>`), то они должны выполняться на всех репликах в строго одинаковом порядке, иначе их результаты будут различными. Это может быть неудобно в случае множества параллельно выполняемых транзакций.
- Операторы с побочными действиями (например, триггеры, хранимые процедуры, пользовательские функции) могут приводить к различным побочным действиям на разных репликах, за исключением случая, когда все побочные действия полностью детерминированы.

Эти проблемы решаемы: например, ведущий узел может заменять все вызовы недетерминированных функций фиксированным возвращаемым значением при записи оператора в журнал, чтобы все ведомые узлы получали одно значение. Однако в связи с большим количеством граничных случаев обычно предпочитают другие методы репликации.

Операторная репликация использовалась в MySQL вплоть до версии 5.1. Она иногда применяется и сейчас, в силу своей компактности, но по умолчанию MySQL переключается на построчную репликацию (которую мы скоро рассмотрим) при малейших признаках недетерминизма в операторе. VoltDB задействует операторную репликацию и обеспечивает ее надежную работу путем требования детерминизма транзакций [15].

Перенос журнала упреждающей записи (WAL)

В главе 3 мы обсуждали, как подсистемы хранения представляют данные на диске, и оказалось, что практически все записываемые данные заносятся в журнал.

- В случае журналированной подсистемы хранения (см. подраздел «SS-таблицы и LSM-деревья» раздела 3.1) этот журнал представляет собой основное место хранения информации. Сегменты его сжимаются и подвергаются сборке мусора в фоновом режиме.
- В случае B-дерева (см. подраздел «B-деревья» раздела 3.1), перезаписывающего отдельные дисковые блоки, все изменения сначала записываются в журнал упреждающей записи, чтобы индекс можно было вернуть в согласованное состояние после фатального сбоя.

Так или иначе журнал представляет собой предназначенную только для дописывания данных в конец последовательность байтов, содержащую результаты всех операций записи в БД. Можно воспользоваться тем же журналом для создания реплики на другом узле: помимо записи журнала на диск, ведущий узел также отправляет его по сети ведомым узлам. А ведомые узлы, обрабатывая этот журнал, создают точные копии тех же структур данных, что и на ведущем.

Помимо прочего, этот метод репликации используется в СУБД PostgreSQL и Oracle [16]. Основной его недостаток состоит в том, что журнал описывает данные на очень низком уровне: WAL содержит все подробности того, какие байты менялись в тех или иных дисковых блоках. Это тесно связывает репликацию с подсистемой хранения. Если база данных меняет формат хранения с одной версии на другую, то обычно становится невозможной работа различных версий СУБД на ведущем и ведомых узлах.

На первый взгляд, это лишь незначительный нюанс реализации, но влияние его на эксплуатацию огромно. Если протокол репликации допускает использование

ведомыми узлами более нового программного обеспечения, чем ведущим, то появляется возможность выполнять обновление ПО базы данных без всякого простоя: сначала обновляются ведомые узлы, а затем производится восстановление после отказа, чтобы сделать один из этих обновленных узлов новым ведущим. Если же протокол репликации не допускает подобного расхождения версий, как часто бывает при переносе журнала упреждающей записи, то подобные обновления требуют простоя системы.

Логическая (построчная) журнальная репликация

Альтернатива — использовать разные форматы журнала для репликации и подсистемы хранения; это даст возможность расцепить журнал репликации с внутренним устройством подсистемы хранения. Такой вид журнала называется *логическим журналом* (logical log), чтобы различать его с *физическим* представлением данных подсистемы хранения.

Логический журнал для реляционных баз данных обычно представляет собой последовательность строк, описывающих операции записи в таблицы базы на уровне строк:

- при вставке строки журнал включает новые значения всех столбцов;
- при удалении строки журнал содержит информацию, достаточную для однозначной идентификации удаляемой строки. Обычно это первичный ключ, но если в таблице он отсутствует, то сохраняются старые значения всех столбцов;
- при обновлении строки журнал содержит информацию, достаточную для однозначной идентификации обновляемой строки, и новые значения всех столбцов (или по крайней мере новые значения всех изменившихся столбцов).

Транзакция, меняющая несколько строк, приводит к генерации нескольких подобных записей в журнале, за которыми следует запись, указывающая, что транзакция была зафиксирована. Этот подход использует бинарный журнал СУБД MySQL (в случае, когда он настроен на применение построчной репликации) [17].

Поскольку логический журнал расцеплен с внутренним устройством подсистемы хранения, оказывается проще поддерживать его обратную совместимость, благодаря чему на ведущем и ведомых узлах могут выполняться различные версии СУБД или даже различные подсистемы хранения.

Формат логического журнала также удобнее для синтаксического разбора внешними приложениями. Этот аспект играет важную роль при необходимости отправить содержимое БД во внешнюю систему, например в склад данных для офлайн-анализа или построения пользовательских индексов и кэшей [18]. Такая методика называется *захватом изменений данных* (change data capture), и мы вернемся к этому вопросу в главе 11.

Триггерная репликация

Описанные до сих пор подходы к репликации реализовались СУБД без участия кода приложения. Во многих случаях именно это и нужно, но встречаются обстоятельства, когда требуется большая гибкость. Например, если необходимо реплицировать только подмножество данных, или выполнить репликацию из БД одного типа в БД другого, или задействовать логику разрешения конфликтов (см. подраздел «Обработка конфликтов записи» раздела 5.3), то может понадобиться перенести репликацию на уровень приложения.

Некоторые утилиты, например Oracle GoldenGate [19], позволяют приложению получить доступ к данным с помощью чтения журнала БД. Или же можно воспользоваться возможностями, имеющимися во многих реляционных БД: *триггерами* (trigger) и *хранимыми процедурами* (stored procedure).

Триггеры позволяют регистрировать пользовательский код, автоматически запускаемый при возникновении в БД события изменения данных (транзакции записи). Триггер получает возможность занести изменения в отдельную таблицу, из которой их сможет прочитать внешний процесс. Затем этот внешний процесс сможет применить любую логику приложения, которая только понадобится, и реплицировать изменения данных в другую систему. Подобным образом работают, например, системы Databus для СУБД Oracle [20] и Bucardo для СУБД Postgres [21].

Накладные расходы при триггерной репликации обычно выше, чем при других типах репликации, и она сильнее подвержена ошибкам и более ограниченна, чем встроенная репликация базы данных. Однако благодаря своей гибкости может оказаться полезной.

5.2. Проблемы задержки репликации

Способность выдерживать без проблем отказы узлов — лишь одна причина для применения репликации. Как упоминалось во введении к части II, также в числе причин — масштабируемость (обработка большего количества запросов, чем доступно одной машине) и снижение времени ожидания (размещение реплик в географической близости от пользователей).

Репликация с ведущим узлом требует, чтобы все операции записи проходили через один узел, но запросы только на чтение могут поступать на любой узел. Для типа нагрузки, состоящей по большей части из операций чтения и лишь небольшого процента операций записи (часто встречающийся в Интернете паттерн), существует привлекательная возможность: создать множество ведомых узлов и распределить запросы на чтение по ним. Это снизит нагрузку на ведущий узел и позволит ближайшим репликам обслуживать запросы на чтение.

В такой *масштабируемой по чтению архитектуре* (read-scaling architecture) можно увеличивать возможности выдачи запросов только на чтение просто с помощью добавления новых ведомых узлов. Однако этот подход реально работает только при асинхронной репликации — если попытаться синхронно реплицировать на все ведомые узлы, то отказ одного-единственного узла или перебой в обслуживании сети сделает всю систему недоступной для операций записи. И чем больше у вас узлов, тем вероятнее отказ одного из них, так что полностью синхронный вариант репликации очень ненадежен.

К сожалению, приложение, читающее данные с *асинхронного* ведомого узла, может получать устаревшую информацию, если такой узел запаздывает. Это приводит к очевидной несогласованности БД: при одновременном выполнении одного и того же запроса на ведущем и ведомом узле результаты могут оказаться различными, поскольку не все операции записи были воспроизведены на ведомом узле. Описанная несогласованность — лишь временное состояние, если прекратить запись в базу данных и подождать немного, ведомые узлы постепенно наверстают упущенное и окажутся согласованными с ведущим узлом. Поэтому такой эффект называется *конечной согласованностью* (eventual consistency) [22, 23][1].

Термин «конечная» умышленно сделан столь неопределенным: в целом нет предела тому, насколько сильно может запаздывать реплика. При обычной эксплуатации задержка между операцией записи на ведущем узле и ее воспроизведением на ведомом узле — *задержка репликации* (replication lag) — может составлять лишь доли секунды и на практике быть совсем незаметной. Однако если система работает на пределе возможностей или присутствуют проблемы с сетью, задержка легко способна вырасти до нескольких секунд или даже минут.

При столь длительной задержке вносимые ею несогласованности представляют не только теоретическую, но и практическую проблему для приложений. В этом разделе мы отметим три проблемы, часто возникающие при задержке репликации, и обрисуем несколько подходов к их решению.

Читаем свои же записи

Многие приложения позволяют пользователю отправить какие-либо данные, а затем просмотреть, что было отправлено. Это может быть запись в пользовательской БД, комментарий в ветке обсуждения или нечто подобное. Новые данные должны сначала отправляться ведущему узлу, но при просмотре пользователем могут

[1] Термин «согласованность в конечном счете» был придуман Дугласом Терри (Douglas Terry) и др. [24], популяризован известным Вернером Вогельсом (Werner Vogels) [22] и стал лозунгом множества MySQL-проектов. Однако не только базы данных NoSQL согласованы в конечном счете: ведомые узлы в асинхронно реплицируемых реляционных БД обладают теми же свойствами.

читаться с одного из ведомых узлов. Это особенно удобно, если данные часто просматриваются, но редко записываются.

В случае асинхронной репликации возникает проблема, показанная на рис. 5.3: если пользователь просматривает данные сразу после выполнения записи, то существует вероятность, что они еще не достигли реплики. Пользователь может счесть при этом, что данные были потеряны, чему он вряд ли обрадуется.



Рис. 5.3. Пользователь выполняет операцию записи с последующим чтением из устаревшей реплики. Для предотвращения этой аномалии требуется согласованность типа «чтение после записи»

В этом случае нам необходима согласованность типа *«чтение после записи»* (read-after-write consistency), называемая также *чтением своих записей* (read-your-writes) [24]. Она гарантирует, что, перезагрузив страницу, пользователь точно увидит внесенные им же изменения. Относительно других пользователей она ничего не обещает: внесенные ими изменения могут еще некоторое время не быть видны. Однако пользователь может быть уверен: внесенные им самим данные были сохранены правильно.

Как же реализовать согласованность типа «чтение после записи» в системе с репликацией с ведущим узлом? Существует несколько возможных способов. Упомяну лишь несколько.

- Данные, которые пользователь мог изменить, читаются с ведущего узла, а все остальные — с одного из ведомых. Для этого нужно как-то узнать, было ли что-то изменено, не выполняя при этом запрос. Например, информация из профиля пользователя в соцсети обычно доступна для редактирования только владельцу профиля. Следовательно, есть простое правило: всегда читать собственный профиль пользователя с ведущего узла, а профили других пользователей — с ведомых.
- Если пользователь способен потенциально редактировать почти все в приложении, то предыдущий подход окажется неэффективным, так как пришлось бы

почти все читать с ведущего узла (что сведет на нет эффект от масштабирования по чтению). В этом случае можно применять другие критерии определения того, следует ли читать с ведущего узла. Например, отслеживать время последнего обновления и в течение одной минуты после этого читать все с ведущего узла. Можно также следить за задержкой репликации на ведомые узлы и предотвращать выполнение запросов на любых узлах, запаздывающих по сравнению с ведущим более чем на минуту.

- Клиент способен запоминать метку даты/времени своей последней операции записи — тогда система сможет гарантировать, что обслуживающая все операции чтения этого пользователя реплика отражает изменения по крайней мере по состоянию на соответствующий метке момент времени. Если же реплика недостаточно актуальна, то можно или делегировать операцию чтения другой реплике, или отложить выполнение запроса, пока реплика не наверстает упущенное. Эта метка может быть *логической* (отражающей последовательность операций записи, например регистрационный номер транзакции в журнале) или фактическим временем системы (в таком случае критически важна синхронизация часов, см. раздел 8.3).
- Если реплики распределены по нескольким ЦОДам (ради географической близости к пользователю), то возникают дополнительные сложности. Все запросы, подлежащие обслуживанию ведущим узлом, необходимо маршрутизировать на ЦОД, в котором находится этот узел.

Еще одна сложность возникает, когда один и тот же пользователь обращается к сервису с разных устройств, например из браузера настольного компьютера и мобильного приложения. В этом случае может пригодиться согласованность типа «чтение после записи» *между разными устройствами*: если пользователь вводит информацию на одном устройстве, а затем просматривает ее с другого, то должен увидеть только что введенную информацию.

В этом случае следует учесть дополнительные нюансы.

- Реализовать подходы, требующие запоминания метки даты/времени последнего выполненного пользователем обновления, становится сложнее, поскольку выполняемый на одном устройстве код не знает про обновления на другом. Эти метаданные необходимо сосредоточить в одном месте.
- При распределении реплик по нескольким ЦОДам нет никаких гарантий, что подключения с разных устройств будут маршрутизированы в один ЦОД (например, настольный компьютер пользователя применяет домашнее широкополосное соединение, а его смартфон — сотовую систему передачи данных; сетевые маршруты устройств при этом могут быть совершенно различными). Если задействованный подход требует чтения с ведущего узла, то необходимо сначала маршрутизировать запросы со всех устройств пользователя в один ЦОД.

Монотонные чтения

Наш второй пример аномалии, происходящей при чтении с асинхронных ведомых узлов: пользователь может наблюдать движение в *обратном направлении времени*.

Это может произойти при выполнении пользователем нескольких операций чтения из различных реплик. Например, рис. 5.4 демонстрирует, как пользователь 2345 выполняет один и тот же запрос дважды: первый раз к ведомому узлу, который запаздывает лишь чуть-чуть, а второй — к ведомому узлу, запаздывающему сильнее. (Такой сценарий вполне вероятен при обновлении пользователем страницы, если все запросы маршрутизируются к случайным серверам.) Первый запрос возвращает комментарий, недавно добавленный пользователем 1234, а второй не возвращает ничего, поскольку сильно запаздывающий ведомый узел еще не обработал эту операцию записи. Фактически второй запрос видит систему по состоянию на более ранний момент времени, чем первый. Это было бы не так плохо, если бы первый запрос ничего не возвращал, поскольку пользователь 2345, вероятно, понятия не имеет о комментарии пользователя 1234, добавленном только что. Однако будет очень странно, если пользователь 2345 увидит, как появляется, а затем исчезает комментарий пользователя 1234.



Рис. 5.4. Пользователь сначала читает данные из актуальной реплики, а затем из устаревшей. Создается впечатление, что время движется в обратную сторону. Для предотвращения этой аномалии необходимо монотонное чтение записей

Монотонное чтение записей [23] гарантирует, что подобная аномалия не произойдет. Это не такая надежная гарантия, как *сильная согласованность* (strong consistency), но более твердая, чем конечная согласованность. Вы можете увидеть старое значение при чтении; монотонное чтение означает только одно: при не-

скольких операциях чтения подряд пользователь не увидит продвижения обратно по времени, то есть старые данные не будут читаться после новых.

Один из способов реализации монотонного чтения записей — обеспечить чтение данных всеми пользователями из одной и той же реплики (различные пользователи могут читать из разных реплик). Например, можно выбирать реплику не случайным образом, а на основе хеша идентификатора пользователя. Однако в случае сбоя реплики пользовательские запросы нужно будет перенаправить на другую реплику.

Согласованное префиксное чтение

Наш третий пример аномалий задержки репликации касается нарушения причинно-следственной связи. Представьте следующий диалог между мистером Пунсом и миссис Кейк.

Мистер Пунс: Насколько далеко в будущее вы можете заглянуть, миссис Кейк?

Миссис Кейк: Обычно секунд на десять, мистер Пунс.

Между этими двумя фразами существует причинная зависимость: миссис Кейк услышала вопрос мистера Пунса и ответила на него.

Теперь представьте третьего человека, слушающего данный разговор через ведомые узлы. Сказанное миссис Кейк проходит через ведомый узел с небольшой задержкой, но задержка репликации сказанного мистером Пунсом значительно больше (рис. 5.5). Этот наблюдатель услышит следующее.

Миссис Кейк: Обычно секунд на десять, мистер Пунс.

Мистер Пунс: Насколько далеко в будущее вы можете заглянуть, миссис Кейк?

С точки зрения наблюдателя, миссис Кейк отвечает на вопрос еще до того, как мистер Пунс его задал. Подобные экстрасенсорные способности впечатляют, но несколько озадачивают [25].

Для предотвращения подобных аномалий необходима другая разновидность гарантий: *согласованное префиксное чтение* (consistent prefix reads) [23]. Она гарантирует, что если операции записи выполняются в определенной последовательности, то в ней же они будут и прочитаны.

Реализация этого метода особенно сложна в секционированных (подвергнутых шардингу) базах данных, которые мы обсудим в главе 6. Если БД всегда применяет операции записи в одном и том же порядке, то префикс при чтении всегда будет согласованным, так что аномалии не случится. Однако во многих распределенных базах различные секции функционируют независимо, поэтому глобального

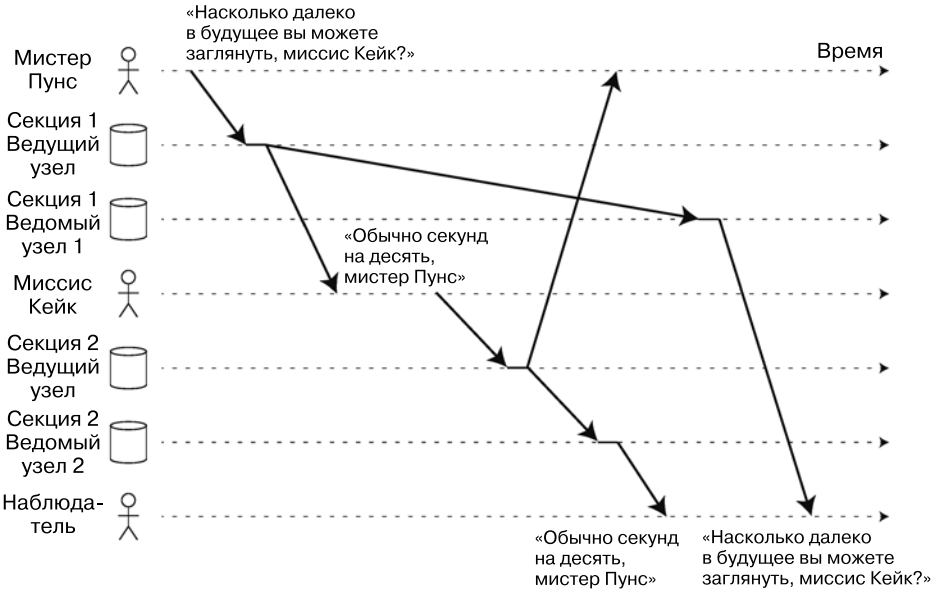


Рис. 5.5. Если часть секций реплицируется дольше, чем другие, наблюдатель может услышать ответ еще до вопроса

упорядочения операций записи нет: при чтении из БД пользователи видят одни части базы в более старом состоянии, а другие — в более новом.

Одно из решений данной проблемы — гарантировать, что все операции записи, связанные друг с другом, записываются в одну секцию, но в ряде приложений сделать это эффективно невозможно. Существуют также алгоритмы явного отслеживания причинно-следственных зависимостей — тема, к которой мы вернемся в пункте «Связь типа “происходит до” и конкурентный доступ» подраздела «Обнаружение конкурентных операций записи» раздела 5.4.

Решения проблемы задержки репликации

При работе с системами с конечной согласованностью не помешает задуматься о том, как поведет себя приложение, если задержка репликации возрастет до нескольких минут или даже часов. При ответе «никаких проблем» все отлично. Однако если в результате пользователи будут испытывать неудобства, то важно спроектировать систему с более сильными гарантиями, например, чтение после записи. Притворяться, что система синхронна, в то время как она асинхронна, — верный рецепт привлечения на свою голову проблем.

Как уже обсуждалось ранее, есть способы, с помощью которых приложение может обеспечить более сильные гарантии, чем лежащая в его основе база данных, —

например, путем выполнения определенных видов операций чтения в ведущем узле. Однако решить эти проблемы в коде приложения непросто, и здесь можно легко ошибиться.

Было бы лучше, если бы разработчики приложений не должны были заботиться о подобных тонких вопросах репликации, а могли просто довериться своим базам данных. Транзакции и существуют для упрощения приложений — как способ обеспечения БД более надежных гарантий.

Одноузловые транзакции существуют уже долгое время. Однако при переходе к распределенным (реплицируемым и секционированным) базам многие системы отказались от них, заявив, что эти транзакции слишком дорогостоящи в смысле производительности и доступности, а конечная согласованность неизбежна в масштабируемой системе. В данном утверждении есть доля правды, но оно излишне упрощает ситуацию. На протяжении книги мы выработаем более дифференцированное мнение по этому вопросу. Мы вернемся к теме транзакций в главах 7 и 9 и обсудим некоторые альтернативные механизмы в части III.

5.3. Репликация с несколькими ведущими узлами

До сих пор в этой главе мы рассматривали только варианты архитектур репликации с одним ведущим узлом. Хотя такой подход является наиболее распространенным, существуют и другие интересные варианты.

У репликации с ведущим узлом есть один крупный недостаток: только один ведущий узел, через который должны проходить все операции записи[1]. Если подключиться к узлу невозможно, например, из-за разрыва сети между вами и ведущим узлом, то нельзя выполнить и запись в базу данных.

Напрашивающееся расширение модели репликации с ведущим узлом — разрешение приема запросов на запись более чем одному узлу. Репликация будет выполняться так же, как и ранее: каждый узел, обрабатывающий операцию записи, должен перенаправлять информацию о данных изменениях всем остальным узлам. Это называется схемой *репликации с несколькими ведущими узлами* (multi-leader replication), или *репликацией типа «главный — главный»* (master — master), или *репликацией типа «активный/активный»* (active/active replication). При такой схеме каждый из ведущих узлов одновременно является ведомым для других ведущих.

---

[1] Если база данных секционирована (см. главу 6), то у каждой секции есть по ведущему узлу. У разных секций ведущими могут быть разные узлы, но у каждой секции тем не менее должен быть один ведущий узел.

Сценарии использования репликации с несколькими ведущими узлами

Использовать репликацию с несколькими ведущими узлами в пределах одного ЦОДа редко имеет смысл, поскольку выгоды от такого решения едва ли перевесят привнесенное усложнение. Однако существуют ситуации, в которых применение этой схемы оправдывает себя.

Эксплуатация с несколькими ЦОДами

Представьте базу данных с репликами в нескольких ЦОДах (например, с целью обеспечить устойчивость к отказу целого ЦОДа или расположить реплики ближе к пользователям). При обычной схеме репликации с ведущим узлом последний должен быть в *одном* из ЦОДов и всем операциям записи следует проходить через этот ЦОД.

В схеме с несколькими ведущими узлами можно установить ведущий узел в *каждом* из ЦОДов. На рис. 5.6 показан пример такой архитектуры. Внутри каждого из ЦОДов используется обычная репликация типа «ведущий — ведомый»; между ЦОДами ведущий узел каждого ЦОДа реплицирует свои изменения ведущим узлам в других ЦОДах.

Сравним схему с одним и несколькими ведущими узлами в смысле развертывания в нескольких ЦОДах.

- *Производительность*. При схеме с одним ведущим узлом каждая операция записи должна проходить через Интернет в ЦОД, содержащий ведущий узел. Это может существенно задержать операцию записи и вообще пойдет вразрез с идеей нескольких ЦОДов. При схеме с несколькими ведущими узлами все операции записи будут обрабатываться в локальных ЦОДах и реплицироваться асинхронно в остальные ЦОДы. Таким образом, сетевые задержки между ЦОДами становятся незаметными для пользователей, а значит, субъективная производительность возрастет.
- *Устойчивость к перебоям в обслуживании ЦОДов*. При схеме с одним ведущим узлом в случае отказа ЦОДа, в котором находится ведущий узел, восстановление после сбоя сделает ведомый узел в другом ЦОДе ведущим. При схеме с несколькими ведущими узлами каждый ЦОД может работать независимо от остальных и репликация наверстывает упущенное после возобновления работы отказавшего ЦОДа.
- *Устойчивость к проблемам с сетью*. Трафик между ЦОДами обычно проходит через общедоступный Интернет — менее надежный, чем локальная сеть внутри ЦОДа. Схема с одним ведущим узлом очень чувствительна к сбоям работы этого соединения между ЦОДами, поскольку операции записи выполняются через него синхронно. Схема асинхронной репликации с несколькими ведущими узлами обычно более устойчива к проблемам с сетью: временные сбои работы сети не мешают обработке операций записи.

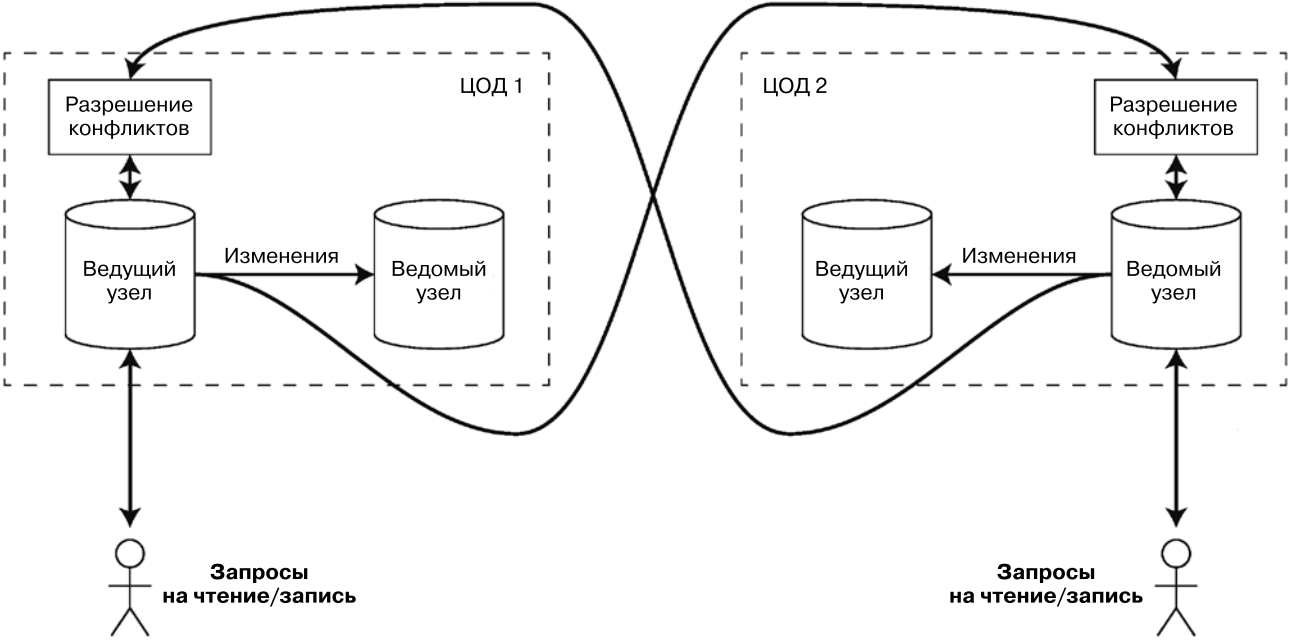


Рис. 5.6. Репликация с несколькими ведущими узлами и несколькими ЦОДами

Некоторые базы данных поддерживают схему с несколькими ведущими узлами по умолчанию, но она также часто реализуется с помощью внешних утилит, например Tungsten Replicator для СУБД MySQL [26], BDR для СУБД PostgreSQL [27] и GoldenGate для СУБД Oracle [19].

У репликации с несколькими ведущими узлами есть свои преимущества. Однако она имеет и серьезный недостаток: одни и те же данные могут одновременно модифицироваться в двух различных ЦОДах, и такие конфликты записи необходимо разрешать (на рис. 5.6 показано в виде прямоугольника с надписью «Разрешение конфликтов»). Мы обсудим этот вопрос в подразделе «Обработка конфликтов записи» текущего раздела.

Поскольку репликация с несколькими ведущими узлами — дополнительно доработанная возможность во многих базах данных, есть немало коварных подводных камней настроек и неожиданных взаимодействий с другими возможностями этих баз. Например, проблематичным может оказаться использование автоматически увеличиваемых ключей, триггеров и ограничений целостности. Поэтому репликация с несколькими ведущими узлами часто считается опасной вещью, которой лучше избегать по мере возможности [28].

Офлайн-клиенты

Другая ситуация, подходящая для использования репликации с несколькими ведущими узлами, — приложения, которые должны продолжать работать, даже будучи отключенными от Интернета.

Например, рассмотрим приложения-календари на мобильном телефоне, ноутбуке и других устройствах. У вас должна сохраняться возможность в любой момент просмотреть расписание ваших совещаний (выполнять запросы на чтение) и вводить в него новые совещания (выполнять запросы на запись) независимо от наличия соединения с Интернетом. Любые выполняемые в офлайн-режиме изменения должны синхронизироваться с сервером и остальными вашими устройствами при следующем подключении к Интернету.

В этом случае у каждого устройства есть своя локальная база данных, служащая ведущим узлом (она принимает запросы на запись). Кроме того, происходит асинхронный процесс репликации (синхронизации) между репликами календаря на всех устройствах. Задержка репликации может составлять часы или даже дни в зависимости от того, когда появится доступ в Интернет.

С точки зрения архитектуры такая схема, по сути, то же самое, что и репликация с несколькими ведущими узлами между ЦОДами, доведенная до крайности: каждое устройство представляет собой ЦОД, а сетевое соединение между ними исключительно ненадежно. Как показывает богатая история неудачных реализаций

синхронизации календарей, реализовать репликацию с несколькими ведущими узлами правильно очень непросто.

Существуют инструменты, призванные упростить реализацию подобных схем репликации с несколькими ведущими узлами. Например, специально для подобного режима эксплуатации создана СУБД CouchDB [29].

Совместное редактирование

Приложения для *совместного редактирования в режиме реального времени* (real-time collaborative editing) предоставляют возможность нескольким людям редактировать документ одновременно. Например, Etherpad [30] и Google Docs [31] позволяют одновременно редактировать документ или электронную таблицу (соответствующий алгоритм вкратце обсуждается во врезке «Автоматическое разрешение конфликтов» (см. пункт «Пользовательская логика разрешения конфликтов» подраздела «Обработка конфликтов записи» текущего раздела).

Обычно совместное редактирование не рассматривают как задачу репликации базы данных, но с ранее упомянутым офлайн-редактированием у него много общего. Когда пользователь редактирует документ, изменения немедленно применяются к его локальной реплике (состоянию документа в его браузере или клиентском приложении) и асинхронно реплицируются на сервер и на всех остальных пользователей, редактирующих этот же документ.

При необходимости обеспечить отсутствие конфликтов редактирования приложение должно, прежде чем пользователь сможет отредактировать документ, запросить на этот документ блокировку. Если другой пользователь хочет отредактировать тот же документ, он должен сначала подождать, пока первый не зафиксирует свои изменения и не снимет блокировку. Такая модель совместной работы эквивалентна репликации с одним ведущим узлом и выполнением транзакций на ведущем узле.

Однако для ускорения совместной работы желательно сделать блок изменений очень маленьким (например, одно нажатие клавиши) и по возможности избегать блокировок. Такой подход позволяет редактировать нескольким пользователям одновременно, но также влечет за собой все проблемы репликации с несколькими ведущими узлами, включая необходимость разрешения конфликтов [32].

Обработка конфликтов записи

Основная проблема репликации с несколькими ведущими узлами — возможность возникновения конфликтов записи, которые требуют разрешения.

Например, рассмотрим страницу «Википедии», одновременно редактируемую двумя пользователями, как показано на рис. 5.7. Пользователь 1 меняет название страницы

с A на B, а пользователь 2 одномоментно меняет это же название с A на C. Внесенные каждым из них изменения успешно применяются к локальному ведущему узлу этого пользователя. Однако при асинхронной репликации изменений возникает конфликт [33]. Такой проблемы не возникло бы при репликации с одним ведущим узлом.

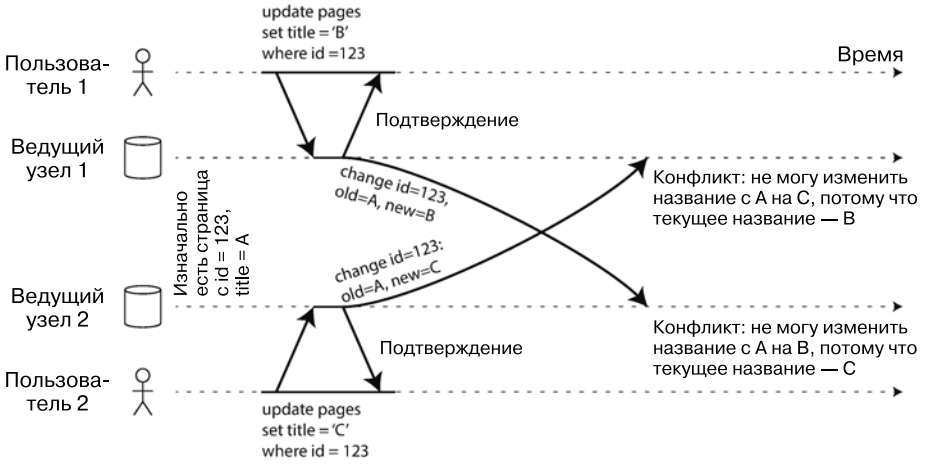


Рис. 5.7. Конфликт записи, вызываемый одновременным обновлением записи двумя ведущими узлами

Асинхронное и синхронное обнаружение конфликтов

В базе данных с одним ведущим узлом вторая операция записи будет или заблокирована и ей придется ждать завершения первой, или преждевременно прекращена, так что пользователю нужно ее повторить. С другой стороны, в схеме с несколькими ведущими узлами обе операции записи завершаются успешно, а конфликт обнаруживается асинхронно в какой-то более поздний момент времени. В этот момент, вероятно, уже слишком поздно просить пользователя разрешить его.

В принципе, можно сделать обнаружение конфликтов синхронным, то есть ждать репликации операции записи на все реплики, прежде чем сообщать пользователю об ее успешном завершении. Однако при этом потеряется главное преимущество репликации с несколькими ведущими узлами: возможность независимой обработки операций записи каждой репликой. Вместо синхронного обнаружения конфликтов можно просто применить репликацию с одним ведущим узлом.

Предотвращение конфликтов

Простейшая стратегия для конфликтов — просто избегать их: если приложение способно гарантировать, что все операции изменения конкретной записи проходят через один и тот же ведущий узел, то конфликт просто невозможен. Поскольку многие реализации репликации с несколькими ведущими узлами не очень хорошо разрешают конфликты, зачастую предотвращение конфликтов — рекомендуемый подход [34].

Например, если в приложении пользователь может редактировать собственные данные, то необходимо обеспечить, чтобы запросы от конкретного человека всегда маршрутизировались на один и тот же ЦОД и применяли ведущий узел последнего для чтения и записи. У разных людей могут оказаться разные «родные» ЦОДы (например, подобранные исходя из географической близости к пользователю), но с точки зрения любого отдельного пользователя это фактически конфигурация с одним ведущим узлом.

Однако иногда может понадобиться сменить назначенный для записи ведущий узел, например, когда произошел сбой одного из ЦОДов и необходимо направить трафик на другой центр или пользователь переехал в другое место и теперь находится ближе к другому ЦОД. В этой ситуации предотвращение конфликтов не работает и приходится как-то справляться с возможностью конкурентных операций записи на различных ведущих узлах.

Сходимость к согласованному состоянию

База данных с одним ведущим узлом применяет операции записи в последовательном порядке: при наличии нескольких изменений одного поля последняя по времени операция записи определяет итоговое значение этого поля.

В схеме с несколькими ведущими узлами нельзя задать порядок операций записи, так что непонятно, каким должно быть итоговое значение. На рис. 5.7 в ведущем узле 1 название сначала меняется на B, а позднее на C, в ведущем узле 2 оно сначала меняется на C, а потом — на B. Никакой из этих порядков нельзя назвать «более правильным, чем другой».

Если все реплики просто будут применять операции записи в том порядке, в котором они эти операции записи получают, то база данных в конце концов окажется в несогласованном состоянии: итоговое значение будет C в ведущем узле 1 и B — в ведущем узле 2. Это недопустимо: каждая схема репликации обязана обеспечить наличие одинаковых данных во всех репликах. Следовательно, база должна разрешать конфликт конвергентным способом, то есть все реплики должны сойтись к одному значению после репликации всех изменений.

Существует множество способов конвергентного разрешения конфликтов.

- Присвоить каждой операции записи уникальный идентификатор (например, метку даты/времени, случайное длинное число, UUID или хеш ключа и значения), после чего просто выбрать операцию *(«победителя»)* с максимальным значением этого идентификатора, а остальные отбросить. В случае использования метки даты/времени в качестве идентификатора этот метод известен под названием *«выигрывает последний»* (last write wins, LWW). Хотя этот подход очень популярен, ему свойственно терять данные [35]. Мы обсудим LWW подробнее в конце этой главы (в подразделе «Обнаружение конкурентных операций записи» раздела 5.4).
- Присвоить уникальный идентификатор каждой реплике и считать, что у исходящих от реплик с большим номером операций записи есть приоритет перед теми, которые исходят от реплик с меньшим. Этот подход также приводит к потерям данных.
- Каким-либо образом слить значения воедино, например, выстроить их в алфавитном порядке, после чего выполнить их конкатенацию (на рис. 5.7 такое объединенное название могло бы выглядеть как B/C).
- Заносить конфликты в заданную в явном виде структуру данных для хранения и написать код приложения, который бы разрешал конфликты позднее (возможно, спрашивая для этого пользователя).

Пользовательская логика разрешения конфликтов

Поскольку то, как лучше всего разрешить конфликт, зависит от приложения, большинство программных средств репликации с несколькими ведущими узлами позволяют создать логику разрешения конфликтов в коде приложения. Этот код может выполняться при записи или чтении.

- *При записи.* При обнаружении конфликта в журнале реплицированных изменений СУБД вызывает обработчик конфликтов. Например, Bucardo позволяет писать с этой целью сниппеты на языке программирования Perl. Подобный обработчик обычно не может спросить что-либо у пользователя — он запускается в фоновом процессе и должен выполняться быстро.
- *При чтении.* При обнаружении конфликта система сохраняет информацию обо всех конфликтующих операциях записи. При следующей операции чтения все эти различные версии данных возвращаются приложению. Далее оно может спросить пользователя или разрешить конфликт автоматически, после чего записать результаты обратно в БД. Подобным образом работает, например, CouchDB.

Обратите внимание: подобное разрешение конфликтов выполняется обычно на уровне отдельных строк или документов, а не транзакций в целом [36]. Следовательно, в случае транзакции, выполняющей атомарно несколько различных операций записи (см. главу 7), каждая операция рассматривается отдельно для целей разрешения конфликтов.

Автоматическое разрешение конфликтов

Правила разрешения конфликтов быстро усложняются, и пользовательский код становится подвержен ошибкам. В качестве примера неожиданных эффектов вследствие разрешения конфликтов часто упоминается Amazon: логика разрешения конфликтов корзины заказов сохраняет в течение некоторого времени добавленные в корзину товары, но не удаляемые из нее. Следовательно, покупатели иногда видят повторное появление в корзине заказов ранее удаленных из нее товаров [37].

Существуют интересные исследования на тему автоматического разрешения конфликтов, вызванных конкурентными изменениями данных. Кое-что из них достойно упоминания.

- *Бесконфликтные реплицируемые типы данных* (conflict-free replicated datatype, CRDT) [32, 38] — семейство структур данных для множеств, ассоциативных словарей, упорядоченных списков, счетчиков и др., допускающие конкурентное редактирование несколькими пользователями и автоматически разрешающие конфликты разумным образом. Некоторые из CRDT были реализованы в Riak 2.0 [39, 40].
- *Объединяемые хранимые структуры данных* (mergeable persistent data structures) [41] обеспечивают отслеживание истории явным образом аналогично системе контроля версий Git и используют трехстороннюю функцию слияния (в отличие от двухстороннего слияния в CRDT).
- *Операциональное преобразование* (operational transformation) [42] представляет собой алгоритм разрешения конфликтов, используемый в таких приложениях для совместного редактирования, как Etherpad [30] и Google Docs [31]. Он был спроектирован специально для совместного редактирования упорядоченного списка элементов, например перечня букв, составляющего текстовый документ.

Автоматическое разрешение конфликтов значительно упрощает для приложений применение синхронизации данных с несколькими ведущими узлами. Реализации этих алгоритмов в БД все еще довольно незрелы, но есть вероятность, что в будущем они будут включаться во все большее количество реплицируемых информационных систем.

Что такое конфликт

Некоторые разновидности конфликтов очевидны. В примере на рис. 5.7 две операции записи одновременно меняют значение одного поля одной записи различным образом. Совершенно очевидно, что это конфликт.

Распознать другие виды конфликтов иногда труднее. Например, рассмотрим систему бронирования конференц-залов: она отслеживает то, на какое время и для какой группы людей забронирован некий зал. Это приложение должно гарантировать, что каждое помещение забронировано на энное время только одной группой людей (то есть не должно быть перекрывающихся бронирований одного конференц-зала). В этом случае может возникнуть конфликт при создании двух различных бронирований одного зала на одно время. Даже если приложение проверяет доступность помещения, прежде чем разрешить пользователю операцию бронирования,

конфликт возможен в случае выполнения двух бронирований на двух различных ведущих узлах.

Быстрого шаблонного решения этой проблемы не существует, но в последующих главах мы наметим путь для него. Мы рассмотрим еще несколько примеров конфликтов в главе 7, а в главе 12 обсудим масштабируемые подходы для обнаружения и разрешения конфликтов в реплицируемой системе.

Топологии репликации с несколькими ведущими узлами

Топология репликации (replication topology) описывает пути, по которым операции записи распространяются с одного узла на другой. В случае двух ведущих узлов, как на рис. 5.7, существует только одна разумная топология: ведущий узел 1 должен отправлять информацию обо всех своих операциях записи ведущему узлу 2 и наоборот. В случае более чем двух ведущих узлов возможны различные варианты топологии. На рис. 5.8 показан ряд примеров.

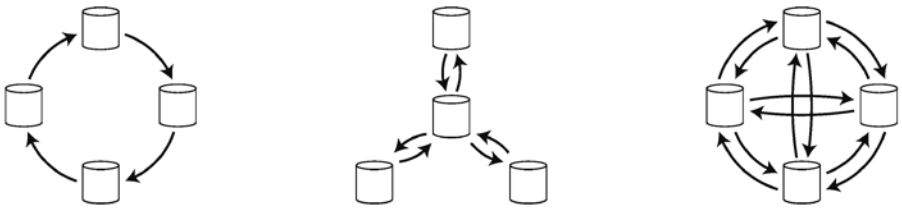


Рис. 5.8. Три примера возможных топологий для репликации с несколькими ведущими узлами

Наиболее общий вариант топологии — *каждый с каждым* (см. рис. 5.8.В), при которой каждый ведущий узел отправляет информацию об операциях записи всем остальным таким узлам. Однако используются и более ограниченные варианты: например, MySQL по умолчанию поддерживает только *топологию типа «кольцо»* [34], при которой каждый узел получает информацию об операциях записи от ровно одного узла и передает ее (плюс информацию о своих собственных операциях записи) ровно одному из других узлов. Еще одна популярная топология имеет форму *звезды*[1]: один узел, назначенный корневым, пересылает информацию об операциях записи всем остальным узлам. Топологию типа «звезда» можно обобщить до дерева.

В топологиях типа «кольцо» и «звезда» операция записи должна пройти через несколько узлов, прежде чем попадет во все реплики. Следовательно, узлам нужно

[1] Не следует путать ее со схемой «звезда» (см. подраздел «“Звезды” и “снежинки”»: схемы для аналитики» раздела 3.2), которая описывает структуру модели данных, а не топологию взаимодействия между узлами.

пересылать получаемые ими от других узлов изменения данных. Во избежание бесконечных циклов репликации всем узлам присваиваются уникальные идентификаторы, а каждая операция записи в журнале репликации помечается идентификаторами всех узлов, через которые она прошла [43]. Узел при получении игнорирует изменения данных, помеченные его собственным идентификатором, поскольку знает, что они уже были обработаны.

Проблема топологий типа «кольцо» и «звезда» заключается в следующем: отказ даже одного узла способен прервать поток сообщений репликации между другими узлами, делая связь между ними невозможной вплоть до момента его восстановления. Топологию можно перестроить, чтобы обойти отказавший узел, но в большинстве случаев такую перестройку придется делать вручную. Отказоустойчивость более тесно связанных топологий (например, типа «каждый с каждым») выше, поскольку они позволяют сообщениям перемещаться различными путями, обходя отказавшие узлы.

С другой стороны, у топологий типа «каждый с каждым» тоже есть свои проблемы. В частности, одни сетевые ссылки могут оказаться быстрее других (например, из-за перегруженности сети), в результате чего одни сообщения репликации могут обгонять другие, как показано на рис. 5.9.

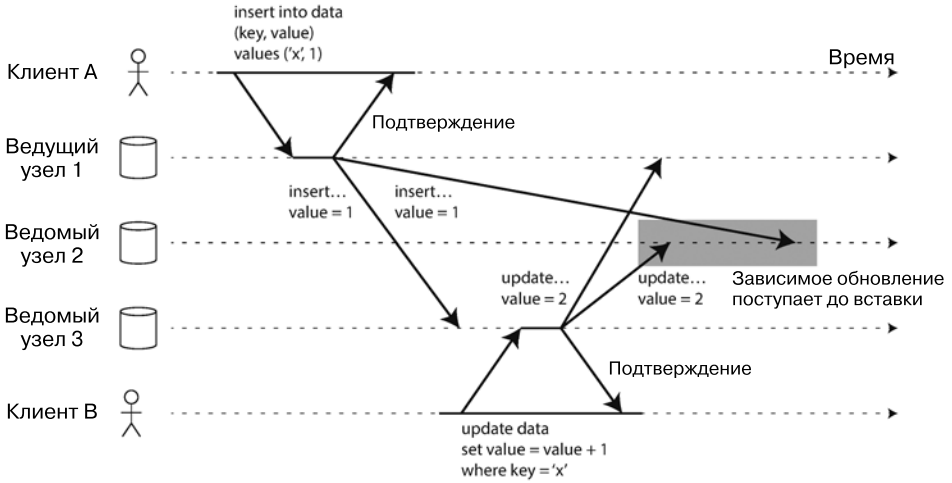


Рис. 5.9. При репликации с несколькими ведущими узлами информация об операциях записи может поступать в некоторые реплики в неправильном порядке

На рис. 5.9 клиент A вставляет строку в таблицу на ведущем узле 1, а клиент B обновляет данную строку на ведущем узле 3. Однако может получиться, что ведущий узел 2 получает информацию об этих операциях записи в другом порядке: сначала получает обновление (и оно с его точки зрения представляет собой обновление отсутствующей в БД строки) и только потом — информацию о соответствующей операции вставки (которая должна предшествовать обновлению).

Эта проблема касается нарушения причинно-следственной связи и аналогична описанной в подразделе «Согласованное префиксное чтение» раздела 5.2: обновление зависит от предыдущей вставки, вследствие чего нужно гарантировать, что все узлы сначала выполняют вставку, а потом обновление. Простого добавления ко всем операциям записи метки даты/времени недостаточно — нет уверенности в синхронизированности часов, достаточной для правильной упорядоченности этих событий в ведущем узле 2 (см. главу 8).

Чтобы правильно упорядочить эти события, используется метод векторов версий, который мы обсудим ниже (см. подраздел «Обнаружение конкурентных операций записи» раздела 5.4). Однако поддержка методов обнаружения конфликтов совершенно недостаточна во многих системах репликации с несколькими ведущими узлами. Например, на момент написания этой книги система BDR базы данных PostgreSQL не поддерживает причинно-следственную упорядоченность операций записи [27], а Tungsten Replicator для СУБД MySQL даже не пытается обнаруживать конфликты [34].

При использовании системы с репликацией с несколькими ведущими узлами не помешает знать о подобных проблемах, внимательно изучить документацию и тщательно тестировать свою базу данных, чтобы убедиться в ее возможностях действительно гарантировать требуемое вами.

5.4. Репликация без ведущего узла

Обсуждавшиеся до сих пор подходы к репликации — с одним ведущим узлом и с несколькими — основаны на идее, что клиент отправляет запрос на запись одному узлу (ведущему), а СУБД берет на себя дублирование этой операции на всех остальных репликах. Ведущий узел определяет порядок обработки операций записи, а ведомые применяют полученные от ведущего операции записи в том же порядке.

Некоторые системы хранения данных используют другой подход, отказываясь от концепции ведущего узла и позволяя непосредственное поступление информации об операциях записи на все реплики. В ряде ранних реплицируемых информационных систем не было ведущего узла [1, 44], но за время доминирования реляционных баз данных эта идея была практически забыта. Такая архитектура снова вошла в моду после того, как Amazon задействовал ее для своей предназначенной для внутреннего использования системы *Dynamo* [37][1]. Riak, Cassandra и Voldemort представляют собой вдохновленные Dynamo склады данных с открытым исходным кодом, применяющие модели репликации без ведущего узла. Поэтому подобный тип БД называют *Dynamo-подобной базой данных* (Dynamo-style database).

---

[1] Dynamo недоступна пользователям вне Amazon. AWS предлагает пользователям хостируемую СУБД DynamoDB, в которой применяется совершенно другая архитектура: она основана на репликации с одним ведущим узлом.

В некоторых реализациях репликации без ведущего узла клиент непосредственно отправляет информацию о своих операциях записи на несколько реплик, в то время как для остальных данную манипуляцию от имени клиента совершает узел-координатор. Однако он, в отличие от БД с ведущим узлом, не навязывает определенный порядок операций записи. Как мы увидим в дальнейшем, это отличие в архитектуре приводит к очень серьезным последствиям в смысле стиля использования базы данных.

Запись в базу данных при отказе одного из узлов

Представьте, что у вас есть БД с тремя репликами, одна из которых в настоящий момент недоступна — допустим, она перезагружается для установки обновления системы. В случае схемы репликации с ведущим узлом при необходимости продолжать обработку операций записи придется выполнить восстановление после отказа (см. подраздел «Перебои в обслуживании узлов» раздела 5.1).

С другой стороны, в схеме репликации без ведущего узла восстановления после отказа не существует. Рисунок 5.10 демонстрирует происходящее в этом случае: клиент (пользователь 1234) параллельно отправляет информацию об операции записи всем трем репликам и две доступные реплики принимают ее, а недоступная — нет. Допустим, вполне достаточно, что две из трех реплик подтвердили получение операции записи: после того, как пользователь 1234 получит два ответа с подтверждением, можно считать операцию записи успешной. Клиент просто игнорирует тот факт, что одна из реплик не получила информацию.

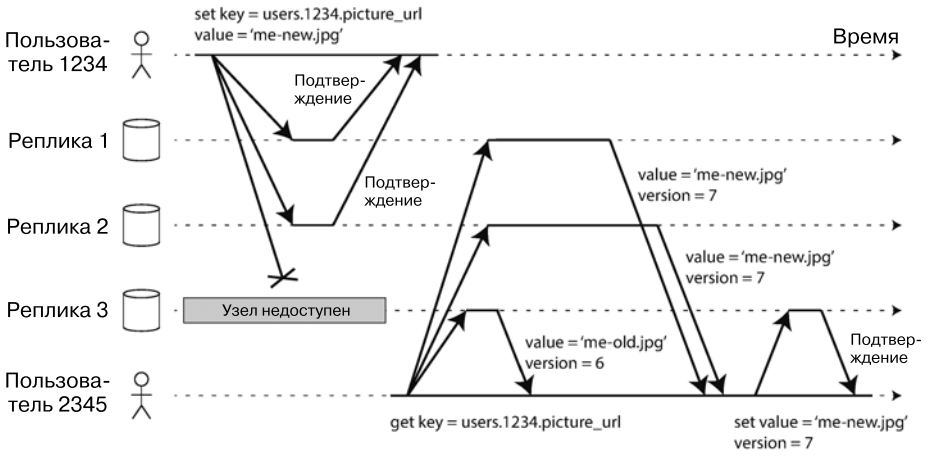


Рис. 5.10. Операция записи по кворуму, операция чтения по кворуму и разрешение конфликтов при чтении после перебоя в обслуживании узла

Теперь представьте, что бывший недоступным узел снова работает и клиенты начинают читать из него данные. В нем отсутствует информация обо всех операциях записи, выполнявшихся, когда он не функционировал. Следовательно, при чтении из него вы рискуете получить в ответ *устаревшие* значения.

Для решения указанной проблемы при чтении из базы данных клиенты отправляют запросы не к одной реплике, а сразу к *нескольким узлам параллельно*. При этом клиент может получить различные ответы от разных узлов, то есть актуальные значения от одного узла и устаревшие от другого. Чтобы определить, какое значение новее, используются номера версий (см. подраздел «Обнаружение конкурентных операций записи» текущего раздела).

Разрешение конфликтов при чтении и антиэнтропия

Схема репликации должна обеспечивать копирование в конечном итоге всех данных в каждую из реплик. Каким образом ранее недоступный узел после возобновления работы может наверстать пропущенные им операции записи?

В Dynamo-подобных складах данных часто используются два механизма.

- *Разрешение конфликтов при чтении*. Клиент, читающий данные с нескольких узлов параллельно, способен обнаружить все устаревшие ответы. Например, на рис. 5.10 пользователь 2345 получает значение с версией 6 от реплики 3 и значение с версией 7 от реплик 1 и 2. Клиент понимает, что значение реплики 3 устарело, и записывает в нее более новое значение.
- *Процесс противодействия энтропии*. Кроме того, в некоторых складах есть фоновый процесс, постоянно выискивающий различия в данных между репликами и копирующий отсутствующие данные из одной реплики в другую. В отличие от журнала репликации при репликации с ведущим узлом, подобный *процесс противодействия энтропии* (anti-entropy process) не копирует информацию об операциях записи в каком-либо определенном порядке, а перед копированием данных может возникнуть значительная задержка.

Не во всех системах реализованы оба этих механизма: например, в СУБД Voldemort в настоящий момент отсутствует процесс противодействия энтропии. Обратите внимание: без этого процесса редко читаемые значения могут отсутствовать в некоторых репликах, что приводит к снижению сохраняемости данных, поскольку разрешение конфликтов при чтении выполняется только при чтении значения приложением.

Операции записи и чтения по кворуму

В примере на рис. 5.10 мы предполагали, что операция записи успешна, несмотря на ее обработку только двумя репликами из трех. А если бы только одна реплика из трех получила данные об этой операции записи? Насколько далеко мы можем здесь зайти?

Если бы была уверенность, что все успешные операции записи попадут как минимум на две реплики из трех, то устаревшей оказалась бы максимум одна реплика. Следовательно, при чтении данных из двух реплик можно быть уверенными: хотя бы одна из двух содержит актуальные данные. Если третья реплика не функционирует или отвечает слишком медленно, то операции чтения все равно будут продолжать возвращать актуальные значения.

Говоря более общим языком, при наличии n реплик операция записи, чтобы считаться успешной, должна быть подтверждена w узлами, причем мы должны опросить как минимум r узлов для каждой операции чтения (в нашем примере $n = 3$, $w = 2$, $r = 2$). Если $w + r > n$, то можно ожидать: полученное при чтении значение будет актуальным, поскольку хотя бы один из r узлов, из которых мы читаем, должен оказаться актуальным. Операции записи и чтения, удовлетворяющие этому соотношению значений r и w, называются операциями чтения и записи *по кворуму* [44][1]. Можно рассматривать r и w как минимальные количества «голосов», необходимых для признания операции чтения или записи приемлемой.

В Dynamo-подобных базах данных параметры n, w и r обычно можно настраивать. Чаще всего n делают равным нечетному числу (обычно 3 или 5), а $w = r = (n + 1) / 2$ (округленному в бóльшую сторону). Однако вам под силу менять эти числа так, как покажется лучше. Например, при нагрузке, состоящей из нескольких операций записи и большого количества операций чтения, может быть выгодно задать $w = n$ и $r = 1$. Такие показатели ускорят операции чтения, но всего лишь один отказавший узел приведет к сбою всей базы данных.



В кластере может быть более n узлов, но любое заданное значение будет сохраняться только на n узлах. Благодаря этому становится возможно секционировать набор данных и поддерживать бóльшие наборы, чем те, что помещаются в одном узле. Мы вернемся к вопросу секционирования в главе 6.

Условие кворума, $w + r > n$, придает системе устойчивость к недоступности узлов:

- при $w < n$ можно продолжать обрабатывать информацию об операциях записи в случае недоступности узла;
- при $r < n$ можно продолжать обрабатывать информацию об операциях чтения в случае недоступности узла;
- при $n = 5$, $w = 3$, $r = 3$ система может позволить себе два недоступных узла. Этот случай показан на рис. 5.11;

[1] Иногда подобный кворум называется строгим (strict quorum) в отличие от нестрогого кворума (sloppy quorum), который мы обсудим в подразделе «Нестрогие кворумы и направленная передача» раздела 5.4.

- обычно информация об операциях чтения и записи отправляется параллельно всем *n* репликам. Параметры *w* и *r* определяют, подтверждения от какого количества узлов мы будем ждать: то есть сколько узлов из *n* должны подтвердить успех, прежде чем можно будет считать операцию чтения или записи успешной в целом.

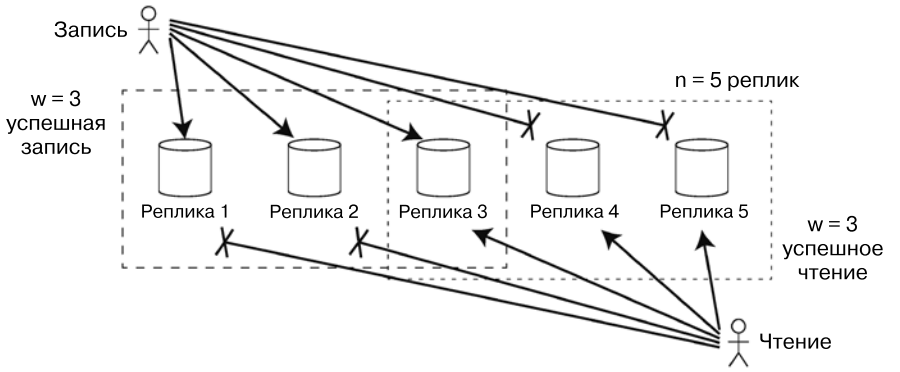


Рис. 5.11. При w + r > n как минимум одна из читаемых r реплик точно получила информацию о самой последней успешной операции записи

Если доступно меньшее количество узлов, чем требуемое *w* или *r*, то операции записи или чтения вернут ошибку. Узел может быть недоступен по множеству причин: из-за того что отключен (произошел фатальный сбой, или отключено питание), вследствие ошибки при выполнении операции (например, нельзя записать данные из-за переполнения диска), в связи с разрывом сетевого соединения между клиентом и узлом или по любой из многих других причин. Нас интересует только, вернул ли узел ответ об успешном выполнении, нет нужды дифференцировать различные виды отказов.

Ограничения согласованности по кворуму

При наличии *n* реплик и выборе таких *w* и *r*, что $w + r > n$, можно, вообще говоря, ожидать возврата каждой операцией чтения наиболее свежего значения из записанных для данного ключа. Дело вот в чем: множество узлов, в которые вы записали данные, и множество узлов, из которых вы данные читаете, должны пересекаться. То есть среди читаемых узлов должен быть по крайней мере один узел с актуальным значением (это проиллюстрировано на рис. 5.11).

Зачастую значения *w* и *r* выбираются таким образом, чтобы составлять большинство (более чем *n/2*) узлов, поскольку это гарантирует, что система может позволить себе вплоть до *n/2* сбойных узлов с соблюдением $w + r > n$. Но кворум не обязательно означает большинство — важен только аспект пересечения хотя бы в одном узле множества узлов, используемых операциями чтения и записи. Воз-

можны и другие значения кворума; это обеспечивает некоторую гибкость при проектировании распределенных алгоритмов [45].

Можно также задать меньшие значения w и r так, что $w + r \leq n$ (то есть условие кворума не удовлетворяется). В этом случае информация об операциях чтения и записи будет по-прежнему отправляться на n узлов, но для успеха операции необходимо меньшее количество подтверждений успеха.

При меньших значениях w и r вероятность прочитать устаревшие значения возрастает, поскольку повышается вероятность того, что при чтении не будет охвачен узел с актуальным значением. Положительный момент состоит в следующем: эта схема обещает более короткую задержку и более высокую доступность: при наличии сбоев сети и недоступности многих реплик шансы на дальнейшую обработку операций записи и чтения повышаются. Недоступной для записи или чтения база данных станет, только когда количество доступных реплик окажется ниже w и r соответственно.

Однако даже при $w + r > n$ возможны граничные случаи, при которых будут возвращены устаревшие значения. Это зависит от реализации, но вероятные сценарии включают следующее.

- При использовании нестрогого кворума (см. подраздел «Нестрогие кворумы и направленная передача» текущего раздела) w операций записи могут прийтись не на те узлы, что r операций чтения, поэтому гарантированного пересечения между множествами r узлов и w узлов нет [46].
- При конкурентном (параллельном) выполнении двух операций записи непонятно, какая из них произошла первой. В этом случае единственным «безопасным» решением будет слить воедино конкурентные операции (см. подраздел «Обработка конфликтов записи» раздела 5.3). Если «победитель» выбирается на основе метки даты/времени («выигрывает последний»), существует вероятность потери информации об операциях записи вследствие рассинхронизации часов [35]. Мы вернемся к этому вопросу в подразделе «Обнаружение конкурентных операций записи» текущего раздела.
- Операция записи, выполняемая параллельно с чтением, может оказаться отраженной только в некоторых репликах. В этом случае неясно, вернет операция чтения новое или старое значение.
- Если операция записи прошла успешно в ряде реплик и неудачно в остальных (например, вследствие переполненности диска в некоторых узлах), причем суммарно прошла успешно менее чем в w репликах, она не откатывается в последних. Это значит, что в случае уведомления о неудачной операции записи последующие операции чтения могут или вернуть соответствующее ей значение, или нет [47].
- В случае сбоя содержащего новое значение узла и его восстановления из содержащей старые данные реплики количество хранящих новое значение реплик может оказаться ниже w, нарушая тем самым условие кворума.

- Даже если все функционирует нормально, существуют граничные случаи с неудачным хронометражем, как мы увидим в пункте «Линеаризуемость и кворумы» подраздела «Реализация линеаризуемых систем» раздела 9.2.

Следовательно, хотя кворум вроде бы гарантирует чтение последнего из записанных значений, на практике все не так просто. Dynamo-подобные базы данных в целом оптимизированы для сценариев использования, допускающих конечную согласованность. Параметры w и r предоставляют возможность влиять на вероятность чтения устаревших значений, но воспринимать их как абсолютные гарантии было бы неразумно.

В частности, обсуждавшиеся ранее в разделе 5.2 гарантии обычно недоступны (чтение своих записей, монотонное чтение и согласованное префиксное чтение), так что в приложениях могут возникать ранее упомянутые аномалии. Для более надежных гарантий обычно необходимы транзакции или консенсус. Мы вернемся к этим вопросам в главах 7 и 9.

Мониторинг устарелости данных. С точки зрения эксплуатации важно контролировать, возвращают ли базы данных актуальные результаты. Даже если приложение допускает чтение устаревших данных, необходимо понимать состояние репликации. Если она существенно отстает, система должна оповестить вас, чтобы вы могли выяснить причину (ею может быть, например, проблема с сетью или перегруженный узел).

В случае репликации с ведущим узлом база данных обычно предоставляет метрики для задержки репликации, которые можно передать системе мониторинга. Это становится возможным благодаря применению операций записи к ведущему и ведомым узлам в одном порядке, причем у каждого узла есть своя позиция в журнале репликации (количество примененных локально операций записи). Вычитая текущую позицию ведомого узла из текущей позиции ведущего узла, можно оценить степень запаздывания репликации.

Однако в системах репликации без ведущего узла не существует фиксированного порядка применения операций записи, что осложняет мониторинг. Более того, если база данных использует только разрешение конфликтов при чтении (никакого процесса противодействия энтропии), то не существует оценки устарелости значения — когда значение читается лишь изредка, возвращаемое устаревшей репликой значение может оказаться очень старым.

Были проведены исследования по измерению степени устарелости реплик в базах данных с репликацией без ведущего узла и предсказанию процентного отношения чтений операций устаревших данных к операциям чтения актуальных в зависимости от параметров n, w и r [48]. К сожалению, пока что в стандартный набор метрик для БД редко включают оценку устарелости, хотя это было бы уместно. Конечная согласованность — нарочно расплывчатая гарантия, но для удобства эксплуатации важна возможность количественной оценки характеристики «конечная».

Нестрогие кворумы и направленная передача

Базы данных с соответствующим образом заданным кворумом могут выдержать отказы отдельных узлов без необходимости восстановления. Могут они выдержать и замедление работы отдельных узлов, поскольку запросам не нужно ждать ответа от всех n узлов — достаточно, чтобы ответили w или r узлов. Эти характеристики делают БД с репликацией без ведущего узла подходящими для сценариев использования, требующих высокой доступности и низкого значения задержки, а также возможности выдержать происходящие иногда операции чтения устаревших данных.

Однако кворум (как описывалось выше) — не столь отказоустойчивый метод, как хотелось бы. Сетевые сбои с легкостью могут прервать связь клиента с большим количеством узлов базы данных. Хотя эти узлы продолжают функционировать и другие клиенты могут к ним подключиться, для отрезанного от них клиента они все равно что не работают. В такой ситуации вполне может остаться менее чем w или r доступных узлов, поэтому клиент более не сможет достичь кворума.

В большом кластере (со значительно превышающим n числом узлов) вполне вероятно следующее: клиент сможет подключиться к *некоторым* узлам базы данных во время сбоя сети, но не к нужным ему, чтобы достичь кворума для конкретного значения. В этом случае у создателей базы есть такие альтернативы.

- Будет ли лучше вернуть ошибки в качестве ответа на все запросы, для которых не получается достичь кворума из w или r узлов?
- Или следует обработать операции записи в любом случае и применить их к ряду узлов, доступных, но не входящих в число n узлов, в которых обычно размещается это значение?

Второй вариант известен под названием *нестрогого кворума* (sloppy quorum) [37]: для операций записи и чтения по-прежнему необходимо w и r подтверждений успешного выполнения, но при этом учитываются узлы, не входящие в число намеченных для значения n «родных» узлов. Здесь будет уместна такая аналогия: если вы потеряли ключи от квартиры, то можете постучать к соседям и попроситься переночевать у них на диване.

После исправления сбоя сети все операции записи, временно отправленные в какой-либо узел вместо недоступного, отправляются в соответствующие «родные» узлы. Это называется *направленной передачей* (hinted handoff). Аналогично, когда вы снова найдете ключи от квартиры, сосед вежливо попросит вас освободить диван и отправиться домой.

Нестрогие кворумы особенно полезны в деле повышения доступности для записи: база данных может принимать операции записи до тех пор, пока доступны *любые* w узлов. Однако это значит, что даже при $w + r > n$ нельзя гарантировать чтение актуального значения для ключа, поскольку актуальное значение может быть временно записано на какие-то узлы вне множества n [47].

Следовательно, нестрогий кворум на самом деле вовсе не кворум в обычном смысле слова. Это просто гарантия сохраняемости данных, а именно сохранения данных в каких-то w узлах. Нет гарантий, что при чтении из r узлов они окажутся видны до тех пор, пока не будет завершена направленная передача.

Нестрогие кворумы — дополнительная возможность во всех распространенных реализациях Dynamo. В Riak они активизированы по умолчанию, а в Cassandra и Voldemort — по умолчанию отключены [46, 49, 50].

Эксплуатация с несколькими ЦОДами. Мы уже обсуждали ранее репликацию между ЦОДами как сценарий использования репликации с несколькими ведущими узлами (см. раздел 5.3). Репликация без ведущего узла тоже подходит для эксплуатации с несколькими ЦОДами, поскольку спроектирована в расчете на конфликтующие конкурентные операции записи, разрывы сети и резкие скачки времени задержки.

СУБД Cassandra и Voldemort реализуют поддержку нескольких ЦОДов в рамках обычной модели репликации без ведущего узла: число реплик n включает узлы во всех ЦОДах, а в настройках можно задать, какая часть из n реплик должна находиться в каждом ЦОДе. Вся поступающая от клиента информация про операции записи отправляется на все реплики независимо от ЦОДа, но клиент обычно ждет подтверждения только от кворума узлов его локального ЦОДа, так что его не затрагивают задержки и сбои соединений между ЦОДами. Зачастую предполагающие бóльшие задержки операции записи, отправляемые в другие ЦОДы, делаются асинхронными, хотя возможны некоторые гибкие настройки [50, 51].

В Riak весь обмен сообщениями между клиентами и узлами базы данных происходит в пределах одного ЦОДа, так что n соответствует количеству реплик в одном ЦОДе. Репликация между ЦОДами, между кластерами БД происходит асинхронно в фоновом режиме и напоминает по стилю репликацию с несколькими ведущими узлами [52].

Обнаружение конкурентных операций записи

В Dynamo-подобных базах данных возможна конкурентная запись значения для одного ключа несколькими клиентами, что означает вероятность конфликтов даже при использовании строгих кворумов. Это аналогично ситуации при репликации с несколькими ведущими узлами (см. подраздел «Обработка конфликтов записи» раздела 5.3), хотя в Dynamo-подобных базах конфликты могут возникать и во время разрешения конфликтов при чтении или направленной передачи.

Проблема в том, что события могут поступать в разные узлы в разном порядке вследствие различных сетевых задержек и частичных отказов. Например, на рис. 5.12 показаны два клиента, одновременно записывающих значение для ключа X в складе данных из трех узлов:

- узел 1 получает информацию об операции записи от A, но не получает информацию об операции записи от B вследствие врéменного перебоя в обслуживании;

- узел 1 получает информацию об операции записи сначала от A, а затем от B;
- узел 1 получает информацию об операции записи сначала от B, а затем от A.

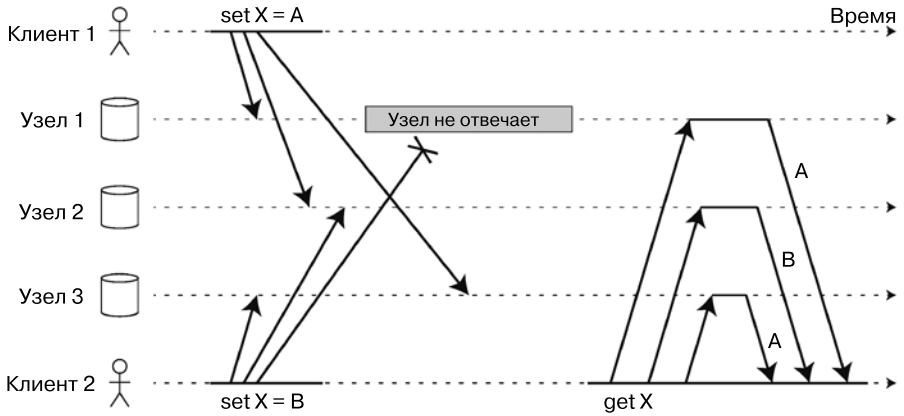


Рис. 5.12. Конкурентные операции записи в Dynamo-подобном складе данных: четко определенный порядок отсутствует

Если каждый из узлов будет просто переписывать значение для ключа при каждом получении запроса на запись от клиента, то узлы будут все время несогласованными, как видно из последнего запроса `get` на рис. 5.12: узел 2 считает, что итоговое значение X равно B, а остальные узлы — что A.

Для достижения конечной согласованности реплики должны стремиться к одному значению. Каким образом? Можно надеяться на автоматическое решение этой проблемы реплицируемыми базами данных, но, к сожалению, большинство реализаций не особо хороши: чтобы избежать потери данных, разработчику приложения необходимо хорошо разбираться в нюансах того, как используемая БД разрешает конфликты.

Мы вкратце затронули некоторые методы разрешения конфликтов в подразделе «Обработка конфликтов записи» раздела 5.3. Прежде чем завершить данную главу, рассмотрим этот вопрос несколько подробнее.

«Выигрывает последний» (отбраковка конкурентных операций записи)

Один из методов достижения в итоге конвергентности — провозгласить, что все реплики должны хранить только самое «свежее» значение, а более «старые» значения могут перезаписываться и игнорироваться. В этом случае все реплики будут стремиться к одному значению, если, конечно, у нас есть способ однозначного выяснения того, какая операция записи самая «свежая», и каждая операция записи в конце концов воспроизводится во всех репликах.

Как понятно из кавычек вокруг слова «свежий», эта идея довольно обманчива. В примере на рис. 5.12 ни один из клиентов не знает ничего о другом при отправке запросов на запись узлам базы данных, поэтому непонятно, что происходит первым. Фактически не имеет смысла говорить о «первенстве» какого-либо из них: говорится, что операции записи конкурентны, вследствие чего порядок их выполнения не определен.

Хотя операции записи не упорядочены сами по себе, мы можем навязать им произвольный порядок. Например, добавить в каждую операцию записи метку даты/времени, выбрать метку даты/времени с максимальным значением в качестве самой «свежей» и отбросить все операции записи с более ранними метками. Такой метод разрешения конфликтов, называемый *«выигрывает последний»* (last write wins, LWW), — единственный алгоритм разрешения конфликтов, поддерживаемый СУБД Cassandra [53], и дополнительная возможность в Riak [35].

LWW позволяет достичь в итоге конвергентности, но за счет сохраняемости данных: в случае нескольких конкурентных операций записи для одного и того же ключа, для которых клиент мог даже получить подтверждение их успешного выполнения (поскольку они записывались на w реплик), только одна из операций записи будет учтена, а остальные — незаметно проигнорированы. Более того, LWW может даже игнорировать неконкурентные операции записи, как мы обсудим в пункте «Метки даты/времени и упорядочение событий» подраздела «Ненадежность синхронизированных часов» раздела 8.3.

Существует ряд ситуаций, например кэширование, при которых, возможно, потеря части операций записи допустима. Если же потеря данных неприемлема, no LWW — не лучший способ разрешения конфликтов.

Единственный безопасный способ использования базы данных с LWW — позаботиться, чтобы ключ записывался лишь один раз, после чего рассматривался как неизменяемый, благодаря чему можно избежать любых конкурентных обновлений одного ключа. Например, рекомендуемый способ применения Cassandra — задействовать в качестве ключа UUID, присваивая, таким образом, каждой операции записи уникальный ключ [53].

Связь типа «происходит до» и конкурентный доступ

Как определить, конкурентны ли две операции? Чтобы лучше разобраться, рассмотрим несколько примеров.

- На рис. 5.9 две операции записи не конкурентны: вставка, выполненная клиентом A, *происходит до* приращения, выполненного клиентом B, поскольку увеличиваемое клиентом B значение — как раз то, которое вставил клиент A.

Другими словами, выполняемая клиентом B операция основана на операции, выполняемой клиентом A, поэтому операция клиента B должна выполняться позднее. Можно также сказать, что B *причинно-следственно зависит* от A.

- B то же время две операции записи на рис. 5.12 конкурентны: в момент начала операции ни один из клиентов не знает, что другой тоже выполняет операцию над тем же ключом. Следовательно, причинно-следственной связи между этими операциями нет.

Говорят, что операция A *происходит до* другой операции B, если B известно про A или B зависит от A либо основана на A каким-либо образом. Совершение одной операции до другой — ключ к формализации понятия конкурентности. Фактически мы можем просто сказать: две операции конкурентны, если ни одна из них не происходит до другой (то есть ни одна не знает про другую) [54].

Следовательно, для двух операций A и B есть три возможности: или A происходит до B, или B происходит до A, или A и B происходят конкурентно. Нам нужен алгоритм для определения, конкурентны ли две операции. Если одна операция происходит до другой, то последняя должна перезаписывать данные более ранней операции, но в случае их конкурентности возникает конфликт, который нужно разрешить.

Конкурентный доступ, время и относительность

Может показаться, что две операции должны называться конкурентными тогда, когда происходят одновременно, но на деле неважно, пересекаются ли они во времени. Вследствие проблем с часами в распределенных системах реально довольно сложно определить, происходят ли два события фактически в одно и то же время — вопрос, который мы обсудим детальнее в главе 8.

Чтобы определить, что такое конкурентный доступ, точное время неважно: мы просто будем называть две операции конкурентными, если ни одна из них не знает о другой, независимо от физического времени их выполнения. Иногда этот принцип связывается со специальной теорией относительности в физике [54], в которой излагается идея о невозможности перемещения информации со скоростью, превышающей скорость света. В результате два происходящих на некоем расстоянии события никак не могут влиять друг на друга, если разделяющий их промежуток времени меньше, чем время, которое потребовалось бы лучу света для прохождения соответствующего расстояния.

В вычислительных системах две операции могут быть конкурентными, несмотря на то что скорость света, в принципе, дает возможность одной из них влиять на другую. Например, если работа сети была замедлена или прерывалась в это время, то две операции могли произойти с некоторым интервалом, но все же оставаться конкурентными, поскольку сетевые проблемы помешали одной из операций получить информацию о другой.

Обнаружение связей типа «происходит до»

Взглянем на алгоритм, который определяет, являются ли две операции конкурентными, или одна происходит до другой. Ради простоты начнем с базы данных, содержащей только одну реплику. Разобравшись с работой этого алгоритма на одной реплике, мы сможем обобщить его на БД без ведущего узла с несколькими репликами.

На рис. 5.13 показаны два клиента, конкурентно добавляющие товары в одну корзину заказов (если этот пример кажется вам бессмысленным, то представьте двух авиадиспетчеров, параллельно добавляющих рейсы в отслеживаемый ими сектор). Изначально корзина заказов пуста. Клиенты выполняют пять операций записи в базу данных.

1. Клиент 1 добавляет в корзину заказов `milk`. Это первая операция записи для данного ключа, так что сервер успешно сохраняет данные и присваивает ей номер версии 1. Сервер также возвращает текущее значение клиенту вместе с номером версии.
2. Клиент 2 добавляет в корзину заказов `eggs`, не зная, что клиент 1 конкурентно добавил `milk` (клиент 2 полагает, что его `eggs` — единственный товар в корзине). Сервер присваивает этой операции записи номер 2 и сохраняет `eggs` и `milk` как два отдельных значения. Затем сервер возвращает клиенту *оба* значения вместе с номером версии — 2.
3. Клиент 1, не замечая операции записи клиента 2, собирается добавить в корзину `flour`, считая, что после этого содержимое корзины заказов будет `[milk, flour]`. Он отправляет данное значение на сервер вместе с номером версии 1, присвоенной ему ранее. Сервер понимает по номеру версии, что запись `[milk, flour]` заменяет предыдущее значение `[milk]`, но конкурентна с `[eggs]`. Следовательно, сервер присваивает `[milk, flour]` версию 3, замещая значение `[milk]` версии 1 и сохраняя значение `[eggs]` версии 2, и возвращает оба оставшихся значения клиенту.
4. Тем временем клиент 2 хочет добавить в корзину `ham`, не подозревая, что клиент 1 только что добавил `flour`. Клиент 2 получил в последнем ответе сервера два значения — `[milk]` и `[eggs]`, вследствие чего сливает эти значения воедино и добавляет `ham` для формирования нового значения — `[eggs, milk, ham]`. Сервер обнаруживает, что версия 2 замещает `[eggs]`, но конкурентна с `[milk, flour]`, поэтому остаются два значения `[milk, flour]` с версией 3 и `[eggs, milk, ham]` с версией 4.
5. Наконец, клиент 1 хочет добавить в корзину `bacon`. Ранее он уже добавил `[milk, flour]` и `[eggs]` с сервера с версией 3, так что он объединяет их, добавляет `bacon` и отправляет итоговое значение `[milk, flour, eggs, bacon]` серверу с номером версии 3. Это значение замещает `[milk, flour]` (обратите внимание: `[eggs]` уже заменены на предыдущем шаге), но конкурентно с `[eggs, milk, ham]`, вследствие чего сервер сохраняет эти два конкурентных значения.

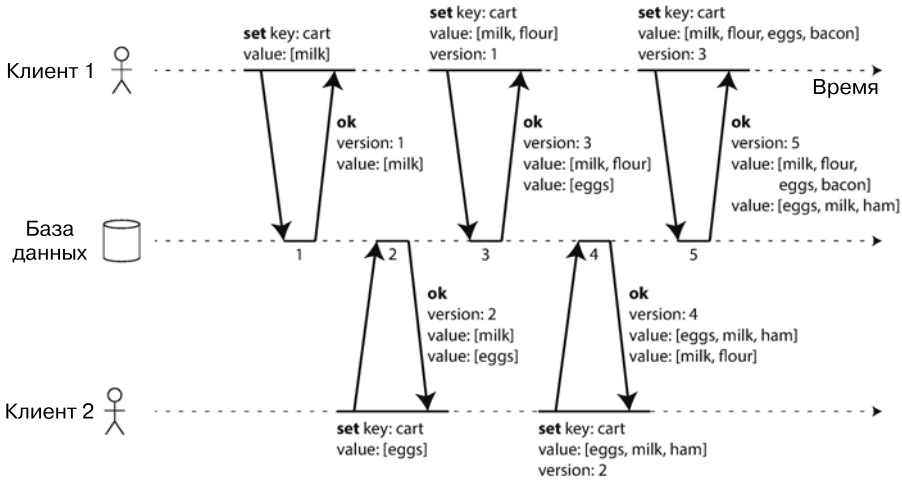


Рис. 5.13. Причинно-следственные зависимости между двумя клиентами, конкурентно редактирующими корзину заказов

Поток данных между операциями на рис. 5.13 проиллюстрирован графически на рис. 5.14. Стрелки указывают, какие операции *произошли до* других операций в том смысле, что более поздняя операция *знает о* более ранней или *зависит* от нее. В этом примере клиенты никогда не идут в ногу с данными на сервере, поскольку всякий раз конкурентно происходит еще одна операция. Но устаревшие версии значений постоянно перезаписываются, и ни одна операция записи не оказывается потерянной.

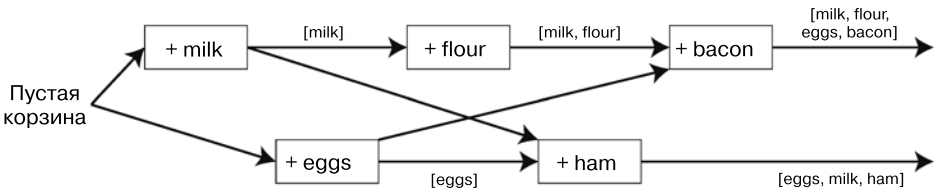


Рис. 5.14. Граф причинно-следственных зависимостей с рис. 5.13

Обратите внимание: сервер может определить, являются ли две операции конкурентными, по номерам версий — ему не требуется интерпретировать сами значения (так что те могут представлять собой произвольные структуры данных). Алгоритм работает следующим образом.

- Сервер хранит номера версий для всех ключей, увеличивая номер версии всякий раз при выполнении записи значения для этого ключа, и сохраняет новый номер версии вместе с записанным значением.

- При чтении ключа клиентом сервер возвращает все неперезаписанные значения, а также последний номер версии. Клиент должен прочитать ключ перед операцией записи.
- Клиент, записывая значение для ключа, должен включить номер версии из предыдущей операции чтения, а также объединить все полученные при предыдущей операции чтения значения. (Полученный в результате операции записи ответ может быть таким же, как и для чтения, с возвратом всех текущих значений, что позволяет соединять несколько операций записи последовательно, подобно примеру с корзиной заказов.)
- Сервер, получив информацию об операции записи с конкретным номером версии, может перезаписать все значения с этим или более низким номером версии (так как знает, что они все слиты воедино в новом значении), но должен сохранить все значения с более высоким номером версии (поскольку эти значения конкурентны данной входящей операции записи).

Номер версии из предыдущей операции, включаемый в операцию записи, обеспечивает информацию о том, на каком предыдущем состоянии основана эта операция записи. Операция записи, в которую не включен номер версии, будет конкурентна всем остальным операциям записи, так что не будет перезаписывать никакие данные, ее данные просто будут возвращены в качестве одного из значений при последующих операциях чтения.

Слияние конкурентно записываемых значений

Вышеприведенный алгоритм гарантирует, что данные не будут незаметно отбрасываться, но при этом клиентам приходится выполнять дополнительную работу: освобождать занимаемые ресурсы после конкурентного совершения нескольких операций путем объединения конкурентно записываемых значений. В Riak такие конкурентные значения называются *родственными значениями* (siblings).

Слияние родственных значений представляет собой, по сути, аналог обсуждавшейся выше задачи разрешения конфликтов при репликации с несколькими ведущими узлами (см. подраздел «Обработка конфликтов записи» раздела 5.3). Простейший подход: выбрать одно из значений на основе номера версии или метки даты/времени («выигрывает последний»), но это означает потерю данных. Поэтому необходимо сделать в коде приложения нечто более изощренное.

В примере с корзинами заказов разумным подходом было бы просто объединить значения. На рис. 5.14 два итоговых родственных значения — `[milk, flour, eggs, bacon]` и `[eggs, milk, ham]`; обратите внимание: `milk` и `eggs` встречаются в обоих, хотя каждое из них записывалось однократно. Объединенное значение может выглядеть примерно следующим образом: `[milk, flour, eggs, bacon, ham]`, без дубликатов.

Однако если вы хотите, чтобы люди могли и *удалять* товары из корзин заказов, а не только добавлять, то результат объединения родственных значений может оказаться не тем, который вам нужен: при объединении двух родственных корзин с удалением товара только из одной удаленный товар снова появится в объединении [37]. Чтобы избежать такой ситуации, нельзя просто удалять товар из базы данных при удалении его из корзины заказов, вместо этого система должна оставить в базе маркер с соответствующим номером версии в качестве указания на то, что товар был удален при слиянии родственных значений. Подобный маркер называется *отметкой об удалении* (tombstone). Мы уже сталкивались с ними в контексте сжатия журналов в подразделе «Хеш-индексы» раздела 3.1.

Поскольку выполнение сжатия родственных значений в коде приложения представляет собой сложную и подверженную потенциальным ошибкам задачу, были предприняты попытки создать структуры данных для автоматического слияния, как обсуждается во врезке «Автоматическое разрешение конфликтов» (см. пункт «Пользовательская логика разрешения конфликтов» подраздела «Обработка конфликтов записи» раздела 5.3). Например, Riak поддерживает семейство структур данных, называемых CRDT [38, 39, 55], способных автоматически объединять родственные значения подходящим способом, включая сохранение информации об удаленных значениях.

Векторы версий

В примере на рис. 5.13 используется только одна реплика. Как же изменится алгоритм при наличии нескольких реплик, но без ведущего узла?

В примере применяется один номер версии для отражения зависимостей между операциями, но его недостаточно при наличии нескольких реплик, конкурентно принимающих информацию об операциях записи. Вместо этого необходимо воспользоваться не только номером версии по ключам, но и номером версии по репликам. У каждой реплики есть свой собственный номер версии, который она увеличивает при выполнении каждой операции записи, отслеживая между тем попадающие к ней номера версий от каждой из других реплик. Исходя из данной информации одни значения перезаписываются, а другие — сохраняются в качестве родственных.

Набор номеров версий всех реплик называется *вектором версий* (version vector) [56]. Используется несколько разновидностей этого понятия, но самая интересная из них, вероятно, *точечный вектор версий* (dotted version vector) [57], применяемый в Riak [58, 59]. Мы не станем углубляться в подробности, но его функционирование весьма напоминает то, что мы видели в примере с корзинами заказов.

Аналогично номерам версии на рис. 5.13, векторы версий отправляются клиентам от реплик базы данных при чтении значений и должны отправляться обратно в БД

при последующей записи значения (Riak кодирует вектор версий в виде строки, называемой *причинно-следственным контекстом* (causal context)). Вектор версий дает базе возможность различать операции перезаписи и конкурентные операции записи.

Кроме того, подобно примеру с одной репликой, приложению может понадобиться объединить родственные значения. Структура вектора версий гарантирует безопасность чтения из одной реплики с последующей записью в другую. Это способно привести к созданию родственных значений, но данные не теряются, если, конечно, родственные значения были объединены правильно.



Векторы версий и векторные часы

Векторы версий иногда называют векторными часами (vector clock), хотя это не совсем одно и то же. Различие между ними едва уловимо (см. подробности в библиографии [57, 60, 61]). Если в двух словах, то при сравнении состояний реплик следует использовать именно векторы версий.

5.5. Резюме

В этой главе мы изучили вопрос репликации. Репликация может служить нескольким целям.

- *Высокая доступность*. Сохранение работоспособности системы в целом даже в случае отказа одной из машин (или нескольких машин, или даже целого ЦОДа).
- *Работа в офлайн-режиме*. Возможность продолжения работы приложения в случае прерывания соединения с сетью.
- *Задержка*. Данные размещаются географически близко к пользователям, чтобы те могли работать с ними быстрее.
- *Масштабирование*. Возможность обрабатывать бóльшие объемы операций чтения, чем способна обработать одна машина, с помощью выполнения операций чтения на репликах.

Несмотря на кажущуюся простоту задачи — хранение копий одних и тех же данных на нескольких машинах, — репликация оказывается весьма непростым делом. Она требует тщательного обдумывания вопроса конкурентного доступа и всех возможных сбоев, а также того, как справиться с их последствиями. Как минимум необходимо что-то делать с недоступными узлами и разрывами сети (и это не говоря уже о более коварных типах отказов, например незаметной порче данных вследствие программных ошибок).

Мы обсудили три основных метода репликации.

- *Репликация с одним ведущим узлом.* Клиенты отправляют информацию обо всех операциях записи одному узлу (ведущему), который отправляет поток событий изменения данных другим репликам (ведомым узлам). Операции чтения могут выполняться в любой реплике, но прочитанные из ведомых узлов данные могут оказаться устаревшими.
- *Репликация с несколькими ведущими узлами.* Клиенты отправляют информацию о каждой из операций записи одному из нескольких ведущих узлов, могущих эту информацию принимать. Ведущие могут отправлять поток событий изменения данных друг другу и любым ведомым.
- *Репликация без ведущего узла.* Клиенты отправляют информацию о каждой из операций записи одному из нескольких узлов и читают из нескольких узлов параллельно, чтобы обнаружить узлы с устаревшими данными и внести поправки.

У каждого из этих методов есть свои достоинства и недостатки. Репликация с одним ведущим узлом широко распространена в силу своей простоты и отсутствия надобности в разрешении конфликтов. Методы репликации с несколькими ведущими узлами и без ведущего узла, возможно, более устойчивы к отказам узлов, разрывам сети и резким скачкам времени задержки за счет усложнения и обеспечения лишь очень слабых гарантий согласованности.

Репликация может быть синхронной или асинхронной; это очень сильно влияет на поведение системы в случае сбоя. Хотя асинхронная репликация может выполняться быстрее в случае нормальной работы системы, важно понимать, что произойдет при росте задержки репликации и отказах серверов. В случае отказа ведущего узла и возведения одного из ведомых узлов в ранг ведущего существует вероятность потери недавно зафиксированных данных.

Мы рассмотрели некоторые необычные эффекты, вызываемые задержкой репликации, и обсудили несколько моделей согласованности, полезных при решении вопроса о требуемом поведении приложения в случае задержки репликации.

- *Согласованность типа «чтение после записи».* Пользователи должны всегда видеть данные, которые они сами отправили в БД.
- *Монотонное чтение.* После того как пользователь увидел данные по состоянию на какой-либо момент времени, он не должен позднее увидеть те же данные по состоянию на более ранний момент времени.
- *Согласованное префиксное чтение.* Пользователи должны видеть данные в состоянии, не нарушающем причинно-следственных связей: например, видеть вопрос и ответ на него в правильном порядке.

Наконец, мы обсудили неотъемлемые для репликации с несколькими ведущими узлами и без ведущего узла вопросы конкурентного доступа: поскольку в них

допускается несколько конкурентных операций записи, вероятны конфликты. Мы изучили алгоритм, который может использоваться в базах данных для выяснения, произошла ли одна операция до другой, или они происходили конкурентно. Мы также затронули методы разрешения конфликтов путем слияния конкурентных обновлений.

В следующей главе мы продолжим рассмотрение распределенных по нескольким машинам данных через призму эквивалента репликации: разбиения большого набора данных на *секции* (partitions).

5.6. Библиография

1. *Lindsay B. G., Griffiths Selinger P., Galtieri C., et al.* Notes on Distributed Databases // IBM Research, Research Report RJ2571(33471), July 1979 [Электронный ресурс]. — Режим доступа: http://domino.research.ibm.com/library/cyberdig.nsf/papers/A776EC17FC2FCE73852579F100578964/$File/RJ2571.pdf.
2. Oracle Active Data Guard Real-Time Data Protection and Availability // Oracle White Paper, June 2013 [Электронный ресурс]. — Режим доступа: http://www.oracle.com/technetwork/database/availability/active-data-guard-wp-12c-1896127.pdf.
3. AlwaysOn Availability Groups // SQL Server Books Online, Microsoft, 2012 [Электронный ресурс]. — Режим доступа: https://docs.microsoft.com/en-us/sql/database-engine/availability-groups/windows/always-on-availability-groups-sql-server.
4. *Qiao L., Surlaker K., Das S., et al.* On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform // ACM International Conference on Management of Data (SIGMOD), June 2013 [Электронный ресурс]. — Режим доступа: https://www.slideshare.net/amywtang/espresso-20952131.
5. *Rao J.* Intra-Cluster Replication for Apache Kafka // ApacheCon North America, February 2013 [Электронный ресурс]. — Режим доступа: https://www.slideshare.net/junrao/kafka-replication-apachecon2013.
6. Highly Available Queues // RabbitMQ Server Documentation, Pivotal Software, Inc., 2014 [Электронный ресурс]. — Режим доступа: https://www.rabbitmq.com/ha.html.
7. *Matsunobu Y.* Semi-Synchronous Replication at Facebook. April 1, 2014 [Электронный ресурс]. — Режим доступа: http://yoshinorimatsunobu.blogspot.com.by/2014/04/semi-synchronous-replication-at-facebook.html.
8. *Renesse van R., Schneider F. B.* Chain Replication for Supporting High Throughput and Availability // 6th USENIX Symposium on Operating System Design and Implementation (OSDI), December 2004 [Электронный ресурс]. — Режим доступа: http://static.usenix.org/legacy/events/osdi04/tech/full_papers/renesse/renesse.pdf.

9. *Terrace Jeff, Freedman M. J.* Object Storage on CRAQ: High Throughput Chain Replication for Read-Mostly Workloads // USENIX Annual Technical Conference (ATC), June 2009 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/legacy/event/usenix09/tech/full_papers/terrace/terrace.pdf.
10. *Calder B., Wang J., Ogus A., et al.* Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency // 23rd ACM Symposium on Operating Systems Principles (SOSP), October 2011 [Электронный ресурс]. — Режим доступа: http://sigops.org/sosp/sosp11/current/2011-Cascais/printable/11-calder.pdf.
11. *Wang A.* Windows Azure Storage. February 4, 2016 [Электронный ресурс]. — Режим доступа: http://umbrant.com/blog/2016/windows_azure_storage.html.
12. Percona Xtrabackup — Documentation // Percona LLC, 2014 [Электронный ресурс]. — Режим доступа: https://www.percona.com/doc/percona-xtrabackup/2.1/index.html.
13. *Newland J.* GitHub Availability This Week. September 14, 2012 [Электронный ресурс]. — Режим доступа: https://github.com/blog/1261-github-availability-this-week.
14. *Imbriaco M.* Downtime Last Saturday. December 26, 2012 [Электронный ресурс]. — Режим доступа: https://github.com/blog/1364-downtime-last-saturday.
15. *Hugg J.* 'All in' with Determinism for Performance and Testing in Distributed Systems // Strange Loop, September 2015 [Электронный ресурс]. — Режим доступа: https://www.youtube.com/watch?v=gJRj3vJL4wE.
16. *Kapila A.* WAL Internals of PostgreSQL // PostgreSQL Conference (PGCon), May 2012 [Электронный ресурс]. — Режим доступа: http://www.pgcon.org/2012/schedule/attachments/258_212_Internals%20Of%20PostgreSQL%20Wal.pdf.
17. MySQL Internals Manual. Oracle, 2014 [Электронный ресурс]. — Режим доступа: https://dev.mysql.com/doc/internals/en/.
18. *Sharma Y., Ajoux P., Ang P., et al.* Wormhole: Reliable Pub-Sub to Support Geo-Replicated Internet Services // 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI), May 2015 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-sharma.pdf.
19. Oracle GoldenGate 12c: Real-Time Access to Real-Time Information // Oracle White Paper, October 2013 [Электронный ресурс]. — Режим доступа: http://www.oracle.com/us/products/middleware/data-integration/oracle-goldengate-realtime-access-2031152.pdf.
20. *Das S., Botev C., Surlaker K., et al.* All Aboard the Databus! // ACM Symposium on Cloud Computing (SoCC), October 2012 [Электронный ресурс]. — Режим доступа: https://915bbc94-a-62cb3a1a-s-sites.googlegroups.com/site/acm2012socc/s18-das.pdf?attachauth=ANoY7cp9ZvqcEzdMKi__oOXwAxHReJqmzWexg4NyUyWnOQiFxf33LLiYyZmKH4msi9NSmg-c0pMxmDVxwd04KQyEre5LTgCIso1geycdxbMLPoBgFClOQvwOFUOXM7kzLjhwPBUNpKqPxoU2gFYsjIdSpZX5tOqe-2NzS5z599ZtTPCeCEtCOxXAm3-X-IPBwRHfQs-kwXk3uCxNIEl6Wf94HShzfICPIg%3D%3D&attredirects=0.

21. *Mullane G. S.* Version 5 of Bucardo Database Replication System // June 23, 2014 [Электронный ресурс]. — Режим доступа: http://blog.endpoint.com/2014/06/bucardo-5-multimaster-postgres-released.html.
22. *Vogels W.* Eventually Consistent // ACM Queue, volume 6, number 6, pages 14–19, October 2008 [Электронный ресурс]. — Режим доступа: http://queue.acm.org/detail.cfm?id=1466448.
23. *Terry D. B.* Replicated Data Consistency Explained Through Baseball // Microsoft Research, Technical Report MSR-TR-2011-137, October 2011 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/replicated-data-consistency-explained-through-baseball/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F157411%2Fconsistencyandbaseballreport.pdf.
24. *Terry D. B., Demers A. J., Petersen K., et al.* Session Guarantees for Weakly Consistent Replicated Data, at 3rd International Conference on Parallel and Distributed Information Systems (PDIS), September 1994 [Электронный ресурс]. — Режим доступа: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.71.2269&rep=rep1&type=pdf.
25. *Pratchett T.* Reaper Man: A Discworld Novel. — Victor Gollancz, 1991.
26. Tungsten Replicator // Continuent, Inc., 2014 [Электронный ресурс]. — Режим доступа: https://github.com/continuent.
27. BDR 0.10.0 Documentation // The PostgreSQL Global Development Group, 2015 [Электронный ресурс]. — Режим доступа: http://bdr-project.org/docs/next/index.html.
28. *Hodges R.* If You *Must* Deploy Multi-Master Replication, Read This First // March 30, 2012 [Электронный ресурс]. — Режим доступа: http://scale-out-blog.blogspot.com.by/2012/04/if-you-must-deploy-multi-master.html.
29. *Anderson J. C., Lehnardt J., Slater N.* CouchDB: The Definitive Guide. — O'Reilly Media, 2010.
30. AppJet, Inc.: Etherpad and EasySync Technical Manual. March 26, 2011 [Электронный ресурс]. — Режим доступа: https://github.com/ether/etherpad-lite/blob/e2ce9dc/doc/easysync/easysync-full-description.pdf.
31. *Day-Richter J.* What's Different About the New Google Docs: Making Collaboration Fast. 23 September 2010 [Электронный ресурс]. — Режим доступа: https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html.
32. *Kleppmann M., Beresford A. R.* A Conflict-Free Replicated JSON Datatype // arXiv:1608.03960, August 13, 2016 [Электронный ресурс]. — Режим доступа: https://arxiv.org/abs/1608.03960.
33. *Clement F.* Eventual Consistency — Detecting Conflicts // October 20, 2011 [Электронный ресурс]. — Режим доступа: http://messagepassing.blogspot.com.by/2011/10/eventual-consistency-detecting.html.

34. *Hodges R.* State of the Art for MySQL Multi-Master Replication // Percona Live: MySQL Conference & Expo, April 2013 [Электронный ресурс]. — Режим доступа: https://www.percona.com/live/mysql-conference-2013/sessions/state-art-mysql-multi-master-replication.

35. *Daily J.* Clocks Are Bad, or, Welcome to the Wonderful World of Distributed Systems. November 12, 2013 [Электронный ресурс]. — Режим доступа: http://basho.com/posts/technical/clocks-are-bad-or-welcome-to-distributed-systems/.

36. *Berton R.* Is Bi-Directional Replication (BDR) in Postgres Transactional? January 4, 2016 [Электронный ресурс]. — Режим доступа: http://sdf.org/~riley/blog/2016/01/04/is-bi-directional-replication-bdr-in-postgres-transactional/.

37. *DeCandia G., Hastorun D., Jampani M., et al.* Dynamo: Amazon's Highly Available Key-Value Store // 21st ACM Symposium on Operating Systems Principles (SOSP), October 2007 [Электронный ресурс]. — Режим доступа: http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf.

38. *Shapiro M., Preguiça N., Baquero C., Zawirski M.* A Comprehensive Study of Convergent and Commutative Replicated Data Types // INRIA Research Report no. 7506, January 2011 [Электронный ресурс]. — Режим доступа: https://hal.inria.fr/inria-00555588/.

39. *Elliott S.* CRDTs: An UPDATE (or Maybe Just a PUT) // RICON West, October 2013 [Электронный ресурс]. — Режим доступа: https://speakerdeck.com/lenary/crdts-an-update-or-just-a-put.

40. *Brown R.* A Bluffers Guide to CRDTs in Riak. October 28, 2013 [Электронный ресурс]. — Режим доступа: https://gist.github.com/russelldb/f92f44bdfb619e089a4d.

41. *Farinier B., Gazagnaire T., Madhavapeddy A.* Mergeable Persistent Data Structures // 26es Journées Francophones des Langages Applicatifs (JFLA), January 2015 [Электронный ресурс]. — Режим доступа: http://gazagnaire.org/pub/FGM15.pdf.

42. *Sun C., Ellis C.* Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements // ACM Conference on Computer Supported Cooperative Work (CSCW), November 1998 [Электронный ресурс]. — Режим доступа: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.53.933&rep=rep1&type=pdf.

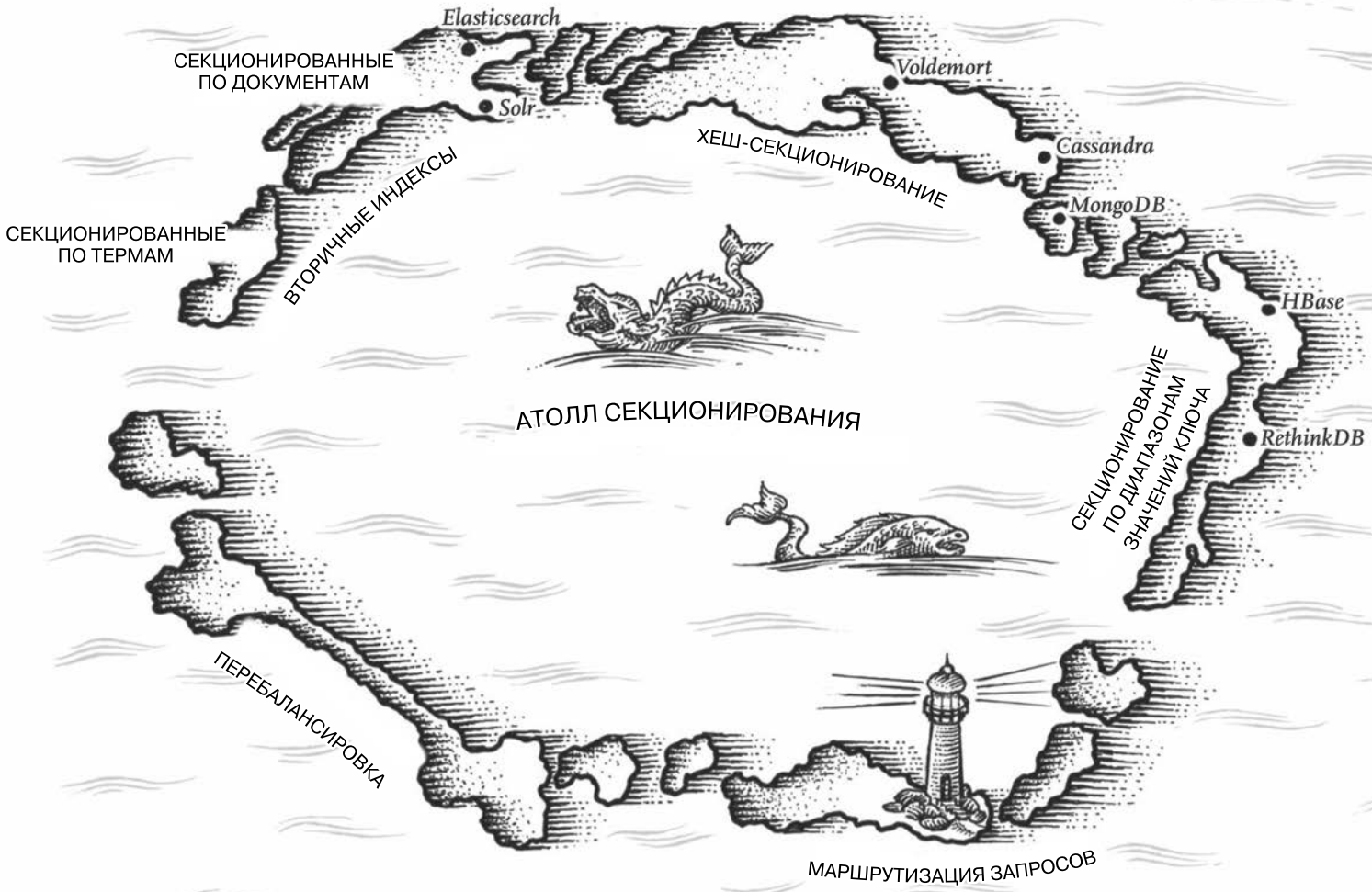
43. *Hofhansl L.* HBASE-7709: Infinite Loop Possible in Master/Master Replication // issues.apache.org, January 29, 2013 [Электронный ресурс]. — Режим доступа: https://issues.apache.org/jira/browse/HBASE-7709.

44. *Gifford D. K.* Weighted Voting for Replicated Data // 7th ACM Symposium on Operating Systems Principles (SOSP), December 1979 [Электронный ресурс]. — Режим доступа: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.84.7698.

45. *Howard H., Malkhi D., Spiegelman A.* Flexible Paxos: Quorum Intersection Revisited. August 24, 2016 [Электронный ресурс]. — Режим доступа: https://arxiv.org/abs/1608.06696.

46. *Blomstedt J.* Re: Absolute Consistency // email to riak-users mailing list, January 11, 2012 [Электронный ресурс]. — Режим доступа: http://lists.basho.com/pipermail/riak-users_lists.basho.com/2012-January/007157.html.
47. *Blomstedt J.* Bringing Consistency to Riak // RICON West, October 2012 [Электронный ресурс]. — Режим доступа: https://vimeo.com/51973001.
48. *Bailis P., Venkataraman S., Franklin M. J., et al.* Quantifying Eventual Consistency with PBS // Communications of the ACM, volume 57, number 8, pages 93–102, August 2014 [Электронный ресурс]. — Режим доступа: http://www.bailis.org/papers/pbs-cacm2014.pdf.
49. *Ellis J.* Modern Hinted Handoff. December 11, 2012 [Электронный ресурс]. — Режим доступа: https://www.datastax.com/dev/blog/modern-hinted-handoff.
50. Project Voldemort Wiki. 2013. https://github.com/voldemort/voldemort/wiki
51. Apache Cassandra 2.0 Documentation // DataStax, Inc., 2014 [Электронный ресурс]. — Режим доступа: http://docs.datastax.com/en/archived/cassandra/2.0/index.html http://basho.com/tag/multi-datacenter-replication/.
52. Riak Enterprise: Multi-Datacenter Replication // Technical whitepaper, Basho Technologies, Inc., September 2014 [Электронный ресурс]. — Режим доступа: http://basho.com/tag/multi-datacenter-replication/.
53. *Ellis J.* Why Cassandra Doesn't Need Vector Clocks. September 2, 2013 [Электронный ресурс]. — Режим доступа: https://www.datastax.com/dev/blog/why-cassandra-doesnt-need-vector-clocks.
54. *Lamport L.* Time, Clocks, and the Ordering of Events in a Distributed System // Communications of the ACM, volume 21, number 7, pages 558–565, July 1978 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Flamport%2Fpubs%2Ftime-clocks.pdf.
55. *Jacobson J.* Riak 2.0: Data Types. March 23, 2014 [Электронный ресурс]. — Режим доступа: https://offline.ghost.org/.
56. *Parker D. Stott Jr., Popek G. J., Rudisin G., et al.* Detection of Mutual Inconsistency in Distributed Systems // IEEE Transactions on Software Engineering, volume 9, number 3, pages 240–247, May 1983 [Электронный ресурс]. — Режим доступа: http://zoo.cs.yale.edu/classes/cs426/2013/bib/parker83detection.pdf.
57. *Preguiça N., Baquero C., Almeida P. Sérgio, et al.* Dotted Version Vectors: Logical Clocks for Optimistic Replication. November 26, 2010 [Электронный ресурс]. — Режим доступа: https://arxiv.org/pdf/1011.5808v1.pdf.
58. *Cribbs S.* A Brief History of Time in Riak // RICON, October 2014. [Электронный ресурс]. — Режим доступа: https://www.youtube.com/watch?v=HHkKPdOi-ZU.
59. *Brown R.* Vector Clocks Revisited Part 2: Dotted Version Vectors. November 10, 2015 [Электронный ресурс]. — Режим доступа: http://basho.com/posts/technical/vector-clocks-revisited-part-2-dotted-version-vectors/.

60. *Baquero C.* Version Vectors Are Not Vector Clocks. July 8, 2011 [Электронный ресурс]. — Режим доступа: https://haslab.wordpress.com/2011/07/08/version-vectors-are-not-vector-clocks/.
61. *Schwarz R., Mattern F.* Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail // Distributed Computing, volume 7, number 3, pages 149–174, March 1994 [Электронный ресурс]. — Режим доступа: https://disco.ethz.ch/courses/hs08/seminar/papers/mattern4.pdf.
62. *Пратчетт Т.* Мрачный жнец. — М. : Эксмо, 2017.


Elasticsearch
СЕКЦИОНИРОВАННЫЕ ПО ДОКУМЕНТАМ
Solr
Voldemort
Cassandra
MongoDB
HBase
RethinkDB
ХЕШ-СЕКЦИОНИРОВАНИЕ
ВТОРИЧНЫЕ ИНДЕКСЫ
СЕКЦИОНИРОВАННЫЕ ПО ТЕРМАМ
АТОЛЛ СЕКЦИОНИРОВАНИЯ
СЕКЦИОНИРОВАНИЕ ПО ДИАПАЗОНАМ ЗНАЧЕНИЙ КЛЮЧА
ПЕРЕБАЛАНСИРОВКА
МАРШРУТИЗАЦИЯ ЗАПРОСОВ

6 Секционирование

> Совершенно очевидно, что нам следует отказаться от последовательного выполнения операций и не ограничивать компьютеры им. Мы должны формулировать определения, расставлять приоритеты и давать описания данных. Мы должны формулировать связи, а не процедуры.
>
> *Грейс Мюррей Хоппер. Управление и компьютер будущего (1962)*

В главе 5 мы обсуждали репликацию — наличие нескольких копий одних данных в различных узлах. В случае очень больших наборов данных или объемов обрабатываемой информации этого недостаточно: необходимо разбить данные на *секции* (partitions), иначе говоря, выполнить *шардинг* (sharding) данных[1].



Путаница в терминологии

То, что мы называем секцией (partition), в MongoDB, Elasticsearch и SolrCloud называется «шард» (shard), в HBase — «регион» (region), в Bigtable — «сегмент» (tablet), в Cassandra и Riak — «виртуальный узел» (vnode) и в Couchbase — «виртуальный участок» (vBucket). Однако «секционирование» (partitioning) — наиболее часто употребляемый термин, поэтому мы будем использовать его.

[1] Секционирование (partitioning), как вы увидите в этой главе, представляет собой способ умышленного разбиения большого набора данных на меньшие. Оно не имеет никакого отношения к разбиению сети на фрагменты (network partitioning). Мы обсудим подобные сбои в главе 8.

Обычно секции задаются таким образом, что каждый элемент данных (запись, строка или документ) относятся ровно к одной секции. Этого можно достичь множеством способов, часть из которых мы обсудим подробно в настоящей главе. Фактически каждая секция сама по себе является маленькой БД, хотя база способна поддерживать операции, затрагивающие сразу несколько секций.

Основная цель секционирования данных — *масштабируемость*. Разные секции можно разместить в различных узлах в кластере, не предусматривающем разделения ресурсов (см. определение термина *«не предусматривающий разделения ресурсов»* во введении к части II). Следовательно, большой набор данных можно распределить по многим жестким дискам, а запросы — по многим процессорам.

При односекционных запросах каждый узел способен независимо выполнять запросы в своей секции, так что пропускную способность по запросам можно масштабировать просто добавлением новых узлов. Большие, сложные запросы можно распараллелить по нескольким узлам, хотя это намного сложнее.

Секционированные БД появились в 1980-х годах. Это были такие программные продукты, как Teradata и Tandem NonStop SQL [1]. Недавно технология была открыта заново базами данных NoSQL и складами данных на основе Hadoop. Одни из систем спроектированы в расчете на нагрузку в виде обработки транзакций, а другие предназначены для аналитики (см. раздел 3.2): это влияет на настройку системы, но основы секционирования не отличаются для обоих видов нагрузки.

В настоящей главе мы сначала рассмотрим различные подходы к секционированию больших наборов данных и проследим, как индексация данных сочетается с секционированием. Затем поговорим о перебалансировке, необходимой при добавлении или удалении узлов из кластера. Наконец, доберемся до обзора маршрутизации базами данных запросов к нужной секции и выполнения запросов.

6.1. Секционирование и репликация

Секционирование обычно идет бок о бок с репликацией, вследствие чего копии каждой из секций хранятся на нескольких узлах. Это значит, что, хотя каждая конкретная запись относится только к одной секции, храниться она может в нескольких различных узлах в целях отказоустойчивости.

В узле может храниться более одной секции. При использовании модели репликации типа «ведущий — ведомый» сочетание секционирования и репликации будет выглядеть так, как показано на рис. 6.1. Для каждой секции один из узлов назначается ведущим, а другие — ведомыми. Каждый узел может быть ведущим для одних секций и ведомым для других.

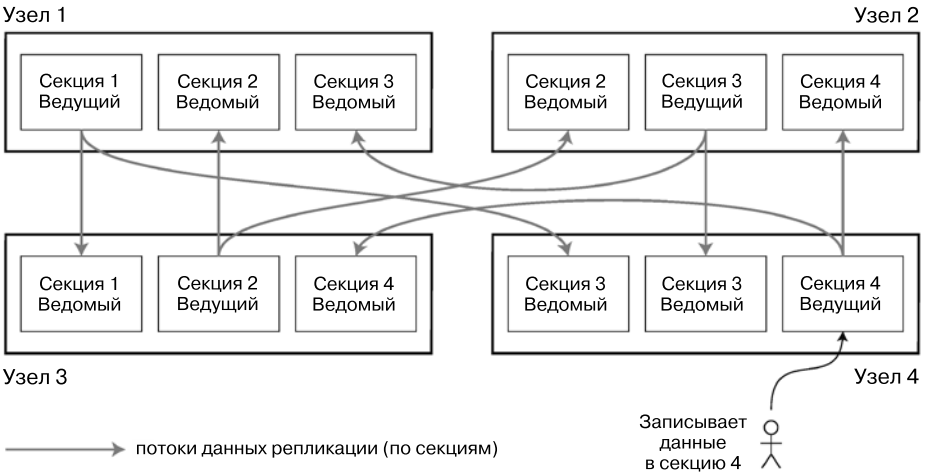


Рис. 6.1. Комбинация секционирования и репликации: каждый узел выступает в качестве ведущего для одних секций и ведомого для других

Все обсуждавшееся в главе 5 относительно репликации баз данных в равной степени применимо к репликации секций. Выбор схемы секционирования по большей части независим от выбора схемы репликации, так что упростим все и не будем учитывать репликацию в этой главе.

6.2. Секционирование данных типа «ключ — значение»

Допустим, у вас имеется большой объем данных, которые вы хотели бы секционировать. Как определить, на каких узлах хранить те или иные записи?

Цель секционирования — равномерно распределить по узлам данные и загрузку по запросам. Если доли всех узлов примерно равны, то теоретически десять узлов смогут обрабатывать в десять раз больше данных и будут иметь в десять раз бóльшую пропускную способность по чтению и записи, чем отдельный узел (без учета репликации).

Если же секционирование выполнено неравномерно, читай — на долю некоторых секций приходится больше данных или запросов, чем на долю других, — то оно называется *асимметричным* (skewed). Наличие асимметрии существенно снижает эффективность секционирования. В предельном случае вся нагрузка ляжет на одну секцию, так что девять из десяти узлов будут простаивать, а узким местом окажется единственный функционирующий узел. Секция с непропорционально высокой нагрузкой называется *горячей точкой* (hot spot).

Простейший способ избежать горячих точек — назначать узлы для записей случайным образом. Это приведет к равномерному распределению данных по узлам; но есть у такого подхода и большой недостаток: при чтении неизвестно, в каком узле находится конкретный элемент данных, вследствие чего приходится параллельно опрашивать все узлы.

Можно добиться большего. Допустим пока, что имеем дело с простой моделью данных типа «ключ — значение», в которой доступ к записям всегда осуществляется по первичному ключу. Например, в старомодной бумажной энциклопедии можно найти статью по ее названию, а поскольку все статьи отсортированы в алфавитном порядке, нужная найдется очень быстро.

Секционирование по диапазонам значений ключа

Один из методов секционирования — назначить каждой из секций непрерывный диапазон значений ключа (от какого-то минимального значения до какого-то максимального), подобно томам бумажной энциклопедии (рис. 6.2). Если вам известны границы между диапазонами, то можно легко определить, в какой секции содержится нужный ключ. Если вдобавок знать, с какими секциями какие узлы соотносятся, то можно выполнить запрос непосредственно к соответствующему узлу (или — в случае энциклопедии — снять требуемую книгу с полки).

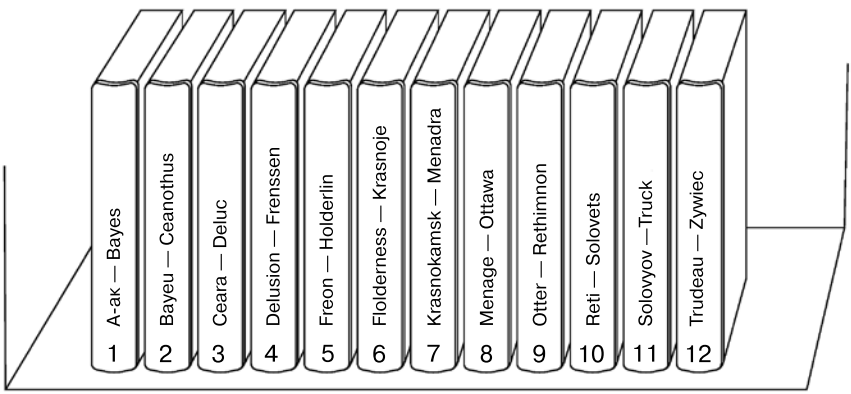


Рис. 6.2. Бумажная энциклопедия секционирована по диапазонам значений ключа

Диапазоны значений ключа не обязательно должны быть одинакового размера, поскольку данные могут быть распределены неравномерно. Например, на рис. 6.2 том 1 содержит слова, начинающиеся с букв A и B, а том 12 — слова, начинающиеся с букв T, U, V, X, Y и Z. Если бы каждый том соответствовал двум буквам алфавита, то некоторые тома были бы намного больше других. Чтобы данные были распределены равномерно, границы секций должны быть подобраны в соответствии с данными.

Границы секций могут выбираться автоматически базой данных или вручную администратором (мы обсудим вопрос выбора границ секций подробнее в разделе 6.4). Подобная стратегия секционирования применяется в БД Bigtable, ее эквиваленте с открытым исходным кодом HBase [2, 3], а также в RethinkDB и MongoDB вплоть до версии 2.4 [4].

В пределах каждой секции можно хранить ключи в отсортированном порядке (см. подраздел «SS-таблицы и LSM-деревья» раздела 3.1). Преимущество этого в упрощении просмотра диапазона по индексу, да и ключи можно рассматривать как сцепленный индекс и извлекать несколько связанных записей одним запросом (см. пункт «Составные индексы» подраздела «Другие индексные структуры» раздела 3.1). Например, рассмотрим приложение, хранящее данные, получаемые от сети датчиков, где ключом выступает метка даты/времени измерения *(год-месяц-день-час-минута-секунда)*. Просмотры диапазонов по индексу очень удобны в этом случае, поскольку позволяют легко извлекать, скажем, все данные за конкретный месяц.

Однако недостатком секционирования по диапазонам значений ключа является то, что некоторые паттерны доступа приводят к горячим точкам. Если ключ представляет собой метку даты/времени, то секции соответствуют отрезкам времени, например одна секция на день. К сожалению, в случаях, когда данные записываются в базу по мере получения значений от датчиков, все операции записи будут выполняться в одной секции (сегодняшней), вследствие чего эта секция окажется перегружена операциями записи, а остальные будут простаивать [5].

Чтобы избежать этой проблемы в базе данных, полученных с датчиков, необходимо использовать в качестве первого элемента ключа не метку даты/времени, а нечто иное. Например, можно предварить метку даты/времени названием датчика, и, как следствие, секционирование будет выполняться сначала по названию датчика, а только потом по времени. При условии одновременной работы множества датчиков нагрузка по записи распределится по секциям более равномерно. Но при необходимости получить значения с нескольких датчиков в пределах определенного промежутка времени придется выполнять отдельный запрос по диапазону для каждого из названий датчиков.

Секционирование по хешу ключа

Многие распределенные базы данных, опасаясь асимметрии и горячих точек, используют для распределения ключей по секциям хеш-функцию.

Хорошая хеш-функция получает на входе асимметричные данные и возвращает равномерно распределенные значения. Допустим, у вас есть 32-битная хеш-функция, получающая на входе строковое значение. Для каждой строки на входе она возвращает псевдослучайное число в диапазоне от 0 до $2^{32} - 1$. Даже если входные строки очень похожи, их хеши равномерно распределены по этому интервалу.

В целях секционирования хеш-функция не должна быть криптографически стойкой: например, СУБД Cassandra и MongoDB используют MD5, а Voldemort — функцию

Фаулера-Нола-Во[1]. Во многих языках программирования есть встроенные простые хеш-функции (они применяются для хеш-таблиц), но могут оказаться неподходящими для секционирования: например, `Object.hashCode()` в языке Java и `Object#hash` в Ruby могут возвращать различные значения хеша для одного ключа в разных процессах [6].

При наличии подходящей хеш-функции для ключей можно поставить каждой секции в соответствие диапазон хешей (вместо диапазона ключей), и каждый ключ, чье значение находится в диапазоне данной секции, будет сохранен в ней. Это показано на рис. 6.3.

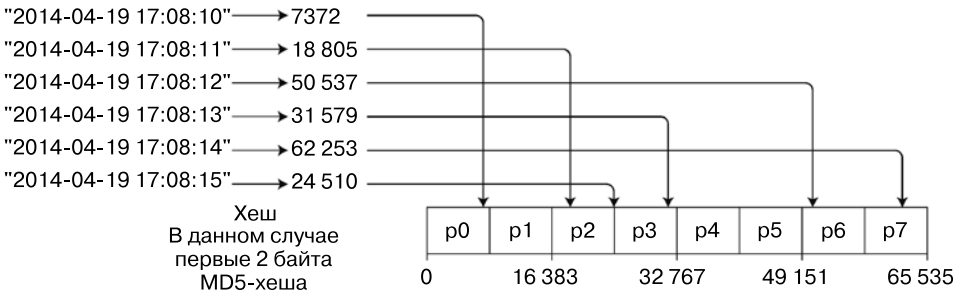


Рис. 6.3. Секционирование по хешу ключа

Это хороший способ равномерного распределения ключей по секциям. Границы секций могут быть или распределены равномерно, или выбраны псевдослучайным образом (в последнем случае метод иногда называется *согласованным хешированием* (consistent hashing)).

> **Согласованное хеширование**
>
> Согласованное хеширование в соответствии с определением, данным Каргером и др. [7], представляет собой способ равномерного распределения нагрузки по системе хешей в масштабах Интернета, например сети доставки контента (content delivery network, CDN). Оно использует выбранные случайным образом границы секций, чтобы избежать необходимости в централизованном управлении или распределенном консенсусе. Обратите внимание: термин *«согласованность»* здесь не имеет ничего общего с согласованностью реплик (см. главу 5) или ACID-согласованностью (см. главу 7), а просто описывает конкретный подход к перебалансировке.
>
> Как мы увидим в разделе 6.4, этот подход не очень хорошо работает в базах данных [8], так что на практике используется редко (документация по некоторым БД тем не менее часто говорит о согласованном хешировании, но это зачастую неправильная формулировка). В связи с этим лучше избегать термина «согласованное хеширование» и называть его *хеш-секционированием* (hash partitioning).

[1] См. https://ru.wikipedia.org/wiki/FNV. — *Примеч. пер.*

Однако, к сожалению, при использовании для секционирования хеша ключа мы теряем удобное свойство секционирования по диапазонам значений ключа: возможность эффективно выполнять запросы по диапазонам. Смежные некогда ключи оказываются разбросаны по всем секциям, и порядок их сортировки теряется. В MongoDB при включении режима шардинга на основе хеша любой запрос по диапазону приходится отправлять все секции [4]. Запросы по диапазонам на основе первичного ключа не поддерживаются ни Riak [9], ни Couchbase [10], ни Voldemort.

Cassandra достигает компромисса между этими двумя стратегиями секционирования [11–13]. В СУБД Cassandra допустимо объявление таблиц с *составным первичным ключом* (compound primary key), состоящим из нескольких столбцов. Для определения секции берется хеш только первой части такого ключа, а остальные используются в качестве сцепленного индекса для сортировки данных в SS-таблицах Cassandra. Следовательно, искать запросом диапазон значений первого столбца составного ключа невозможно, но задание для первого столбца конкретного значения позволяет выполнять эффективный поиск по диапазонам значений других столбцов ключа.

Подход со сцепленным индексом позволяет задействовать изящную модель данных для связей «один-ко-многим». Например, на сайте соцсети один пользователь может отправлять в свой блог множество сообщений. Если выбрать в качестве первичного ключа для сообщений `(user_id, update_timestamp)`, то можно будет эффективно извлекать все сообщения от конкретного пользователя за определенный промежуток времени, отсортированные по метке даты/времени. Сообщения различных пользователей могут храниться в разных секциях, но в пределах одного пользователя все сообщения хранятся в одной секции отсортированными по этой метке.

Асимметричные нагрузки и разгрузка горячих точек

Как уже обсуждалось выше, хеширование ключа для определения секции может несколько исправить ситуацию с горячими точками, но не избавиться от них полностью: в предельном случае, когда все операции записи и чтения выполняются для одного ключа, все запросы все равно приходятся на одну секцию.

Подобная нагрузка, вероятно, необычна, но не беспрецедентна: например, на сайтах соцсетей действия знаменитостей, насчитывающих миллионы подписчиков, могут вызывать бурю активности [14]. Такие события могут привести к огромным объемам операций записи для одного ключа (где ключом, вероятно, будет ID пользователя этой знаменитости или ID действия, которое комментируют подписчики). Хеширование ключа не поможет в такой ситуации, ведь хеш двух одинаковых ключей — одинаковый.

На сегодняшний день большинство информационных систем не умеют автоматически выравнивать подобную высоко асимметричную нагрузку, вследствие чего

снижение асимметрии — обязанность приложения. Например, если известно, что конкретный ключ — очень горячий, то простейшим решением будет добавление в начало или конец этого ключа случайного числа. Простое двузначное десятичное число приведет к разбиению операций записи для ключа равномерно по 100 различным ключам, что позволит распределить их по разным секциям.

Однако при условии разбиения операций записи по различным ключам операциям чтения придется совершать дополнительные действия по чтению и объединению данных для всех этих ключей. Описанный метод требует также дополнительных вспомогательных операций: добавлять случайное число имеет смысл только для небольшого числа горячих ключей, для абсолютного большинства ключей с низкими объемами операций записи это окажется лишними накладными расходами. Следовательно, понадобится отслеживать, какие ключи были разбиты.

Возможно, в будущем информационные системы смогут автоматически обнаруживать и выравнивать асимметричную нагрузку, но пока что вам придется самим обдумывать плюсы и минусы подобного подхода для ваших приложений.

6.3. Секционирование и вторичные индексы

Обсуждавшиеся до сих пор схемы секционирования основаны на модели данных «ключ — значение». Если обращение к записям происходит исключительно по их первичному ключу, то по нему можно узнать секцию и воспользоваться этими сведениями в целях маршрутизации запросов в соответствующую секцию.

Ситуация осложняется в случае применения вторичных индексов (см. также подраздел «Другие индексные структуры» раздела 3.1). Вторичный индекс обычно не идентифицирует запись однозначно, а представляет собой скорее способ поиска вхождений конкретного значения: «найти все действия пользователя `123`», «найти все статьи, содержащие слово `hogwash`», «найти все машины красного цвета» и т. д.

Вторичные индексы — каждодневная реальность реляционных баз данных, и в документоориентированных БД они тоже часто используются. Многие хранилища данных типа «ключ — значение» (такие как HBase и Voldemort) избегают вторичных индексов из-за усложнения реализации, но некоторые (например, Riak) начали добавлять их в силу их исключительного удобства для моделирования данных. И наконец, вторичные индексы — сама суть таких поисковых серверов, как Solr и Elasticsearch.

Основная проблема вторичных индексов — в том, что невозможно поставить их в четкое соответствие секциям. Существует два основных подхода к секционированию базы данных с вторичными индексами: секционирование по документам (document-based partitioning) и секционирование по термам (term-based partitioning).

Секционирование вторичных индексов по документам

Например, представьте, что у вас есть сайт для продажи подержанных автомобилей (как показано на рис. 6.4). У каждого описания есть уникальный ID — назовем его *идентификатором документа*, — и мы можем секционировать базу данных по этому идентификатору (например, ID с 0 по 499 — в секции 0, ID с 500 по 999 — в секции 1 и т. д.).

Необходимо, чтобы пользователи могли искать автомобили, фильтруя их по цвету и производителю, поэтому нам понадобится вторичный индекс по `color` и `make` (в документоориентированной базе данных они будут полями, в реляционной — столбцами). Достаточно будет описать индекс, БД может выполнять индексацию автоматически[1]. Например, при каждом добавлении автомобиля красного цвета база автоматически добавляет в список идентификаторов документов для индекса запись `color:red`.

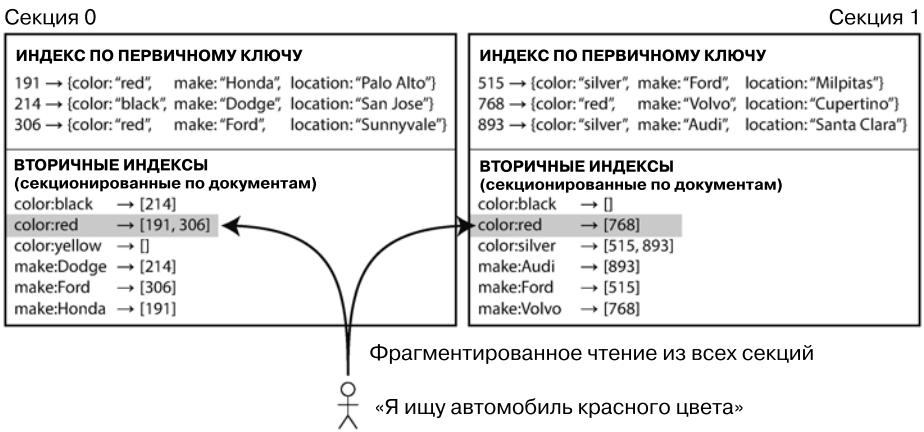


Рис. 6.4. Секционирование вторичных индексов по документам

При таком подходе к индексации все секции совершенно самостоятельны: каждая секция поддерживает свои собственные вторичные индексы, охватывающие только

[1] Если БД поддерживает только модель данных типа «ключ — значение», может показаться заманчивым реализовать вторичный индекс самостоятельно, путем создания соответствия значений идентификаторам документов в коде приложения. Но при этом необходимо внимательно следить за тем, чтобы индексы были согласованы с самими данными. Состояния гонки и периодические отказы при записи (при которых часть изменений сохраняется, а часть — нет) могут легко привести к рассинхронизации данных (см. пункт «Востребованность многообъектных транзакций» подраздела «Однообъектные и многообъектные операции» раздела 7.1).

документы из этой секции. Данные из других секций значения для нее не имеют. При необходимости выполнить запись в БД — для добавления, удаления или обновления документа — приходится работать только с секцией, в которой содержится идентификатор записываемого документа. Поэтому секционированный по документам индекс также называется *локальным индексом* (local index), в отличие от *глобальных индексов* (global index), описываемых в следующем подразделе.

Однако чтение из индекса, секционированного по документам, требует внимательности: если ничего не сделать с идентификаторами документов, то совершенно не обязательно, что все машины конкретного цвета или конкретного производителя окажутся в одной секции. На рис. 6.4 как в секции 0, так и в секции 1 есть красные автомобили. Следовательно, при поиске таких автомобилей придется выполнять запрос ко *всем* секциям с объединением полученных результатов.

Подобная методика выполнения запросов к секционированной базе данных известна под названием *фрагментированной* (scatter/gather), и запросы на чтение при ней могут оказаться весьма затратными. Даже при параллельном выполнении запросов к секциям фрагментированное чтение часто приводит к усилению «хвостового» времени ожидания (см. врезку «Процентили на практике» в подразделе «Описание производительности» раздела 1.3). Тем не менее она широко используется: MongoDB, Riak [15], Cassandra [16], Elasticsearch [17], SolrCloud [18] и VoltDB [19] — все применяют секционированные по документам вторичные индексы. Большинство производителей БД рекомендуют структурировать схему секционирования так, чтобы запросы по вторичным индексам могли выдаваться из одной секции, хотя это не всегда оказывается возможно, особенно при использовании нескольких вторичных индексов в одном запросе (например, при фильтрации автомобилей по цвету и производителю одновременно).

Секционирование вторичных индексов по термам

Вместо отдельного вторичного индекса для каждой секции (*локальные индексы*) можно сконструировать *глобальный индекс* (global index), охватывающий данные из всех секций. Однако такой индекс нельзя хранить в одном узле, иначе он превратится в узкое место и сведет на нет все выгоды от секционирования. Глобальный индекс тоже нужно секционировать, но его можно секционировать не так, как индекс по первичному ключу.

Рисунок 6.5 демонстрирует это: красные автомобили из всех секций фигурируют в данном индексе под `color:red`, но индекс секционирован так, что цвета, начинающиеся с букв от *a* до *r*, находятся в секции 0, а цвета, начинающиеся с букв от *s* до *z*, — в секции 1. Индекс по производителям автомобилей секционирован аналогично (граница секций проходит между буквами *f* и *h*).

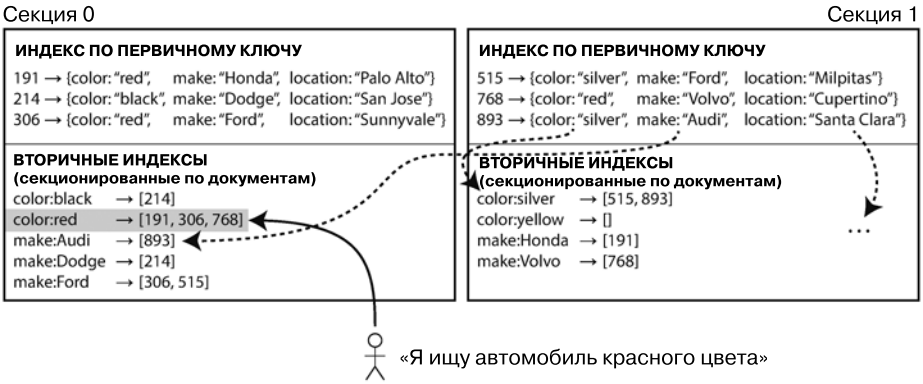


Рис. 6.5. Секционирование вторичных индексов по термам

Такая разновидность индекса называется *секционированной по терму* (term-partitioned), поскольку искомый терм определяет секционирование индекса. В данном случае, например, термом является `color:red`. Название *«терм»* (term) берет свое начало от полнотекстовых индексов (одного из видов вторичных индексов), в которых термами являются все слова, встречающиеся в документе.

Как и ранее, мы можем секционировать индекс по самому терму или по его хешу. Первое может оказаться полезным для поисков по диапазону (например, по числовому свойству, как при запросе цены автомобиля), а второе обеспечивает более равномерное распределение нагрузки.

Преимуществ глобальных (секционированных по терму) индексов над секционированными по документу состоит в повышении производительности чтения: вместо фрагментированного чтения по всем секциям клиенту нужно только выполнить запрос к секции, содержащей нужный терм. Однако недостаток глобальных индексов — в замедлении и усложнении операций записи, поскольку запись в отдельный документ может затронуть несколько секций индекса (все термы в документе могут находиться в разных секциях и в разных узлах).

В идеальном мире индексы были бы всегда актуальными, а каждый записываемый в базу данных документ сразу же отражался бы в индексе. Однако в секционированном по терму индексе это потребует выполнения распределенной транзакции над всеми секциями, затрагиваемыми текущей операцией записи, а такую возможность поддерживают далеко не все СУБД (см. главы 7 и 9).

На практике обновления глобальных индексов часто выполняются асинхронно (то есть при чтении индекса вскоре после операции записи выполненные изменения могут оказаться еще не видны). Например, компания Amazon утверждает, что глобальные индексы в ее СУБД DynamoDB при обычных условиях обновляются за доли секунды, но в случае сбоев в инфраструктуре задержки распространения могут быть более длительными [20].

Среди других мест применения секционированных по термам глобальных индексов — функциональность поиска СУБД Riak [21] и склад данных Oracle, позволяющие выбирать между локальной и глобальной индексацией [22]. Мы вернемся к вопросу реализации секционированных по термам вторичных индексов в главе 12.

6.4. Перебалансировка секций

С течением времени положение дел в базе данных меняется:

- количество обрабатываемых запросов растет, так что для возросшей нагрузки понадобятся дополнительные процессоры;
- размер набора данных растет, поэтому для его хранения понадобятся дополнительные жесткие диски и оперативная память;
- некоторые компьютеры испытывают сбои, вследствие чего другим компьютерам приходится брать на себя их обязанности.

Все эти изменения требуют перемещения данных и перенаправления запросов из одних узлов в другие. Процесс перемещения нагрузки с одного узла в кластере на другой называется *перебалансировкой* (rebalancing).

Вне зависимости от используемой схемы секционирования перебалансировка обычно должна отвечать определенным минимальным требованиям:

- после перебалансировки нагрузка (хранение данных, запросы на чтение и запись) должна быть распределена равномерно по узлам кластера;
- база данных должна продолжать принимать запросы на чтение и запись во время перебалансировки;
- между узлами должно перемещаться ровно то количество данных, которое необходимо, ради ускорения перебалансировки и минимизации нагрузки по вводу/выводу на сеть и жесткие диски.

Методики перебалансировки

Существует несколько различных способов распределения секций по узлам [23]. Обсудим вкратце их все по очереди.

Как делать не следует: хеширование по модулю N

При секционировании по хешу ключа, как мы упоминали ранее (рис. 6.3), лучше всего разбить возможные хеши на диапазоны и поставить в соответствие каждому диапазону секцию (например, *key* относится к секции 0, если $0 \le hash(key) < b_0$, к секции 1, если $b_0 \le hash(key) < b_1$ и т. д.).

Вероятно, вы недоумеваете, почему мы не используем просто *mod* (оператор `%` во многих языках программирования). Например, *hash(key) mod 10* возвращает число

от 0 до 9 (если записывать хеш в виде десятичного числа, то хеш по модулю 10 будет равен его последней цифре). В случае десяти узлов, пронумерованных от 0 до 9, это кажется удобным способом назначения узлов для ключей.

Проблема с методом *mod N* состоит в том, что при изменении количества *N* узлов придется перенести большинство ключей из одного узла в другой. Например, допустим, *hash(key)* = 123456. Если изначально у нас десять узлов, то этот ключ окажется в узле номер 6 (поскольку 123456 *mod* 10 = 6). Когда система вырастет до 11 узлов, этот ключ придется перенести в узел 3 (поскольку 123456 *mod* 11 = 3), а когда вырастет до 12 — в узел 0 (поскольку 123456 *mod* 12 = 0). Такие частые перемещения делают перебалансировку очень дорогим удовольствием.

Нам требуется подход, при котором не нужно перемещать данные больше, чем необходимо.

Фиксированное количество секций

К счастью, существует очень простое решение: создать намного больше секций, чем узлов в системе, и распределить по нескольку секций на каждый узел. Например, можно разбить работающую на кластере из десяти узлов базу данных на 1000 секций из расчета по 100 секций на каждый узел.

Тогда добавляемый в кластер новый узел может «позаимствовать» по нескольку секций у каждого из существующих узлов на время, до тех пор пока секции не станут снова распределены равномерно. Этот процесс проиллюстрирован на рис. 6.6. При удалении узла из кластера выполняется обратный процесс.

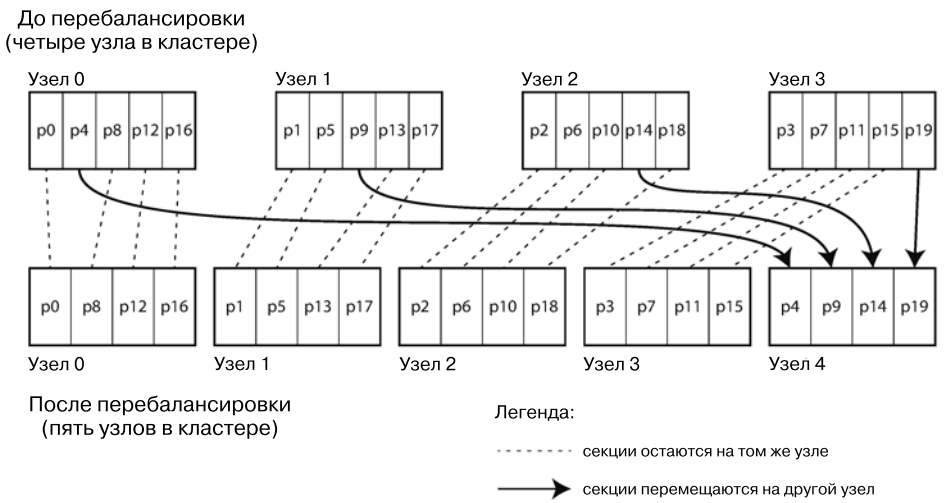


Рис. 6.6. Добавление нового узла в кластер базы данных с несколькими секциями на узел

Между узлами перемещаются только секции целиком. Количество последних не меняется, как и соответствие ключей секциям. Единственное, что меняется, — распределение секций по узлам. Это не происходит мгновенно (перемещение значительного количества данных по сети занимает некоторое время), и, как следствие, для всех происходящих во время перемещения операций записи и чтения используется старое распределение секций.

В принципе, можно даже принять во внимание различия в аппаратном обеспечении машин кластера: распределив на более производительные узлы больше секций, можно вынудить их взять на себя бóльшую долю нагрузки.

Такой подход к перебалансировке используется в Riak [15], Elasticsearch [24], Couchbase [10] и Voldemort [25].

При такой конфигурации количество секций обычно задается при первой настройке базы данных и потом не меняется. Хотя, в принципе, секции можно разбивать и сливать (см. следующий раздел), эксплуатация фиксированного количества секций проще, так что многие БД с фиксированным количеством секций предпочитают не реализовывать разбиение секций вообще. Следовательно, заданное вначале количество секций равно максимальному количеству возможных узлов, поэтому его рекомендуется выбирать достаточно большим для обеспечения возможности будущего роста. Однако каждая секция означает дополнительные накладные расходы на управление, вследствие чего выбирать слишком большое количество неразумно.

Выбор правильного количества секций особенно сложен при сильной изменчивости общего размера набора данных (например, если вначале он невелик, но может сильно вырасти со временем). Поскольку в каждой секции содержится фиксированная доля всех данных, размер секций растет пропорционально общему объему данных в кластере. При очень большом размере секций перебалансировка и восстановление после отказов узлов требуют значительных ресурсов. А слишком маленький размер секций приводит к слишком большим накладным расходам. Наилучшая производительность достигается при «идеальном» размере секций, когда они не слишком большие и не слишком маленькие, чего непросто добиться при фиксированном количестве секций и меняющемся размере набора данных.

Динамическое секционирование

Для использующих динамическое секционирование БД (см. подраздел «Секционирование по диапазонам значений ключа» раздела 6.2) фиксированное количество секций с фиксированными границами было бы очень неудобно: если вы зададите границы неправильно, то можете оказаться в ситуации, когда все данные находятся в одной секции, а остальные секции пусты. А перепланировка границ секций вручную — очень утомительная работа.

Поэтому секционированные по диапазонам значений ключа базы данных, такие как HBase и RethinkDB, создают секции динамически. Когда размер секции перерастает заданный (в HBase размер по умолчанию равен 10 Гбайт), она разбивается на две секции, таким образом, чтобы примерно половина данных оказалась в одной, а половина — в другой [26]. И наоборот, если значительное количество данных удаляется и секция становится меньше определенного порогового значения, то ее можно слить с соседней секцией. Этот процесс аналогичен происходящему на верхнем уровне B-дерева (см. подраздел «B-деревья» раздела 3.1).

Каждая секция относится к какому-либо узлу, причем к одному узлу может относиться несколько секций, как в случае с фиксированным количеством секций. После разбиения большой секции одну из ее частей можно переместить в другой узел, чтобы сбалансировать нагрузку. В случае СУБД HBase перемещение файлов секций происходит в используемой ею распределенной файловой системе HDFS [3].

Преимуществом динамического секционирования является адаптация количества секций к общему объему данных. Если данных немного, то достаточно небольшого количества секций и накладные расходы невелики; в противном случае размер каждой секции ограничивается заранее задаваемым максимумом [23].

Однако следует понимать, что пустая БД начинается с одной секции, ведь никакой *априорной* информации, которая позволила бы задать границы секций, нет. Пока набор данных невелик — до момента разбиения первой секции, — все операции записи обрабатываются одним узлом, в то время как все остальные узлы простаивают. Чтобы минимизировать влияние этого фактора, HBase и MongoDB позволяют задавать начальный набор секций для пустой БД (это называется *предварительным разбиением* (pre-splitting)). В случае секционирования по диапазонам значений ключа для предварительного разбиения нужно заранее знать ожидаемое распределение ключа [4, 26].

Динамическое секционирование подходит не только для данных, секционированных по диапазонам значений ключа, но и может с равным успехом использоваться для хеш-секционированных данных.

Секционирование пропорционально количеству узлов

При динамическом секционировании количество секций пропорционально размеру набора данных, поскольку размеры всех секций поддерживаются с помощью процессов разбиения и слияния в пределах фиксированных минимума и максимума. С другой стороны, при фиксированном количестве секций, размер каждой секции пропорционален размеру набора данных. В обоих случаях количество секций не зависит от количества узлов.

При третьем возможном варианте, используемом в СУБД Cassandra и Ketama, количество секций пропорционально количеству узлов — другими словами, *на каждый узел* приходится фиксированное количество секций [23, 27, 28]. В этом случае размеры всех секций растут пропорционально размеру набора данных, если количество узлов остается неизменным, но при увеличении количества узлов секции снова уменьшаются. Поскольку больший объем данных обычно требует для хранения большего количества узлов, такой подход обеспечивает практически постоянные размеры отдельных секций.

При добавлении в кластер новый узел случайным образом выбирает фиксированное количество существующих секций для разбиения, после чего забирает по одной половине каждой из разбиваемых секций, оставляя вторые половины на месте. Получаемые в результате разбиения могут оказаться неравными, но при усреднении по большому количеству секций (по умолчанию в СУБД Cassandra 256 секций на узел) новый узел забирает у других узлов равную долю общей нагрузки. В Cassandra 3.0 появился другой алгоритм перебалансировки, позволяющий избежать неравных разбиений [29].

Выбор границ секций случайным образом требует использования хеш-секционирования (чтобы выбрать границы на основе диапазона чисел, возвращаемых хеш-функцией). Безусловно, такой подход лучше всего соответствует исходному определению согласованного хеширования [7] (см. врезку «Согласованное хеширование» в подразделе «Секционирование по хешу ключа» раздела 6.2). Новейшие хеш-функции позволяют достичь того же результата с меньшей избыточностью метаданных [8].

Эксплуатация: автоматическая или ручная перебалансировка

Мы сознательно обошли стороной один важный вопрос, касающийся перебалансировки: происходит ли перебалансировка автоматически, или ее нужно выполнять вручную?

Существует множество градаций между полностью автоматической перебалансировкой и полностью ручной. В первом случае система сама определяет, когда переместить секцию из одного узла в другой, без всякого вмешательства администратора, во втором — распределение секций по узлам явным образом задается администратором и меняется только по его явному указанию. Например, в Couchbase, Riak и Voldemort распределение секций генерируется автоматически, но вступает в силу только после подтверждения от администратора.

Полностью автоматическая перебалансировка удобна тем, что уменьшает объем работ по обслуживанию системы. Однако ее результаты могут оказаться неожиданными. Перебалансировка — операция с большими накладными расходами, тре-

бующая изменения маршрутов запросов и перемещения большого объема данных из одного узла в другой. Если не выполнять этот процесс с большой осторожностью, то он может привести к перегрузке сети или узлов и ухудшению производительности других запросов во время перебалансировки.

Подобная автоматизация опасна в сочетании с автоматическим обнаружением отказов. Например, один из узлов перегружен и временно возвращает ответы на запросы слишком медленно. Другие узлы приходят к выводу: перегруженный узел отказал — и автоматически выполняют перебалансировку кластера для снятия с него нагрузки. Такое решение приводит к дополнительной нагрузке на этот и так перегруженный узел, другие узлы и сеть, что только ухудшает положение дел и потенциально приводит к каскадному сбою.

Исходя из вышесказанного присутствие человека в цикле перебалансировки может быть хорошей идеей. Это медленнее, чем полностью автоматический процесс, но позволит предотвратить операционные сюрпризы.

6.5. Маршрутизация запросов

Мы секционировали набор данных по нескольким узлам на ряде машин. Но остается открытым следующий вопрос: откуда клиент, выполняющий запрос, знает, к какому узлу ему нужно подключиться? При перебалансировке секций распределение их по узлам меняется. Необходим какой-то «наблюдатель сверху» за этими изменениями, который бы мог ответить клиенту на вопрос: по какому IP-адресу и к какому порту мне нужно подключиться, если я хочу прочитать или записать значение для ключа `foo`?

Это частный случай более общей задачи, называемой *обнаружением сервисов* (service discovery), относящейся не только к базам данных. Задача касается любого доступного программного обеспечения, предназначенного для обращения по сети, особенно стремящегося к высокой доступности (при работе на нескольких машинах в избыточной конфигурации). Многие компании создали собственные, предназначенные для внутреннего использования утилиты обнаружения сервисов, многие из которых были выпущены в качестве ПО с открытым исходным кодом [30].

На высоком уровне существует несколько различных способов решения этой задачи (показаны на рис. 6.7).

1. Разрешить клиентам обращаться к любому узлу (например, с помощью циклического балансировщика нагрузки). Если на этом узле случайно окажется секция, которая нужна для ответа на запрос, то он обработает запрос непосредственно, в противном случае перенаправит его соответствующему узлу, получит от него ответ и перепрaвит его клиенту.

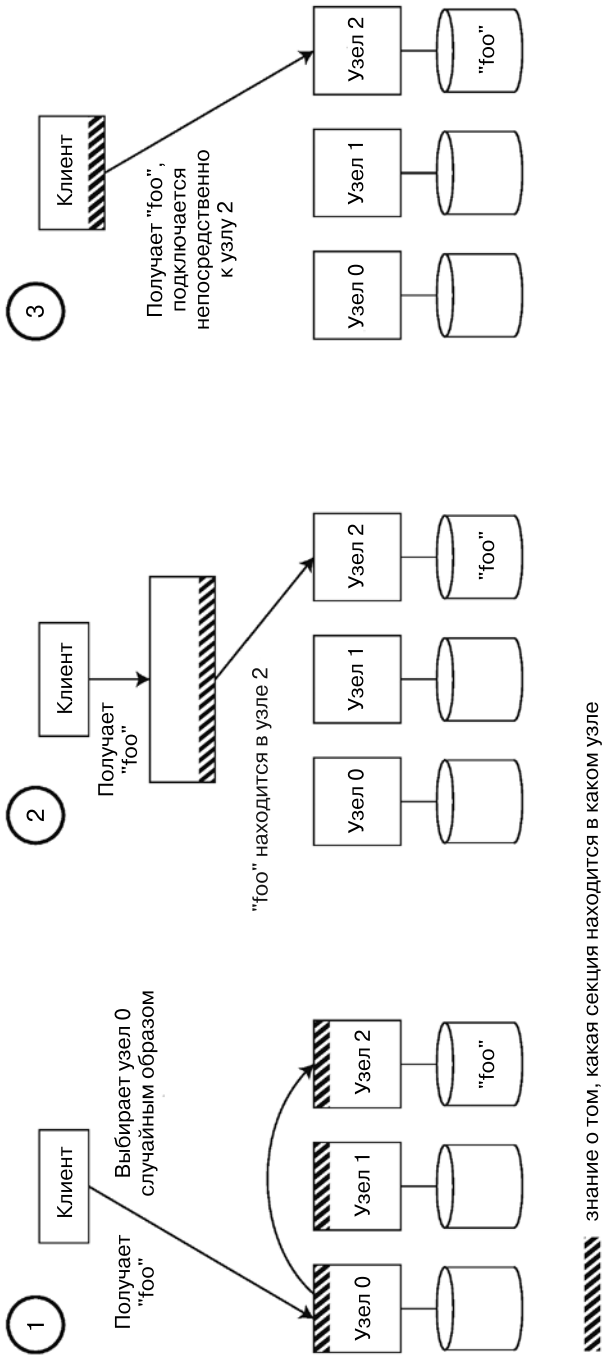


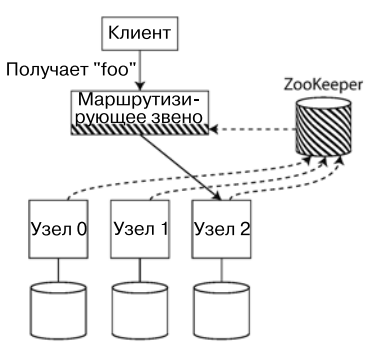
Рис. 6.7. Три различных способа маршрутизации запроса к нужному узлу

2. Отправлять все поступающие от клиентов запросы сначала маршрутизирующему звену, которое определяет, какой узел должен обрабатывать данный запрос, и переправляет его соответствующим образом. Это маршрутизирующее звено само не обрабатывает никаких запросов, а служит только учитывающим секции балансировщиком нагрузки.
3. Потребовать, чтобы клиенты учитывали секционирование и распределение секций по узлам. В этом случае клиент может подключаться непосредственно к соответствующему узлу, без всяких посредников.

Во всех случаях основная проблема формулируется следующим образом: откуда отвечающий за маршрутизацию компонент (это может быть один из узлов, маршрутизирующее звено или клиент) знает об изменениях в распределении секций по узлам?

Это непростая задача, ведь нужен консенсус всех участвующих сторон — в противном случае запросы будут отправляться не тем узлам и обрабатываться неправильно. Существуют протоколы для достижения консенсуса в распределенной системе, но реализовать их правильно непросто (см. главу 9).

Многие распределенные информационные системы используют для отслеживания этих метаданных кластера отдельный сервис координации, например ZooKeeper, как показано на рис. 6.8. Все узлы регистрируются в сервисе, который поддерживает актуальную карту соответствий секций узлам. Другие действующие лица, например маршрутизирующее звено или учитывающий секции клиент, могут подписываться на данную информацию в ZooKeeper. При каждой смене узла-владельца секции или добавлении/удалении узла сервис оповещает маршрутизирующее звено, так что маршрутизация остается актуальной.



Диапазон ключа	Секция	Узел	IP-Address
A-ak — Bayes	partition 0	node 0	10.20.30.100
Bayeu — Ceanothus	partition 1	node 1	10.20.30.101
Ceara — Deluc	partition 2	node 2	10.20.30.102
Delusion — Frenssen	partition 3	node 0	10.20.30.100
Freon — Holderlin	partition 4	node 1	10.20.30.101
Holderness — Krasnoje	partition 5	node 2	10.20.30.102
Krasnokamsk — Menadra	partition 6	node 0	10.20.30.100
Menage — Ottawa	partition 7	node 1	10.20.30.101
Otter — Rethimnon	partition 8	node 2	10.20.30.102
Reti — Solovets	partition 9	node 0	10.20.30.100
Solovyov — Truck	partition 10	node 1	10.20.30.101
Trudeau — Zywiec	partition 11	node 2	10.20.30.102

знание о том, какая секция находится на каком узле

Рис. 6.8. Применение ZooKeeper для отслеживания распределения секций по узлам

Например, база данных Espresso соцсети LinkedIn использует для управления кластером Helix [31] (который, в свою очередь, задействует ZooKeeper), реализуя маршрутизирующее звено так, как показано на рис. 6.8. Базы данных HBase, SolrCloud и Kafka тоже применяют ZooKeeper для отслеживания распределения секций. Архитектура MongoDB аналогична, но основана на собственной реализации *сервера конфигураций* и демонов *mongos* в качестве маршрутизирующего звена.

Cassandra и Riak применяют другой подход: используют для обмена сообщениями между узлами *gossip-протокол* для распространения всех изменений состояния кластера. Запросы можно отправлять любому узлу, который перенаправит их к узлу с соответствующей секцией (подход 1 на рис. 6.7). Эта модель усложняет узлы базы данных, но позволяет избежать зависимости от внешнего сервиса координации, такого как ZooKeeper.

Couchbase не производит перебалансировку автоматически, что упрощает его архитектуру. Обычная его конфигурация включает маршрутизирующее звено *moxi*, получающее информацию об изменениях маршрутизации от узлов кластера [32].

При использовании маршрутизирующего звена или отправке запросов случайному узлу клиентам все равно приходится выяснять IP-адреса для подключения. Они изменяются не так часто, как распределение секций по узлам, так что обычно достаточно задействовать для этой цели DNS.

Параллельное выполнение запросов. До сих пор мы акцентировали внимание на очень простых запросах, выполняющих чтение или запись значения для одного ключа (плюс фрагментированные запросы в случае секционированных по документам вторичных индексов). Это примерно соответствует уровню доступа, поддерживаемому большинством распределенных хранилищ данных NoSQL.

Однако реляционные базы данных с *массово-параллельной архитектурой* (massively parallel processing, MPP), часто используемые для аналитики, поддерживают гораздо более сложные типы запросов. Типичный запрос склада данных содержит несколько операций соединения, фильтрации, группировки и агрегирования. Оптимизатор запросов MPP разбивает этот сложный запрос на несколько стадий выполнения и секций, и многие из них можно выполнять параллельно в различных узлах кластера БД. Подобный параллелизм приносит особенную выгоду запросам, для которых требуется просмотр больших частей набора данных.

Быстрое параллельное выполнение запросов складов данных — отдельная тема, вызывающая большой коммерческий интерес вследствие важности аналитики для бизнеса. Мы обсудим некоторые методы параллельного выполнения запросов в главе 10. Более подробный обзор используемых в параллельных базах данных методик можно найти в источниках [1, 33].

6.6. Резюме

В этой главе мы рассмотрели различные методы секционирования большого набора данных на меньшие поднаборы. Секционирование требуется, когда размер набора более не позволяет хранить и обрабатывать его на одной машине.

Цель секционирования заключается в распределении нагрузки по данным и запросам равномерно по нескольким машинам, а также в том, чтобы избежать горячих точек (узлов с непропорционально высокой нагрузкой). Это требует выбора подходящей для набора данных схемы секционирования и перебалансировки секций при добавлении или удалении узлов из кластера.

Мы обсудили два основных подхода к секционированию.

- *Секционирование по диапазонам значений ключа*, при котором ключи сортируются и секция содержит все ключи, начиная с определенного минимума до определенного максимума. Преимущество сортировки состоит в возможности выполнять эффективные запросы по диапазонам; но если приложение часто обращается к расположенным близко (в соответствии с сортировкой) ключам, то возникает риск возникновения горячих точек.

  При этом подходе обычно производится динамическая перебалансировка секций с помощью разбиения диапазона на два поддиапазона в случае, когда секция становится слишком велика.
- *Хеш-секционирование*, при котором вычисляется хеш-функция каждого ключа и к каждой секции относится определенный диапазон хешей. Этот метод нарушает упорядоченность ключей, делая запросы по диапазонам неэффективными, но позволяет более равномерно распределять нагрузку.
- При хеш-секционировании часто заранее создается фиксированное количество секций, по нескольку для каждого узла, а при добавлении или удалении узлов между узлами часто перемещаются целые секции. При этом также можно использовать динамическое секционирование.

Возможны и гибридные подходы, например с составным ключом: применение одной части ключа для идентификации секции, а другой — для определения порядка сортировки.

Кроме того, мы обсудили взаимосвязь между секционированием и вторичными индексами. Последние тоже необходимо секционировать, для чего существуют два метода.

- *Секционирование индексов по документам* (локальные индексы). Вторичные индексы хранятся в одной секции с первичным ключом и значением. Это значит, что при операции записи нужно обновлять только одну секцию, но чтение вторичного индекса требует фрагментированного чтения по всем секциям.

- *Секционирование индексов по термам* (глобальные индексы), при котором вторичные индексы секционируются отдельно, с использованием индексированных значений. Элемент этого вторичного индекса может включать записи из всех секций первичного ключа. При записи документа приходится обновлять несколько секций вторичного индекса, однако результат чтения можно выдать из одной секции.

Наконец, мы обсудили методы маршрутизации запросов к соответствующей секции, начиная с простого, учитывающего секции балансировщика нагрузки и заканчивая сложными параллельными механизмами выполнения запросов.

Все секции умышленно работают практически независимо — это позволяет масштабировать секционированную базу данных на несколько машин. Однако остается вопрос с операциями, для которых требуется запись в несколько секций: например, что будет, если операция записи в одну секцию завершится успешно, а в другую — нет? Мы займемся этим вопросом в следующих главах.

6.7. Библиография

1. *DeWitt D. J., Gray J. N.* Parallel Database Systems: The Future of High Performance Database Systems // Communications of the ACM, volume 35, number 6, pages 85–98, June 1992 [Электронный ресурс]. — Режим доступа: http://www.cs.cmu.edu/~pavlo/courses/fall2013/static/papers/dewittgray92.pdf.
2. *George L.* HBase vs. BigTable Comparison. November 2009 [Электронный ресурс]. — Режим доступа: http://www.larsgeorge.com/2009/11/hbase-vs-bigtable-comparison.html.
3. The Apache HBase Reference Guide // Apache Software Foundation, 2014 [Электронный ресурс]. — Режим доступа: https://hbase.apache.org/book.html#book.
4. MongoDB, Inc.: New Hash-Based Sharding Feature in MongoDB 2.4. April 10, 2013 [Электронный ресурс]. — Режим доступа: https://www.mongodb.com/blog/post/new-hash-based-sharding-feature-in-mongodb-24.
5. *Lan I.* App Engine Datastore Tip: Monotonically Increasing Values Are Bad. January 25, 2011 [Электронный ресурс]. — Режим доступа: https://ikaisays.com/2011/01/25/app-engine-datastore-tip-monotonically-increasing-values-are-bad/.
6. *Kleppmann M.* Java's hashCode Is Not Safe for Distributed Systems. June 18, 2012 [Электронный ресурс]. — Режим доступа: http://martin.kleppmann.com/2012/06/18/java-hashcode-unsafe-for-distributed-systems.html.
7. *Karger D., Lehman E., Leighton T., et al.* Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web // 29th Annual ACM Symposium on Theory of Computing (STOC), pages 654–663,

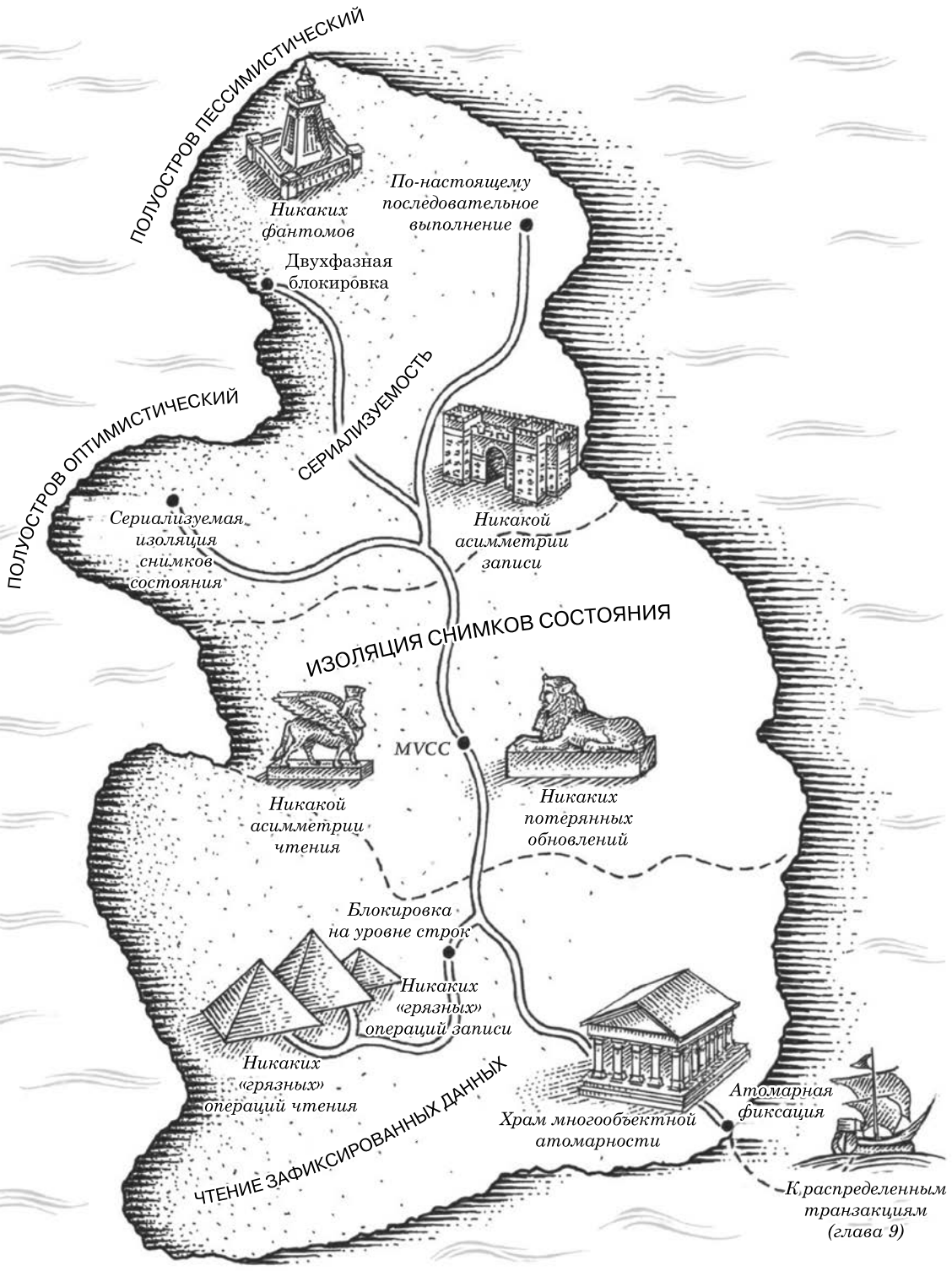
1997 [Электронный ресурс]. — Режим доступа: https://www.akamai.com/us/en/multimedia/documents/technical-publication/consistent-hashing-and-random-trees-distributed-caching-protocols-for-relieving-hot-spots-on-the-world-wide-web-technical-publication.pdf.

8. *Lamping J., Veach E.* A Fast, Minimal Memory, Consistent Hash Algorithm. June 2014 [Электронный ресурс]. — Режим доступа: https://arxiv.org/ftp/arxiv/papers/1406/1406.2294.pdf.
9. *Redmond E.* A Little Riak Book // Version 1.4.0, Basho Technologies, September 2013 [Электронный ресурс]. — Режим доступа: https://www.boomtownbingo.com/new-slots-sites.
10. Couchbase 2.5 Administrator Guide // Couchbase, Inc., 2014 [Электронный ресурс]. — Режим доступа: http://docs.couchbase.com/couchbase-manual-2.5/cb-admin/.
11. *Lakshman A., Malik P.* Cassandra — A Decentralized Structured Storage System // 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS), October 2009 [Электронный ресурс]. — Режим доступа: http://www.cs.cornell.edu/Projects/ladis2009/papers/Lakshman-ladis2009.PDF.
12. *Ellis J.* Facebook's Cassandra Paper, Annotated and Compared to Apache Cassandra 2.0. September 12, 2013 [Электронный ресурс]. — Режим доступа: http://docs.datastax.com/en/articles/cassandra/cassandrathenandnow.html.
13. Introduction to Cassandra Query Language // DataStax, Inc., 2014 [Электронный ресурс]. — Режим доступа: http://docs.datastax.com/en/cql/3.1/cql/cql_intro_c.html.
14. *Axon S.* 3 % of Twitter's Servers Dedicated to Justin Bieber. September 7, 2010 [Электронный ресурс]. — Режим доступа: http://mashable.com/2010/09/07/justin-bieber-twitter/#sFfmr17MhOqS.
15. Riak 1.4.8 Docs // Basho Technologies, Inc., 2014 [Электронный ресурс]. — Режим доступа: http://docs.basho.com/riak/1.4.8/.
16. *Low R.* The Sweet Spot for Cassandra Secondary Indexing. October 21, 2013 [Электронный ресурс]. — Режим доступа: http://www.wentnet.com/blog/?p=77.
17. *Tong Z.* Customizing Your Document Routing. June 3, 2013 [Электронный ресурс]. — Режим доступа: https://www.elastic.co/blog/customizing-your-document-routing.
18. Apache Solr Reference Guide // Apache Software Foundation, 2014 [Электронный ресурс]. — Режим доступа: https://lucene.apache.org/solr/guide/6_6/.
19. *Pavlo A.* H-Store Frequently Asked Questions. October 2013 [Электронный ресурс]. — Режим доступа: http://hstore.cs.brown.edu/documentation/faq/.

20. Amazon DynamoDB Developer Guide // Amazon Web Services, Inc., 2014 [Электронный ресурс]. — Режим доступа: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html.
21. *Klophaus R.* Difference Between 2I and Search // email to riak-users mailing list, October 25, 2011 [Электронный ресурс]. — Режим доступа: http://lists.basho.com/pipermail/riak-users_lists.basho.com/2011-October/006220.html.
22. *Burleson D. K.* Object Partitioning in Oracle. November 8, 2000 [Электронный ресурс]. — Режим доступа: http://www.dba-oracle.com/art_partit.htm.
23. *Evans E.* Rethinking Topology in Cassandra // ApacheCon Europe, November 2012 [Электронный ресурс]. — Режим доступа: https://www.slideshare.net/jericevans/virtual-nodes-rethinking-topology-in-cassandra.
24. *Kuć R.* Reroute API Explained. September 30, 2013 [Электронный ресурс]. — Режим доступа: http://elasticsearchserverbook.com/reroute-api-explained/.
25. Project Voldemort Documentation [Электронный ресурс]. — Режим доступа: http://www.project-voldemort.com/voldemort/.
26. *Soztutar E.* Apache HBase Region Splitting and Merging. February 1, 2013 [Электронный ресурс]. — Режим доступа: https://hortonworks.com/blog/apache-hbase-region-splitting-and-merging/.
27. *Williams B.* Virtual Nodes in Cassandra 1.2. December 4, 2012 [Электронный ресурс]. — Режим доступа: https://www.datastax.com/dev/blog/virtual-nodes-in-cassandra-1-2.
28. *Jones R.* libketama: Consistent Hashing Library for Memcached Clients. April 10, 2007 [Электронный ресурс]. — Режим доступа: https://www.metabrew.com/article/libketama-consistent-hashing-algo-memcached-clients.
29. *Lambov B.* New Token Allocation Algorithm in Cassandra 3.0. January 28, 2016 [Электронный ресурс]. — Режим доступа: https://www.datastax.com/dev/blog/token-allocation-algorithm.
30. *Wilder J.* Open-Source Service Discovery. February 2014 [Электронный ресурс]. — Режим доступа: http://jasonwilder.com/blog/2014/02/04/service-discovery-in-the-cloud/.
31. *Gopalakrishna K., Lu S., Zhang Z., et al.* Untangling Cluster Management with Helix // ACM Symposium on Cloud Computing (SoCC), October 2012 [Электронный ресурс]. — Режим доступа: https://915bbc94-a-62cb3a1a-s-sites.googlegroups.com/site/acm2012socc/helix_onecol.pdf?attachauth=ANoY7coIO0VJabzwZjxwDz6vxl-CHom-0DdLVzFn2Y2WohafZUBtEioUbLFnjZP4Uwfesf-Dn-Jg4iShzJvIa2CSAaIry0A5Ghsh6jhX6ixtwrQ22a4jm4zR1Iy0bZmCa_jEavgMdqItdK1GYDL3KplgfXffpikfVQFJ1-q5iduqBdnqm7Nm3I8HMzyXFCNK50G8-yG3EA9Qvb6pkEx_uG9hW7eXVn7uTA%3D%3D&attredirects=0.

32. Moxi 1.8 Manual // Couchbase, Inc., 2014 [Электронный ресурс]. — Режим доступа: http://docs.couchbase.com/moxi-manual-1.8/.

33. *Babu S., Herodotou H.* Massively Parallel Databases and MapReduce Systems // Foundations and Trends in Databases, volume 5, number 1, pages 1–104, November 2013 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/massively-parallel-databases-and-mapreduce-systems/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F206464%2Fdb-mr-survey-final.pdf.


ПОЛУОСТРОВ ПЕССИМИСТИЧЕСКИЙ
Никаких фантомов
По-настоящему последовательное выполнение
Двухфазная блокировка
ПОЛУОСТРОВ ОПТИМИСТИЧЕСКИЙ
СЕРИАЛИЗУЕМОСТЬ
Сериализуемая изоляция снимков состояния
Никакой асимметрии записи
ИЗОЛЯЦИЯ СНИМКОВ СОСТОЯНИЯ
MVCC
Никакой асимметрии чтения
Никаких потерянных обновлений
Блокировка на уровне строк
Никаких «грязных» операций записи
Никаких «грязных» операций чтения
ЧТЕНИЕ ЗАФИКСИРОВАННЫХ ДАННЫХ
Храм многообъектной атомарности
Атомарная фиксация
К распределенным транзакциям (глава 9)

7 Транзакции

> Некоторые авторы утверждают, что поддержка общей двухфазной фиксации транзакций обходится слишком дорого из-за возникающих проблем с производительностью и доступностью. Мы полагаем, что лучше уж разработчикам заниматься решением проблем с производительностью при возникновении узких мест из-за злоупотребления транзакциями, чем постоянно создавать код для обхода проблем, связанных с их отсутствием.
>
> *Джеймс Корбетт и др. Spanner: глобально-распределенная база данных от компании Google (2012)*

В суровой реальности информационных систем очень многое может пойти не так:

- программное или аппаратное обеспечение базы данных может отказать в любой момент (в том числе посередине операции записи);
- в любой момент может произойти фатальный сбой приложения (в том числе когда последовательность операций выполнена наполовину);
- разрывы сети могут неожиданно отрезать приложение от базы данных или один узел базы от другого;
- несколько клиентов могут производить операции записи одновременно, перезаписывая изменения друг друга;
- клиент может прочитать данные, которые не имеют смысла, поскольку были обновлены только частично;
- состояния гонки между клиентами могут привести к неожиданным ошибкам.

Надежная система должна уметь справляться со всеми этими сбоями и гарантировать, что они не приведут к внезапному и полному отказу всей системы. Однако реализация механизмов отказоустойчивости непроста. Она требует тщательного продумывания всех возможных сбоев и немалой толики тестирования, чтобы убедиться в работе решения.

Транзакции в течение десятилетий считались предпочтительным механизмом решения этой проблемы. Транзакция — способ группировки приложением нескольких операций записи и чтения в одну логическую единицу. По сути, все операции записи и чтения в ней выполняются как одна: вся транзакция или целиком выполняется успешно (с *фиксацией* изменений), или целиком завершается неудачно (с *прерыванием* и *откатом*). В первом случае приложение может спокойно попробовать выполнить ее еще раз. Транзакции значительно упрощают для приложения обработку ошибок, поскольку нет нужды заботиться о частичных отказах, то есть случаях, когда часть операций завершилась успешно, а часть — нет (неважно, по каким причинам).

Если вы в течение многих лет работаете с транзакциями, то они кажутся вам чем-то очевидным, но мы не станем считать их само собой разумеющимися. Транзакции не закон природы, они были созданы с определенной целью, а именно для *упрощения модели программирования* приложений, работающих с базами данных. Благодаря использованию транзакций приложение получает возможность игнорировать определенные сценарии ошибок и вопросы конкурентного доступа, поскольку вместо него этим занимается база (мы будем называть данный аспект *гарантиями функциональной безопасности*).

Не для всякого приложения нужны транзакции, и иногда лучше ослабить транзакционные гарантии или вообще отказаться от них (например, для повышения производительности или доступности). Некоторые из свойств функциональной безопасности достижимы и без транзакций.

Как определить, нужны ли транзакции? Чтобы ответить на этот вопрос, следует сначала точно выяснить, какие гарантии функциональной безопасности могут обеспечить транзакции и каких затрат они потребуют. Хотя транзакции кажутся простыми на первый взгляд, с ними связано множество тонких, но важных нюансов.

В этой главе мы рассмотрим немало примеров возможных проблем и изучим алгоритмы, которые используют базы данных для их предотвращения. Мы особенно глубоко рассмотрим вопрос управления конкурентным доступом, обсудим различные виды возникающих состояний гонки, а также реализацию в базах таких уровней изоляции, как *чтение зафиксированных данных* (read committed), *изоляция снимков состояния* (snapshot isolation) и *сериализуемость*, или *сериализуемые транзакции* (serializability).

Описываемое в настоящей главе относится как к одноузловым, так и к распределенным базам данных. В главе 8 мы обсудим подробнее трудности, возникающие только в распределенных системах.

7.1. Неустоявшаяся концепция транзакции

Практически все реляционные базы данных, да и некоторые нереляционные, поддерживают транзакции. Большинство из них следуют стилю появившейся в 1975 году первой базы данных SQL, System R компании IBM [1–3]. Хотя отдельные детали реализации поменялись, общая идея оставалась практически той же самой на протяжении прошедших 40 лет: поддержка транзакций в СУБД MySQL, PostgreSQL, Oracle, SQL Server и др. очень напоминает таковую в System R.

В конце 2000-х годов приобрели популярность нереляционные (NoSQL) базы данных. Их целью было улучшить существующее положение дел с реляционными БД с помощью новых моделей данных (см. главу 2), репликации (см. глава 5) и секционирования (см. главу 6). Транзакции оказались главной жертвой этого новшества: многие базы нового поколения полностью от них отказались или поменяли значение термина: теперь он стал у них означать намного более слабый набор функциональных гарантий, чем ранее [4].

Вместе с шумихой вокруг этого нового обильного урожая распределенных баз данных стало широко распространяться мнение, что транзакции — антоним масштабируемости и любой крупномасштабной системе необходимо отказаться от них ради сохранения хорошей производительности и высокой доступности [5, 6]. С другой стороны, производители БД иногда представляют транзакционные функциональные гарантии как обязательное требование для «серьезных приложений», оперирующих «ценными данными». Обе точки зрения — чистейшей воды преувеличение.

Истина не столь проста: как и любое другое техническое проектное решение, транзакции имеют свои достоинства и ограничения. Чтобы лучше разобраться в их плюсах и минусах, глубже заглянем в подробности предоставляемых транзакциями функциональных гарантий — как при обычной эксплуатации, так и в различных нестандартных (хотя и реалистичных) обстоятельствах.

Смысл аббревиатуры ACID

Обеспечиваемые транзакциями гарантии функциональной безопасности часто описываются известной аббревиатурой ACID (atomicity, consistency, isolation, durability — *атомарность*, *согласованность*, *изоляция* и *сохраняемость*). Она был придумана в 1983 году Тео Хэрдером (Theo Härder) и Андреасом Ройтером (Andreas Reuter) [7] в попытке создать четкую терминологию для механизмов обеспечения отказоустойчивости в базах данных.

Однако на практике реализации ACID в разных базах отличаются друг от друга. Например, как мы увидим, существуют серьезные различия в понимании термина *«изоляция»* (isolation) [8]. Идея в целом правильная, но дьявол, как обычно, кроется

в деталях. На сегодняшний день заявление о «совместимости системы с ACID» не дает четкого представления о предоставляемых гарантиях. К сожалению, ACID стал скорее термином из области маркетинга.

Системы, не соответствующие критериям ACID, иногда называются *BASE*, что расшифровывается следующим образом: *«как правило, доступна»* (**B**asically **A**vailable), *«гибкое состояние»* (**S**oft state) и *«конечная согласованность»* (**E**ventual consistency) [9]. Это понятие еще более расплывчатое, чем ACID. Похоже, единственное разумное определение BASE — «не ACID», то есть оно может значить практически все что угодно.

Посмотрим на определения атомарности, согласованности, изоляции и сохраняемости. Это позволит уточнить наши представления о транзакции.

Атомарность

В общем *атомарность* определяется как «невозможность разбиения на меньшие части». Данный термин означает немного различные вещи в разных отраслях информатики. Например, если в многопоточном программировании один из потоков выполняет атомарную операцию, это значит, что ни при каких обстоятельствах другие потоки не могут увидеть ее промежуточные результаты. Система может находиться или в состоянии, в котором она была до операции, или в том, в котором она окажется после, но не в каком-либо промежуточном.

Напротив, в контексте ACID атомарность *не связана* с конкурентным доступом. Этот термин не описывает, что происходит, когда несколько процессов пытаются обратиться к одним и тем же данным одновременно, поскольку относится к понятию *изоляции* (см. пункт «Изоляция» данного подраздела), то есть букве *I* в аббревиатуре ACID.

Атомарность в ACID описывает происходящее при сбое (например, фатальном сбое процесса, разрыве сетевого соединения, переполнении диска или нарушении одного из ограничений целостности) в процессе выполнения клиентом нескольких операций записи, в момент, когда выполнена лишь их часть. Если операции записи сгруппированы в атомарную транзакцию и ее не удается завершить (зафиксировать изменения) из-за сбоя, то она *прерывается* (abort) и базе данных приходится отбросить или откатить все уже выполненные в рамках этой транзакции операции записи.

При возникновении ошибки во время выполнения нескольких изменений без атомарности было бы сложно понять, какие из них вступили в действие. Приложение способно попытаться выполнить их снова, но здесь возникает риск выполнения одних и тех же изменений дважды; это может привести к дублированию или к ошибкам в них. Атомарность упрощает задачу: если транзакция была прервана, то

приложение может быть уверено, что ничего не было изменено и можно безопасно повторить изменения.

Возможность прервать транзакцию при ошибке и игнорировать все ее операции записи — отличительная черта атомарности ACID. Вероятно, слово *«прерываемость»* подошло бы лучше, но мы будем использовать термин *«атомарность»* как общепринятый.

Согласованность

Слово *«согласованность»* ужасно перегружено.

- В главе 5 мы обсуждали *согласованность реплик* и вопрос *конечной согласованности*, возникающий в асинхронно реплицируемых системах (см. раздел 5.2).
- *Согласованное хеширование* — метод секционирования, используемый в некоторых системах для перебалансировки (см. врезку «Согласованное хеширование» в подразделе «Секционирование по хешу ключа» раздела 6.2).
- В теореме CAP (см. главу 9) слово *«согласованность»* используется для обозначения *линеаризуемости* (linearizability) (см. раздел 9.2).
- В контексте ACID под *согласованностью* понимается то, что база данных находится, с точки зрения приложения, в «хорошем состоянии».

К сожалению, одно и то же слово применяется как минимум в четырех различных смыслах.

Идея согласованности в смысле ACID состоит в том, что определенные утверждения относительно данных *(инварианты)* должны всегда оставаться справедливыми — например, в системе бухгалтерского учета кредит всегда должен сходиться с дебетом по всем счетам. Если транзакция начинается при допустимом (в соответствии с этими инвариантами) состоянии базы данных и любые производимые во время транзакции операции записи сохраняют это свойство, то можно быть уверенными, что система всегда будет удовлетворять инвариантам.

Однако подобная идея согласованности зависит от точки зрения приложения на инварианты, и формулировать транзакции таким образом, чтобы сохранялась согласованность, — обязанность приложения. База данных не в состоянии это гарантировать, так как не сможет помешать записать «плохие» данные, нарушающие условия инвариантов. Она способна проверять некоторые специальные виды инвариантов, например, с помощью ограничений внешних ключей или ограничений уникальности. Однако в целом допустимость или недопустимость данных определяется приложением — БД только обеспечивает хранение.

Атомарность, изоляция и сохраняемость — свойства базы данных, в то время как согласованность (в смысле ACID) — свойство приложения. Оно может полагаться

на свойства атомарности и изоляции базы данных, чтобы обеспечить согласованность, но не на одну только базу. Следовательно, букве C на самом деле не место в аббревиатуре ACID[1].

Изоляция

К большинству баз данных обращается одновременно несколько клиентов. Это не вызывает проблем, когда они читают и записывают в различные части базы данных. Но если они обращаются к одним и тем же записям базы, то могут возникнуть проблемы конкурентного доступа (состояния гонки).

На рис. 7.1 представлен простой пример подобной проблемы. Допустим, два клиента одновременно увеличивают значение счетчика, хранимого в базе данных. Каждый из них должен прочитать текущее значение, добавить 1 и записать новое значение обратно (если в базе нет встроенной операции инкремента). На рис. 7.1 значение счетчика должно увеличиться с 42 до 44, поскольку произошло два его приращения, но на самом деле оно дошло только до 43 из-за состояния гонки.

Изоляция в смысле ACID означает, что конкурентно выполняемые транзакции изолированы друг от друга — они не могут помешать друг другу. Классические учебники по базам данных понимают под изоляцией *сериализуемость* (serializability). То есть каждая транзакция выполняется так, будто она единственная во всей базе. БД гарантирует, что результат фиксации транзакций такой же, как если бы они выполнялись *последовательно* (serially, одна за другой), хотя в реальности они могут выполняться конкурентно [10].

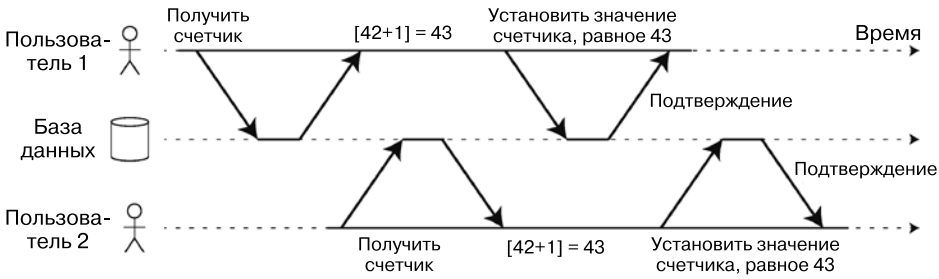


Рис. 7.1. Состояние гонки между двумя клиентами, конкурентно увеличивающими значение счетчика

Однако на практике сериализуемые транзакции используются редко, поскольку влекут за собой проблемы с производительностью. В некоторых популярных базах данных, например в Oracle 11g, они вообще не реализованы. В Oracle есть уровень

[1] Джо Хеллерштейн (Joe Hellerstein) заметил, что буква C была добавлена в ACID в статье Хэрдера и Ройтера [7] для красоты аббревиатуры и не считалась в то время чем-то важным.

изоляции транзакций SERIALIZABLE, но он фактически реализует то, что называется *изоляцией снимков состояния* и представляет собой более слабую гарантию, чем сериализуемость [8, 11]. Мы рассмотрим изоляцию снимков состояния и другие формы изоляции в разделе 7.2.

Сохраняемость

Задача СУБД — предоставить надежное место для хранения данных. *Сохраняемость* (durability) — обязательство базы не терять записанных (успешно зафиксированных) транзакций данных, даже в случае сбоя аппаратного обеспечения или фатального сбоя самой БД.

В одноузловой базе сохраняемость обычно означает запись данных на энергонезависимый носитель информации, например жесткий диск или SSD. Она обычно подразумевает также наличие журнала упреждающей записи или чего-то в этом роде (см. пункт «Обеспечение надежности B-деревьев» подраздела «B-деревья» раздела 3.1), обеспечивающего возможность восстановления в случае повреждения структуры данных на диске. В реплицируемой БД сохраняемость может означать, что данные были успешно скопированы на некоторое количество узлов. Для обеспечения гарантии сохраняемости база должна дожидаться завершения этих операций записи или репликаций, прежде чем сообщать об успешной фиксации транзакции.

Как обсуждалось в разделе 1.2, абсолютная надежность недостижима: если все жесткие диски и резервные копии будут уничтожены одновременно, то база данных, безусловно, не сможет никак вас спасти.

Репликация и сохраняемость

Исторически сохраняемость означала запись на магнитную архивную ленту. Затем ее начали интерпретировать как запись на диск или SSD. В последнее время этот термин начал означать репликацию. Какая же из реализаций лучше?

На самом деле ни одна из них не идеальна.

- При записи на диск данные оказываются недоступны в случае поломки компьютера до его починки или переноса жесткого диска на другой компьютер. В реплицируемых системах данные остаются доступны.
- Коррелированный сбой — отключение питания или ошибка, приводящая к сбою любого из узлов на конкретных входных данных, — может вывести из строя все реплики сразу (см. раздел 1.2), приведя к потере всех данных, хранящихся в этот момент только в оперативной памяти. Запись данных на жесткий диск, следовательно, сохраняет свое значение и для размещаемых в оперативной памяти БД.
- В асинхронной реплицируемой системе может происходить потеря последних операций записи при внезапной недоступности ведущего узла (см. подраздел «Перебои в обслуживании узлов» раздела 5.1).

- При внезапном отключении питания было отмечено, что, в частности, SSD иногда нарушают ожидаемые от них гарантии: не гарантируется корректная работа даже команды `fsync` [12]. В прошивках жестких дисков могут быть ошибки, как и в любом другом программном обеспечении [13, 14].
- Малозаметные взаимодействия между подсистемой хранения и реализацией файловой системы могут приводить к труднообнаруживаемым ошибкам и повреждению файлов на диске при сбое [15, 16].
- Данные на диске могут постепенно незаметно портиться [17]. А если они в какой-то момент оказались испорчены, то реплики и дальнейшие резервные копии тоже будут повреждены. В этом случае необходимо попытаться восстановить данные из более ранней резервной копии.
- Одно исследование твердотельных накопителей показало, что в 30–80 % SSD за первые четыре года эксплуатации появляется хотя бы один сбойный блок [18]. У магнитных жестких дисков частота появления сбойных блоков ниже, зато выше частота полных отказов, чем у SSD.
- При отключении от питания SSD через несколько недель начинает терять данные, в зависимости от температуры [19].

На практике не существует ни одного метода, дающего абсолютную гарантию сохраняемости данных. Существуют только различные методики снижения рисков, включая запись на диск, репликацию на удаленные машины и выполнение резервных копий, которые можно и нужно комбинировать. Как и всегда, к любым теоретическим «гарантиям» следует относиться с долей здорового скептицизма.

Однообъектные и многообъектные операции

Резюмируя: в ACID понятия атомарности и изоляции характеризуют действия, которые должна предпринимать база данных в случае выполнения клиентом нескольких операций записи в одной транзакции.

- *Атомарность*. Если посередине последовательности операций записи происходит ошибка, то транзакцию необходимо прервать, а выполненные до того момента операции аннулировать. Другими словами, база данных ограждает вас от забот о частичных отказах, предоставляя гарантию типа «все или ничего».
- *Изоляция*. Конкурентно выполняемые транзакции не должны мешать друг другу. Например, если одна транзакция выполняет несколько операций записи, то другая должна видеть или все их результаты, или никакие, но не какое-то подмножество.

Эти определения предполагают, что необходимо модифицировать несколько объектов (строк, документов или записей) одновременно. Подобные *многообъектные транзакции* часто оказываются нужны для обеспечения синхронизации нескольких элементов данных. На рис. 7.2 демонстрируется пример, взятый из приложения

электронной почты. Для отображения пользователю количества не прочитанных им сообщений можно выполнить примерно такой запрос:

```
SELECT COUNT(*) FROM emails WHERE recipient_id = 2 AND unread_flag = true
```

Однако может оказаться, что он работает слишком медленно в случае большого количества сообщений электронной почты и вы решите хранить количество непрочитанных сообщений в отдельном поле (один из видов денормализации). Теперь при каждом входящем сообщении придется увеличивать также значение счетчика непрочитанных сообщений, а когда оно помечается как прочитанное — уменьшать его.

На рис. 7.2 пользователь 2 сталкивается с аномалией: список в почтовом ящике демонстрирует непрочитанное сообщение, а счетчик показывает отсутствие таковых сообщений, поскольку увеличение его значения еще не произошло[1]. Изоляция могла бы предотвратить эту проблему, гарантировав, что пользователь 2 увидит либо и вставленное сообщение электронной почты, и обновленное значение счетчика, либо ни то ни другое, но не какой-то несогласованный средний вариант.

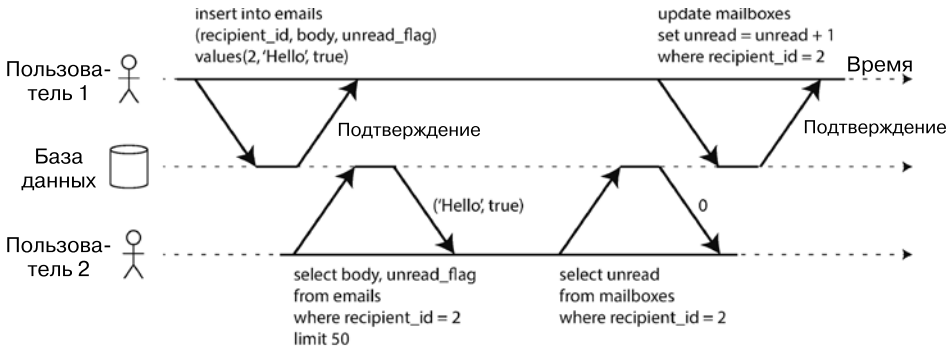


Рис. 7.2. Нарушение изоляции: одна транзакция читает результаты незафиксированных операций записи другой транзакции («грязная» операция записи)

Рисунок 7.3 демонстрирует, зачем нужна атомарность: если во время транзакции происходит ошибка, то содержимое почтового ящика и счетчик непрочитанных сообщений могут оказаться рассогласованными. Атомарная же транзакция прерывается в случае невозможности обновления счетчика, и происходит откат вставки сообщения электронной почты.

Для многообъектных транзакций необходимо знать, какие операции записи и чтения относятся к одной транзакции. В реляционных базах данных это можно узнать из TCP-соединения клиента с сервером БД: считается, что все находящееся между

[1] Можно поспорить, что неправильное значение счетчика в приложении электронной почты не особенно серьезная проблема. Но представьте баланс счета клиента вместо счетчика непрочитанных сообщений и транзакцию оплаты вместо сообщения электронной почты.

операторами `BEGIN TRANSACTION` и `COMMIT` конкретного соединения относится к одной транзакции[1].



Рис. 7.3. Атомарность гарантирует, что в случае возникновения ошибки все ранее выполненные операции записи данной транзакции откатываются во избежание несогласованности состояния базы данных

С другой стороны, во множестве нереляционных БД нет возможности подобной группировки операций. Даже наличие многообъектного API (например, в хранилищах данных типа «ключ — значение» могут существовать многообъектные операции `put`, обновляющие сразу несколько ключей за одну операцию) не обязательно означает транзакционность: команда может выполниться успешно на одних ключах и неудачно — на других, в результате чего база данных окажется в наполовину обновленном состоянии.

Однообъектные операции записи

Атомарность и изоляция также применимы при обновлении одного объекта. Например, допустим, что вы записываете в базу данных JSON-документ размером 20 Кбайт.

- При разрыве сетевого соединения после отправки первых 10 Кбайт будут ли сохранены в базе эти 10 Кбайт, синтаксический разбор которых невозможен?
- При сбое питания посередине перезаписи базой данных предыдущего значения на диске не перемешаются ли старое и новое значения?
- Увидит ли частично обновленное значение другой клиент, читающий данный документ во время операции записи?

Эти вопросы могут оказаться весьма сложными, так что подсистемы хранения практически всегда стараются обеспечивать атомарность и изоляцию на уровне отдельных объектов (например, пар «ключ — значение») в каждом узле. Атомарность

[1] Это неидеальное решение. В случае разрыва TCP-соединения транзакцию приходится прерывать. Если разрыв произойдет после просьбы клиента зафиксировать изменения, но до того, как сервер подтвердил выполнение, клиент не будет знать, была ли транзакция зафиксирована. Для решения данной проблемы диспетчер транзакций может группировать операции по уникальному идентификатору транзакции, не связанному с конкретным TCP-соединением. Мы вернемся к этому вопросу в подразделе «Сквозные аргументы в базе данных» раздела 12.3.

можно обеспечить с помощью журнала для восстановления после сбоев (см. пункт «Обеспечение надежности B-деревьев» подраздела «B-деревья» раздела 3.1), а изоляцию — путем блокировок отдельных объектов (благодаря которым обращаться к объекту одновременно может только один поток выполнения).

Отдельные базы данных также предоставляют более сложные атомарные операции, например операцию атомарного инкремента[1]; благодаря ей становится не нужен цикл чтения-изменения-записи, показанный на рис. 7.1. Столь же популярна операция сравнения с обменом, при которой операция записи разрешается, только когда значение не меняется конкурентно кем-то другим (см. пункт «Сравнение с обменом» подраздела «Предотвращение потери обновлений» раздела 7.2).

Подобные однообъектные операции полезны, поскольку предотвращают потерю обновлений в случаях, когда несколько клиентов пытаются конкурентно выполнить операцию записи в один объект (см. подраздел «Предотвращение потери обновлений» раздела 7.2). Однако они не являются транзакциями в обычном смысле этого слова. В маркетинговых целях о сравнении с обменом и других однообъектных операциях говорят как об «облегченных операциях» или даже как о ACID [20–22], но такая терминология ошибочна. Под транзакцией обычно понимается механизм группировки нескольких операций над несколькими объектами в одну единицу выполнения.

Востребованность многообъектных транзакций

Многие распределенные хранилища данных отказались от многообъектных транзакций по причине сложностей в реализации при работе с несколькими секциями, а также в силу того, что они могут оказаться только помехой в ряде сценариев выполнения, при которых необходима высокая доступность или производительность. Однако никаких существенных препятствий для использования транзакций в распределенных базах данных нет, и мы обсудим реализации распределенных транзакций в главе 9.

Но нужны ли нам вообще многообъектные транзакции? Нельзя ли реализовать любое приложение на основе только модели данных типа «ключ — значение» и однообъектных операций?

Существуют сценарии использования, при которых однообъектных вставок, обновлений и удалений вполне достаточно. Однако во многих сценариях необходимо согласование операций записи в несколько различных объектов.

- В реляционной модели данных строки в таблицах часто ссылаются на строки в других таблицах с помощью внешних ключей (подобно тому как в графоподобной модели вершины могут быть соединены ребрами с другими вершинами).

[1] Строго говоря, в термине «атомарный инкремент» (atomic increment) слово «атомарный» используется в смысле многопоточного программирования. В контексте ACID его следовало бы скорее называть изолированным или сериализуемым инкрементом. Но это уже придирки.

Многообъектные транзакции гарантируют, что эти ссылки останутся действующими: при вставке нескольких записей, ссылающихся друг на друга, внешние ключи должны быть правильными и актуальными, иначе данные потеряют смысл.

- В документной модели данных поля, которые должны обновляться совместно, часто находятся в одном документе, рассматриваемом как единый объект, так что при обновлении отдельного документа не нужны никакие многообъектные транзакции. Однако документоориентированные БД, в которых нет функциональности соединений, создают условия для денормализации (см. подраздел «Реляционные и документоориентированные базы данных сегодня» раздела 2.1). А обновление денормализованной информации, как показано на рис. 7.2, требует обновления нескольких документов за один проход. В этом случае транзакции оказываются очень удобны для предотвращения рассогласования денормализованных данных.
- В БД со вторичными индексами (а это почти все базы, кроме чистых хранилищ данных типа «ключ — значение») при каждом изменении значения необходимо обновлять индексы. Последние представляют собой различные объекты БД с точки зрения транзакций: например, без изоляции транзакций запись может встречаться в одном индексе и не встречаться в другом, поскольку второй индекс еще не был обновлен.

Все же подобные приложения можно реализовать и без транзакций. Однако без атомарности обработка ошибок значительно усложняется, а отсутствие изоляции способно привести к проблемам конкурентного доступа. Мы обсудим эти вопросы в разделе 7.2 и изучим альтернативные подходы в главе 12.

Обработка ошибок и прерывание транзакций

Отличительная особенность транзакций — возможность их прерывания и безопасного повторного выполнения в случае возникновения ошибки. На этом принципе построены базы данных ACID: при возникновении риска нарушения гарантий атомарности, изоляции или сохраняемости БД скорее полностью отменит транзакцию, чем оставит ее незавершенной.

Но не все системы следуют этой стратегии. В частности, хранилища данных, использующие репликацию без ведущего узла (см. раздел 5.4), работают более или менее на основе принципа «лучшее из возможного». Он формулируется следующим образом: «База данных делает все, что может, и при столкновении с ошибкой не станет откатывать уже выполненные действия», поэтому восстановление после ошибок является обязанностью приложения.

Ошибки неизбежны, но многие разработчики программ склонны думать только о хорошем, вместо того чтобы углубляться в детали обработки ошибок. Например, популярные фреймворки объектно-реляционного отображения (ORM), такие как ActiveRecord для Rails и Django, не повторяют попытку выполнения прерванных

транзакций — ошибка в них обычно приводит к «всплытию» исключения по стеку, так что все введенные пользователем данные теряются, а сам он получает сообщение об ошибке. Стыд и позор, ведь весь смысл прерывания транзакций как раз в обеспечении возможности безопасного их повторения.

Хотя повторение прерванных транзакций — простой и эффективный механизм обработки ошибок, он неидеален.

- В случае, когда транзакция была выполнена успешно, но произошел сбой сети при подтверждении клиенту ее успешной фиксации (вследствие чего клиент думает, что она завершилась неудачей), повтор приведет к выполнению этой транзакции дважды — если у вас не предусмотрен дополнительный механизм дедупликации на уровне приложения.
- Если причина ошибки — в перегруженности, то повтор транзакции только усугубит проблему. Во избежание подобных циклов обратной связи можно ограничить количество повторов, воспользоваться экспоненциальной отсрочкой отправки и обрабатывать связанные с перегруженностью ошибки не так, как все остальные (при условии, что это возможно, конечно).
- Имеет смысл повторять выполнение транзакций только для вре́менных ошибок (происходящих, например, из-за взаимной блокировки, нарушения изоляции, вре́менных проблем с сетью или восстановления после сбоя). Попытка повтора выполнения при постоянной ошибке (допустим, при нарушении ограничения) бессмысленна.
- Если у транзакции есть побочные действия вне базы данных, то они могут выполняться даже в случае ее прерывания. Например, вряд ли вы захотите повторять отправку сообщения электронной почты при каждой попытке повтора транзакции. Для гарантии того, что несколько различных систем будут фиксировать изменения или прерывать транзакцию только все вместе, может оказаться полезна двухфазная фиксация транзакции (мы обсудим ее в подразделе «Атомарная и двухфазная фиксация (2PC)» раздела 9.4).
- При сбое клиентского процесса во время повтора попытки выполнения все данные, которые он пытается записать в базу, теряются.

7.2. Слабые уровни изоляции

Транзакции, не затрагивающие одних и тех же данных, могут спокойно выполняться конкурентно, поскольку друг от друга не зависят. Проблемы конкурентного доступа (состояния гонки) возникают, только если одна транзакция читает данные, модифицируемые в этот момент другой, или две транзакции пытаются одновременно модифицировать одни и те же данные.

Ошибки конкурентного доступа трудно обнаружить при тестировании, поскольку они возникают только при проблемах с хронометражем. Подобные сложности могут проявляться очень редко, и воспроизвести их непросто. О конкурентном

доступе также очень сложно рассуждать, особенно когда речь идет о большом приложении, где непонятно, какие из других фрагментов кода могут обращаться к БД. Разработка приложений достаточно сложна и при работе лишь одного пользователя одновременно, а несколько совместно работающих пользователей еще более все усложняют, поскольку значение любого элемента данных может в любой момент неожиданно поменяться.

Поэтому базы данных долгое время пытались инкапсулировать вопросы конкурентного доступа от разработчиков приложений путем *изоляции транзакций* (transaction isolation). Теоретически изоляция должна была облегчить жизнь разработчиков, которые смогли бы сделать вид, что никакого конкурентного выполнения не происходит: *сериализуемая* изоляция означает гарантию базой данных такого режима выполнения транзакций, как будто они выполняются *последовательно* (то есть по одной, без всякого конкурентного доступа).

На практике, к сожалению, с изоляцией не все так просто. Затраты на сериализуемую изоляцию довольно высоки, и многие базы данных не согласны платить столь высокую цену [8]. Так что многие системы часто задействуют более слабые уровни изоляции, защищающие от *части* проблем конкурентного доступа, а не от всех. Разобраться в сути этих уровней изоляции гораздо труднее, и они могут приводить к возникновению гораздо более сложных ошибок, но тем не менее на практике используются [23].

Связанные с конкурентным доступом ошибки, обоснованные слабой изоляцией транзакций, отнюдь не только теоретическая проблема. Они вызывали немалые денежные потери [24, 25], приводили к расследованиям финансовых аудиторов [26] и становились причиной порчи пользовательских данных [27]. Частый комментарий при возникновении таких проблем — «Задействуйте базу данных ACID, если работаете с финансовыми данными!» — попадает пальцем в небо. Многие популярные реляционные БД (обычно считающиеся ACID) применяют слабую изоляцию, и поэтому не факт, что они бы спасли от этих ошибок.

Вместо того чтобы слепо полагаться на свои инструменты, необходимо хорошо разобраться в существующих проблемах конкурентного доступа и научиться их предотвращать. После этого мы сможем создавать надежные и безошибочные приложения на основе имеющихся инструментов.

В данном разделе мы рассмотрим несколько слабых (несериализуемых) уровней изоляции, используемых на практике, и обсудим подробно возможные состояния гонки, чтобы вы могли решить для себя, какой уровень лучше всего подходит для вашего приложения. Затем подробно обсудим сериализуемость (см. раздел 7.3). Обсуждать уровни изоляции мы будем без лишнего формализма, на примерах. Строгие определения и анализ свойств этих уровней изоляции вы можете найти в соответствующей научной литературе [28–30].

Чтение зафиксированных данных

Самый базовый уровень изоляции транзакций — *чтение зафиксированных данных*[1]. Он обеспечивает две гарантии.

- При чтении из БД клиент видит только зафиксированные данные (никаких *«грязных»* операций чтения).
- При записи в БД можно перезаписывать только зафиксированные данные (никаких *«грязных»* операций записи).

Обсудим эти две гарантии подробнее.

Никаких «грязных» операций чтения

Представьте, что транзакция записала какие-то данные в базу, но еще не была зафиксирована или была прервана. Может ли другая транзакция увидеть эти незафиксированные данные? Если да, то такая операция чтения называется *«грязной»* (dirty read) [2].

Выполняемые при уровне изоляции транзакций *read committed* (чтение зафиксированных данных) транзакции должны предотвращать «грязные» операции чтения. Это значит, что любые операции записи, выполняемые транзакцией, становятся видны другим транзакциям только после фиксации данной (после чего становятся видимы результаты сразу всех ее операций записи). Это показано на рис. 7.4, на котором пользователь 1 задает значение $x = 3$, а пользователь 2 *продолжает* получать в ответ старое значение 2 до тех пор, пока пользователь 1 не выполнит фиксации изменений.

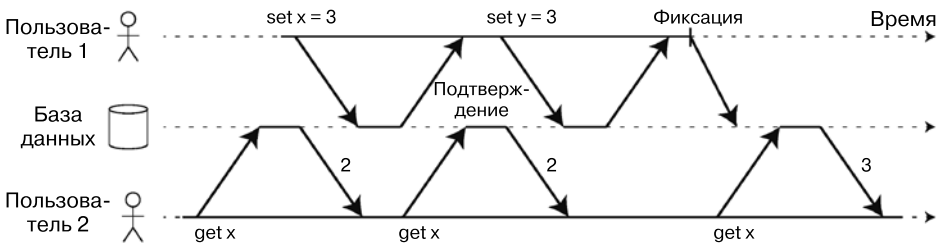


Рис. 7.4. Никаких «грязных» операций чтения: пользователь 2 видит новое значение x только после фиксации результатов транзакции, выполняемой пользователем 1

[1] Некоторые БД поддерживают еще более слабый уровень изоляции — чтение незафиксированных данных (read uncommitted). Он предотвращает «грязные» операции записи, но не «грязные» операции чтения.

Приведу несколько причин, по которым следует избегать «грязных» операций чтения.

- Когда в транзакции необходимо обновить значения нескольких объектов, «грязная» операция чтения означает, что другие транзакции могут увидеть часть обновлений, но не все. Например, на рис. 7.2 пользователь видит новое непрочитанное сообщение электронной почты, но не видит значения обновленного счетчика. Это «грязная» операция чтения сообщения электронной почты. База данных в частично обновленном состоянии представляет собой сбивающее с толку пользователей зрелище и может привести к принятию другими транзакциями неправильных решений.
- В случае прерывания транзакции необходимо откатить все уже выполненные ею операции записи (как показано на рис. 7.3). Если база допускает «грязные» операции чтения, то транзакции могут видеть данные, которые потом подвергнутся откату, то есть так и не зафиксированные в БД. От обсуждения последствий этого голова идет кругом.

Никаких «грязных» операций записи

Каково развитие событий, если две транзакции попытаются конкурентно обновить какой-либо объект в базе данных? Неизвестно, в каком порядке будут происходить операции записи, но обычно предполагается, что более поздняя операция записи перезаписывает значения более ранней.

Однако что произойдет, если более ранняя операция записи представляет собой часть еще не зафиксированной транзакции, вследствие чего более поздняя транзакция перезаписывает незафиксированное значение? Это называется *«грязной» операцией записи* [28]. Транзакции, выполняемые на уровне изоляции чтения зафиксированных данных, обязаны предотвращать такие операции, обычно путем откладывания второй операции записи до того момента, когда транзакция первой операции будет зафиксирована или прервана.

Благодаря предотвращению «грязных» операций чтения этот уровень изоляции лишен некоторых проблем конкурентного доступа.

- Если транзакция обновляет несколько объектов, «грязные» операции записи могут привести к нежелательным последствиям. Например, рассмотрим рис. 7.5, демонстрирующий сайт продаж подержанных автомобилей, на котором двое, Алиса и Боб, пытаются одновременно купить одну и ту же машину. Покупка машины требует двух операций записи в базу данных: обновления списка на сайте с целью отразить в нем покупателя машины и отправки покупателю счета-фактуры. В случае, показанном на рис. 7.5, покупателем становится Боб (поскольку его обновление таблицы `listings` выполняется позже), а счет-фактура отправляется Алисе (поскольку она позднее обновляет таблицу `invoices`). Чтение зафиксированных данных предотвращает подобные казусы.

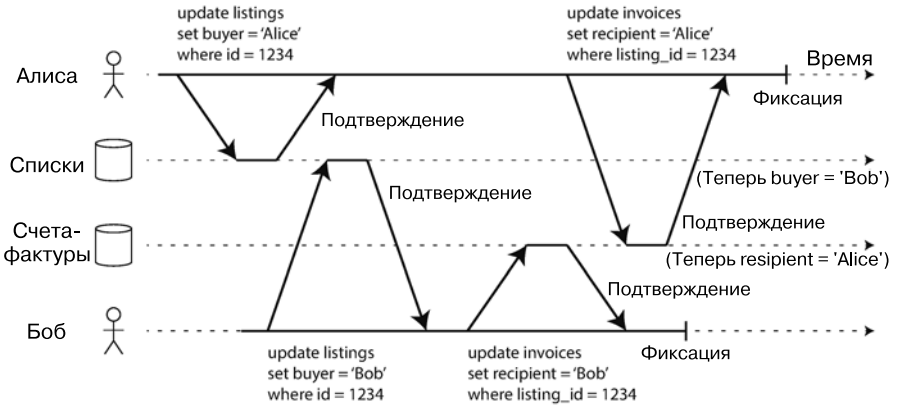


Рис. 7.5. В случае «грязных» операций записи конфликтующие операции записи различных транзакций могут оказаться перепутаны

- Однако чтение зафиксированных данных *не* предотвращает состояния гонки между двумя увеличениями счетчика на рис. 7.1. В этом случае вторая операция записи происходит после фиксации первой транзакции, так что не является «грязной». Но значения счетчиков все равно неправильные, правда, по другой причине — в подразделе «Предотвращение потери обновлений» текущего раздела мы обсудим, как обезопасить подобные приращения счетчиков.

Реализация чтения зафиксированных данных

Чтение зафиксированных данных — очень популярный уровень изоляции. Он используется по умолчанию в Oracle 11g, PostgreSQL, SQL Server 2012, MemSQL и многих других базах данных [8].

Чаще всего базы используют блокировки строк для предотвращения «грязных» операций записи: прежде чем модифицировать конкретный объект (строку или документ), транзакция должна сначала установить блокировку на этот объект. Данная блокировка должна удерживаться вплоть до фиксации или прерывания транзакции. Удерживать блокировку на конкретный объект может только одна транзакция одновременно, другой транзакции, желающей выполнить операцию записи в этот объект, придется дождаться фиксации или прерывания первой транзакции и лишь затем получить на него блокировку и продолжить свою работу. Подобные блокировки выполняются базами автоматически в режиме чтения зафиксированных данных (и на более сильных уровнях изоляции).

Как же предотвратить «грязные» операции чтения? Одна из возможностей: воспользоваться теми же блокировками и потребовать, чтобы желающие прочитать объект транзакции устанавливали блокировку, освобождая ее сразу же после чтения. Это гарантирует, что чтение не будет производиться при «грязном»,

незафиксированном значении объекта (поскольку в данное время блокировка удерживается транзакцией, выполняющей операцию записи).

Однако на практике требование блокировок чтения работает плохо — из-за случаев, когда множеству выполняющих только чтение транзакций приходится ждать завершения одной длительной транзакции записи. Это плохо сказывается на времени отклика выполняющих только чтение транзакций и удобстве эксплуатации: замедление в одной части приложения может привести к эффекту домино в совершенно других его частях вследствие ожидания блокировок.

Поэтому большинство БД предотвращают[1] «грязные» операции чтения с помощью подхода, показанного на рис. 7.4: база запоминает для каждого записываемого объекта как старое зафиксированное значение, так и новое, устанавливаемое транзакцией, удерживающей в данный момент блокировку записи. Во время выполнения транзакции всем другим транзакциям, читающим объект, просто возвращается старое значение. Только после фиксации нового значения транзакции начинают получать его при чтении.

Изоляция снимков состояния и воспроизводимое чтение

На первый взгляд уровня изоляции чтения зафиксированных данных вполне достаточно для транзакций: оно позволяет прерывать транзакции (что требуется для атомарности), предотвращает чтение промежуточных результатов транзакций и предотвращает смешивание конкурентных операций записи. Конечно, это полезные возможности и намного более сильные гарантии, чем у систем без транзакций.

Однако на этом уровне изоляции все еще существует множество возможных ошибок конкурентного доступа. Например, на рис. 7.6 показана одна из вероятных проблем при чтении зафиксированных данных.

Допустим, у Алисы есть в банке $1000 сбережений, разбитых между двумя счетами, на каждом из которых лежит по $500. Транзакция переводит $100 с одного принадлежащего Алисе счета на другой. Если ей не повезло и она посмотрит на перечень балансов своих счетов в момент обработки этой транзакции, то увидит баланс одного счета до поступления входящего платежа (когда баланс равен $500), а другой — после выполнения исходящего платежа (когда новый баланс равен $400). Алисе покажется при этом, что общая сумма на ее счетах равна $900 — $100 как будто провалились сквозь землю.

Подобная аномалия носит название *невоспроизводимого чтения* (nonrepeatable read) или *асимметрии чтения* (read skew): если Алиса прочитала бы баланс счета 1 опять в конце транзакции, то увидела бы значение ($600), отличное от прочитанного

[1] На момент написания книги единственные широко используемые БД, которые применяют блокировки для уровня чтения зафиксированных данных, — IBM DB2 и Microsoft SQL Server при включенной опции read_committed_snapshot=off [23, 26].

предыдущим запросом. Асимметрия считается допустимой при изоляции уровня чтения зафиксированных данных: видимые Алисе балансы счетов были, безусловно, зафиксированы на момент их чтения.

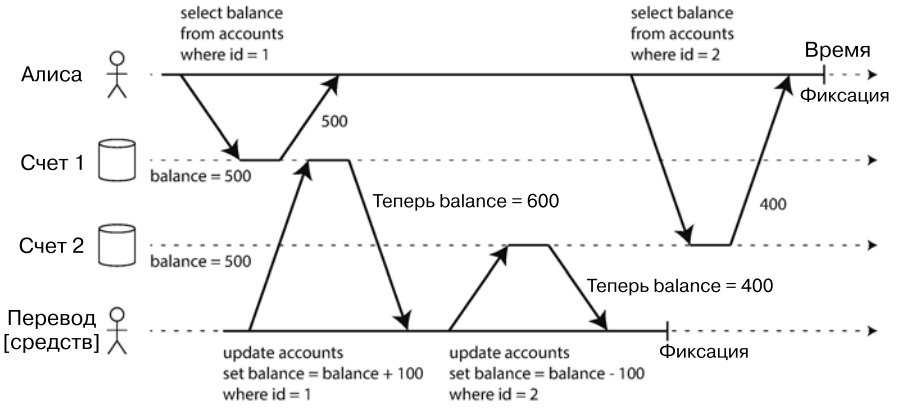


Рис. 7.6. Асимметрия чтения: Алиса видит базу данных в несогласованном состоянии



Термин «асимметрия» (skew), к сожалению, перегружен: мы ранее использовали его в смысле несбалансированной нагрузки с горячими точками (см. подраздел «Асимметричные нагрузки и разгрузка горячих точек» раздела 6.2), в то время как здесь он означает аномалию хронометража.

В случае Алисы это лишь временная проблема, поскольку она, скорее всего, увидит согласованные балансы счетов, перезагрузив сайт онлайн-банкинга всего через несколько секунд. Однако в некоторых ситуациях подобные временные несоответствия недопустимы.

- *Резервное копирование*. Резервная копия представляет собой копию всей базы данных, и ее создание на большой БД может занять несколько часов. Операции записи в базу продолжают выполняться во время создания резервной копии. Следовательно, может оказаться, что одни части копии содержат старые версии данных, а другие — новые. В случае восстановления БД из подобной резервной копии упомянутые расхождения (например, пропавшие деньги) станут из временных постоянными.
- *Аналитические запросы и проверки целостности*. Иногда приходится выполнять запросы, просматривающие значительные части базы данных. Подобные запросы — частое явление в аналитике (см. раздел 3.2). Они также могут быть частью периодической проверки целостности (мониторинга на предмет порчи данных). Если подобные запросы будут видеть разные части БД по состоянию на различные моменты времени, то их результаты будут совершенно бессмысленными.

Изоляция снимков состояния [28] — чаще всего используемое решение этой проблемы. Ее идея состоит в том, что каждая из транзакций читает данные из *согласованного снимка состояния* БД, то есть видит данные, которые были зафиксированы в базе на момент ее (транзакции) начала. Даже если данные затем были изменены другой транзакцией, каждая транзакция видит только старые данные, по состоянию на конкретный момент времени.

Изоляция снимков состояния — просто находка для долго выполняющихся запросов, которые только читают данные, например для создания резервных копий и аналитических запросов. Смысл запроса становится малопонятным, если данные, которыми он оперирует, меняются во время его выполнения. Он становится гораздо вразумительнее, когда транзакция работает на основе согласованного снимка состояния, «замороженного» в определенный момент времени.

Изоляция снимков состояния — часто используемая возможность, она поддерживается такими СУБД, как PostgreSQL, MySQL с подсистемой хранения InnoDB, Oracle, SQL Server и др. [23, 31, 32].

Реализация изоляции снимков состояния

Подобно уровню изоляции чтения зафиксированных данных, реализации изоляции снимков состояния обычно используют блокировки записи для предотвращения «грязных» операций записи (см. пункт «Реализация чтения зафиксированных данных» подраздела «Чтение зафиксированных данных» текущего раздела). Это значит, выполняющая операцию записи транзакция может блокировать выполнение другой транзакции, записывающей в тот же объект. Однако операции чтения не требуют никаких блокировок. С точки зрения производительности основной принцип изоляции снимков состояния звучит как *«чтение никогда не блокирует запись, а запись — чтение»*. Благодаря этому база данных способна выполнять длительные запросы на чтение, продолжая в то же время обработку операций записи, без какой-либо конкуренции блокировок между ними.

Базы данных применяют для реализации изоляции снимков состояния обобщение механизма, действие которого по части предотвращения «грязных» операций чтения мы наблюдали на рис. 7.4. БД должна хранить для этого несколько различных зафиксированных версий объекта, поскольку разным выполняемым транзакциям может понадобиться состояние базы на различные моменты времени. Вследствие хранения одновременно нескольких версий объектов этот метод получил название *многоверсионного управления конкурентным доступом* (multiversion concurrency control, MVCC).

Если базе необходима только изоляция уровня чтения зафиксированных данных, но не уровня изоляции снимков состояния, достаточно было бы хранить только две версии объекта: зафиксированную версию и перезаписанную, но еще не зафиксированную версию. Однако поддерживающие изоляцию снимков состояния

подсистемы хранения обычно используют MVCC и для изоляции уровня чтения зафиксированных данных. При этом обычно при чтении таких данных применяется отдельный снимок состояния для каждого запроса, а при изоляции снимков состояния — один и тот же снимок состояния для всей транзакции.

Рисунок 7.7 иллюстрирует реализацию изоляции снимков состояния на основе MVCC в СУБД PostgreSQL [31] (другие реализации аналогичны). В самом начале выполнения транзакция получает уникальный, монотонно возрастающий[1] идентификатор транзакции (`txid`). При каждой записи транзакцией записываемые в базу данные помечаются номером этой транзакции.

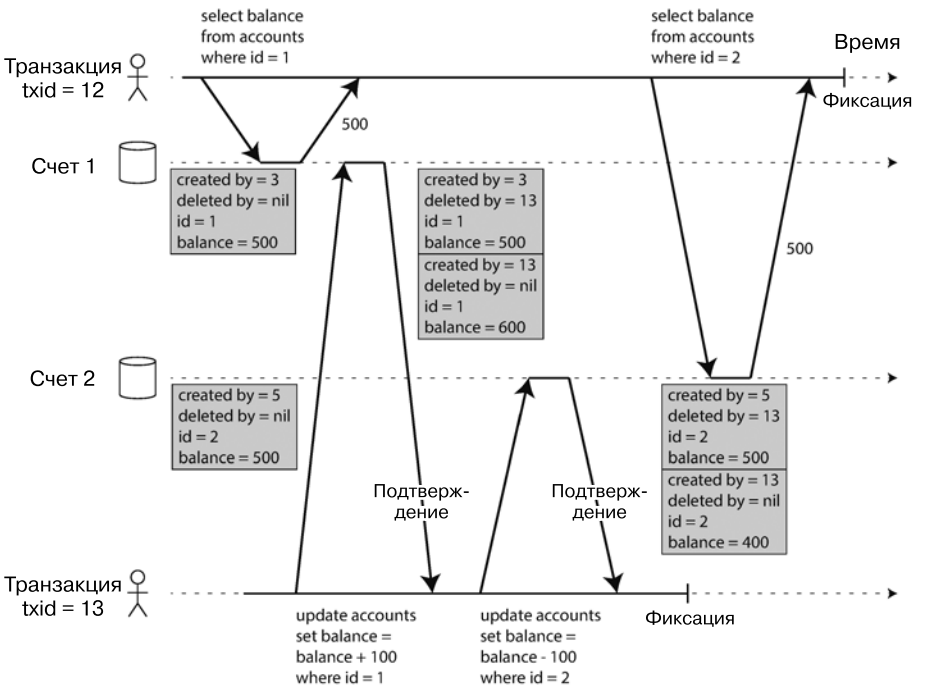


Рис. 7.7. Реализация изоляции снимков состояния с помощью многоверсионных объектов

В каждой строке таблицы есть поле `created_by`, содержащее идентификатор транзакции, вставившей эту строку в таблицу. Более того, в каждой строке таблицы есть поле `deleted_by`, изначально пустое. Если транзакция удаляет строку, то строка

[1] Точнее говоря, идентификаторы транзакций представляют собой 32-битные целые числа, и после примерно 4 миллиардов транзакций происходит их переполнение. Процесс vacuum PostgreSQL производит очистку, гарантируя, что это переполнение не повлияет на данные.

на самом деле не удаляется из базы данных, а помечается для удаления путем установки значения этого поля в соответствии с идентификатором запросившей удаление транзакции. В дальнейшем, когда уже никакая транзакция точно не обратится к удаленным данным, процесс сборки мусора БД удалит все помеченные для удаления строки и освободит занимаемое ими место.

Обновление превращается внутри базы данных в удаление и создание. Например, на рис. 7.7 транзакция 13 снимает $100 со счета 2, изменяя баланс с $500 на 400. Таблица `accounts` после этого содержит две строки для счета 2: строку с балансом $500, помеченную как удаленную транзакцией 13, и созданную транзакцией 13 строку с балансом $400.

Правила видимости для согласованных снимков состояния

Выполняя операции чтения из базы данных, транзакция использует идентификаторы транзакций, чтобы определить, какие объекты она может видеть, а какие — нет. Благодаря тщательно определяемым правилам видимости БД представляет приложению согласованный снимок состояния. Эти правила определяются следующим образом.

1. В начале каждой транзакции база данных создает список всех остальных выполняемых на текущий момент транзакций (но еще не зафиксированных или прерванных). Все выполненные этими транзакциями изменения игнорируются, даже если впоследствии они будут зафиксированы.
2. Все операции записи, выполненные прерванными транзакциями, игнорируются.
3. Все операции записи, выполненные транзакциями с более поздним идентификатором транзакции (то есть начавшиеся после запуска текущей транзакции), игнорируются независимо от того, были ли они зафиксированы.
4. Результаты всех остальных операций записи видны запросам приложения.

Эти правила применяются как при создании, так и при удалении объектов. На рис. 7.7 показано: когда транзакция 12 читает счет 2, видит баланс $500, поскольку удаление его производилось транзакцией 13 (а в соответствии с правилом 3 транзакция 12 не видит выполняемого транзакцией 13 удаления). И результат создания баланса $400 тоже ей не виден (в соответствии с тем же правилом).

Другими словами, объект является видимым, если одновременно справедливы следующие два условия:

- на момент начала выполнения читающей объект транзакции создавшая его транзакция уже зафиксирована;

- объект не помечен для удаления, а если и помечен, то запросившая удаление транзакция еще не была зафиксирована на момент начала выполнения читающей объект транзакции.

Длительные транзакции могут продолжать работать со снимком состояния и читать значения еще долгое время после того, как эти значения (с точки зрения других транзакций) были перезаписаны или удалены. База данных, создавая новую версию при каждом изменении значения вместо обновления значений, может обеспечить согласованные снимки состояния при совсем небольших дополнительных затратах.

Индексы и изоляция снимков состояния

Как могут работать индексы в многоверсионной базе данных? Один из вариантов: сделать так, чтобы индекс указывал на все версии объекта, и требовать выполнения запроса к индексу для фильтрации всех версий объекта, которые не должны быть видимы текущей транзакции. При удалении процессом сборки мусора старых версий объектов, более не видимых никаким транзакциям, соответствующие элементы индексов также удаляются.

На практике производительность управления конкурентным доступом с помощью многоверсионности определяют множество нюансов реализации. Например, в PostgreSQL имеются усовершенствования, позволяющие избежать обновлений индекса, если несколько версий одного объекта могут быть размещены на одной странице [31].

Другой подход используется в СУБД CouchDB, Datomic и LMDB. Хотя они тоже задействуют B-деревья (см. подраздел «B-деревья» раздела 3.1), но применяют схему *дописывания данных/копирования при записи* (append-only/copy-on-write), при которой происходит не перезапись страниц дерева при обновлении, а создание новой копии каждой из модифицированных страниц. Родительские страницы вплоть до корневой вершины дерева меняются так, чтобы указывать на новые версии их дочерних страниц. Страницы, на которые операция записи не повлияла, не копируются и остаются неизменяемыми [33–35].

При использовании B-деревьев, допускающих только добавление, каждая транзакция записи (или пакет транзакций) создает новую корневую вершину B-дерева, который является согласованным снимком состояния базы данных на момент его создания. Нет нужды фильтровать объекты по идентификаторам транзакций, поскольку последующие операции записи не модифицируют существующее B-дерево, а только создают новые корневые вершины. Однако такой подход требует фонового процесса для уплотнения данных и сбора мусора.

Воспроизводимое чтение и путаница с названиями

Изоляция снимков состояния — удобный уровень изоляции, особенно в случае транзакций только для чтения. Однако множество баз данных используют для него различные названия. В Oracle он называется уровнем *сериализации (*serializable)[1], а в PostgreSQL и MySQL — *воспроизводимым чтением* (repeatable read) [23].

Причина этой путаницы с названиями в том, что в стандарте SQL отсутствует понятие изоляции снимков состояния, поскольку он основан на определении уровней изоляции СУБД System R 1975 года [2], а данная изоляция тогда еще не была изобретена. Взамен он определяет воспроизводимое чтение, очень ее напоминающее. Базы данных PostgreSQL и MySQL называют свой уровень изоляции воспроизводимым чтением, поскольку он соответствует требованиям стандарта, что позволяет им заявлять о поддержке стандарта.

К сожалению, определения уровней изоляции из стандарта SQL небезупречны: они неоднозначны, неточны и не настолько независимы от реализации, как полагалось бы стандарту [28]. Хотя несколько баз данных и реализуют воспроизводимое чтение, фактически предоставляемые в них гарантии сильно различаются, несмотря на заявленную стандартизованность [23]. В научной литературе встречается формальное определение воспроизводимого чтения [29, 30], однако большинство реализаций ему не удовлетворяет. В довершение ко всему прочему, СУБД DB2 компании IBM называет воспроизводимым чтением сериализуемость [8].

В результате никто не может точно сказать, что же такое воспроизводимое чтение.

Предотвращение потери обновлений

Обсуждавшиеся до сих пор уровни чтения зафиксированных данных и изоляции снимков состояния в основном гарантировали, что транзакция только для чтения может встречаться в случае конкурентных операций записи. Вопрос двух транзакций, выполняющих одновременно операции записи, мы почти не затрагивали. Мы обсудили только «грязные» операции записи (см. пункт «Никаких “грязных” операций чтения» подраздела «Чтение зафиксированных данных» текущего раздела) — одну из возможных разновидностей конфликтов двойной записи (write-write conflict).

Существует несколько других интересных видов конфликтов, возникающих между транзакциями, конкурентно записывающими данные. Самый известный из них — проблема *потерянного обновления* (lost update), проиллюстрированная на рис. 7.1 примером конкурентного увеличения счетчика двумя пользователями.

---

[1] Или уровнем сериализуемых транзакций. — *Примеч. пер.*

Проблема потерянного обновления может возникать, когда приложение читает значение из базы данных, меняет его и записывает обратно измененное значение *(цикл чтения изменения записи)*. При конкурентном выполнении таких действий двумя транзакциями существует риск потери одного из изменений, поскольку вторая операция записи не учитывает предыдущего изменения (иногда говорят, что более поздняя операция записи *затирает* более раннее значение). Этот паттерн возникает в различных сценариях использования таких, как:

- увеличение счетчика или обновление баланса счета (требует чтения текущего значения, вычисления нового значения и записи обратно старого);
- выполнение локального изменения в составном значении, например, добавления элемента в список в JSON-документе (требует синтаксического разбора документа, выполнения изменения и записи обратно модифицированного документа);
- одновременное редактирование двумя пользователями страницы «Википедии», причем каждый из них сохраняет свои изменения путем отправки на сервер всего содержимого страницы, перезаписывая данные, содержащиеся в этот момент в базе.

Поскольку проблема встречается так часто, было разработано множество разнообразных ее решений.

Атомарные операции записи

Во многих базах данных есть возможность выполнять атомарные операций записи, что позволяет отказаться от циклов чтения изменения записи в коде приложения. Если необходимую логику можно выразить на языке этих операций, то они будут оптимальным решением. Например, следующая инструкция подходит для безопасного конкурентного выполнения в большинстве реляционных БД:

```
UPDATE counters SET value = value + 1 WHERE key = 'foo';
```

Аналогично документоориентированные базы данных, например MongoDB, предоставляют возможность использовать атомарные операции для локальных изменений частей JSON-документов, а СУБД Redis позволяет применять их для модификации таких структур данных, как очереди по приоритету. Далеко не все операции записи выражаемы на языке атомарных операций — в качестве примера приведу изменения страниц «Википедии», включающие произвольное редактирование текста[1], — но когда их вообще можно задействовать, атомарные операции обычно оказываются оптимальным решением.

---

[1] Редактирование текста можно, хотя и не без труда, выразить в виде потока атомарных изменений. Некоторые ссылки на литературу по этому вопросу вы можете найти во врезке «Автоматическое разрешение конфликтов» (см. пункт «Пользовательская логика разрешения конфликтов» подраздела «Обработка конфликтов записи» раздела 5.3).

Атомарные операции обычно реализуются путем эксклюзивной блокировки объекта при чтении, чтобы никакие другие транзакции не могли его прочитать до фиксации изменения. Этот метод иногда называют *чтением по установленному курсору* (cursor stability) [36, 37]. Другой вариант — просто выполнять все атомарные операции в отдельном потоке выполнения.

К сожалению, при использовании фреймворков объектно-реляционного отображения очень легко случайно написать код, выполняющий небезопасные циклы чтения-изменения-записи вместо применения предоставляемых базой данных атомарных операций [38]. Это не проблема, если вы понимаете, что делаете, но потенциальный источник ошибок, плохо обнаруживаемых при тестировании.

Явные блокировки

Другой способ предотвращения потери обновлений, если функциональности встроенных атомарных операций базы данных недостаточно, — явная блокировка приложением предназначенных для обновления объектов. После блокировки приложение может безопасно выполнить цикл чтения — изменения — записи, а другим транзакциям при попытке конкурентного чтения того же объекта придется ждать до завершения первого такого цикла.

Например, рассмотрим многопользовательскую игру, в которой несколько игроков могут одновременно двигать одну фигуру. В этом случае атомарных операций может оказаться недостаточно, поскольку приложение должно гарантировать соответствие ходов пользователя правилам игры, что требует логики, которая вряд ли реализуема в виде запроса базы данных. Вместо этого можно задействовать блокировки для предотвращения конкурентного перемещения одной фигуры двумя игроками, как показано в примере 7.1.

Пример 7.1. Явная блокировка строк для предотвращения потери обновлений

```
BEGIN TRANSACTION;

SELECT * FROM figures
  WHERE name = 'robot' AND game_id = 222
  FOR UPDATE; (1)

-- Проверяем допустимость хода, после чего обновляем
-- возвращенную предыдущим SELECT позицию фигуры.
UPDATE figures SET position = 'c4' WHERE id = 1234;

COMMIT;
```

(1) Предложение `FOR UPDATE` указывает базе данных блокировать все возвращенные запросом строки

Данный метод работает, но, чтобы сделать все правильно, нужно тщательно продумать логику приложения. Очень легко случайно где-нибудь в коде забыть добавить нужную блокировку, и это приведет к возникновению состояния гонки.

Автоматическое обнаружение потери обновлений

Атомарные операции и блокировки позволяют предотвратить потерю обновлений с помощью последовательного выполнения циклов чтения — изменения — записи. Но можно и разрешить им выполняться конкурентно, прерывая транзакцию в случае обнаружения потери обновления, с принудительным повтором цикла чтения — изменения — записи.

Преимуществом такого подхода является возможность эффективно выполнять эту проверку в связке с изоляцией снимков состояния. Конечно, на уровнях воспроизводимого чтения СУБД PostgreSQL, сериализуемости СУБД Oracle и изоляции снимков состояния СУБД SQL Server происходит автоматическое обнаружение потери обновлений и прерывание проблемных транзакций. Однако при воспроизводимом чтении в базах данных MySQL/InnoDB потерянные обновления не обнаруживаются [23]. Некоторые авторы [28, 30] доказывают, что база, претендующая на обеспечение изоляции снимков состояния, обязана предотвращать потерю обновлений, вследствие чего MySQL под это определение не подходит.

Обнаружение потерянных обновлений — замечательное свойство, поскольку приложению не требуется использовать какие-либо специальные возможности базы данных — можно забыть установить блокировку или задействовать атомарную операцию и вызвать тем самым ошибку, а обнаружение потерянных обновлений выполняется автоматически и, следовательно, меньше подвержено ошибкам.

Сравнение с обменом

В базах данных без транзакций иногда встречается операция «сравнение с обменом» (compare-and-set), упомянутая выше в пункте «Однообъектные операции записи» подраздела «Однообъектные и многообъектные операции» раздела 7.1. Цель этой операции — избежать потери обновлений благодаря тому, что обновления допускаются, только если значение не менялось с момента его прошлого чтения. При несовпадении текущего значения с прочитанным ранее обновление не выполняется и выполнение цикла чтения — изменения — записи необходимо повторить.

Например, для предотвращения конкурентного обновления страницы «Википедии» двумя пользователями можно попробовать следующее. Сделаем так, чтобы обновление происходило, только если содержимое страницы не менялось с момента начала ее редактирования пользователем:

```
-- В зависимости от реализации базы данных такой запрос
-- может быть безопасным или небезопасным
UPDATE wiki_pages SET content = 'new content'
  WHERE id = 1234 AND content = 'old content';
```

Если содержимое изменилось и более не совпадает с `'old content'`, то обновление не будет выполнено, так что необходимо проверить, сработало ли обновление, и повторить его при необходимости. Однако если БД позволяет предложению `WHERE` чтение из старого снимка состояния, то данный оператор не предотвратит потери

изменений, поскольку условие может оказаться истинным, несмотря на другую операцию записи, выполняемую конкурентно. Прежде чем использовать операцию сравнения с обменом вашей базы данных, проверьте, безопасна ли она.

Разрешение конфликтов и репликация

В реплицируемых базах данных (см. главу 5) предотвращение потери обновлений приобретает новый размах: поскольку у них есть копии в нескольких узлах и данные могут потенциально изменяться одновременно в различных узлах, необходимо предпринять дополнительные меры для предотвращения потери обновлений.

Блокировки и операции сравнения с обменом предполагают существование одной актуальной копии данных. Однако базы с репликацией без ведущего узла или с несколькими ведущими узлами допускают несколько одновременных операций записи и их асинхронную репликацию, так что гарантировать наличие одной актуальной копии данных невозможно. Следовательно, основанные на блокировках или сравнении с обменом методы в этом контексте неприменимы (мы вернемся к этому вопросу более обстоятельно в разделе 9.2).

Вместо этого, как обсуждалось в подразделе «Обнаружение конкурентных операций записи» раздела 5.4, в подобных реплицируемых базах данных конкурентным операциям записи часто разрешается создавать несколько конфликтующих версий значения (известных под названием *родственных значений*) и использовать код приложения или специализированных структур данных для разрешения и слияния этих версий постфактум.

Атомарные операции способны хорошо работать в контексте репликации, особенно если они коммутативны (то есть результат при их использовании в разном порядке в различных репликах остается тем же самым). Например, приращение счетчика или добавление элемента в множество — коммутативные операции. На этой идее основаны типы данных базы данных Riak 2.0, предотвращающие потери обновлений между репликами. В случае конкурентного обновления значения различными клиентами Riak автоматически сливает воедино эти обновления таким образом, что они не теряются [39].

С другой стороны, метод разрешения конфликтов *«выигрывает последний»* (last write wins, LWW) склонен к потере обновлений, как обсуждалось в пункте «“Выигрывает последний” (отбраковка конкурентных операций записи)» подраздела «Обнаружение конкурентных операций записи» раздела 5.4. К сожалению, LWW используется по умолчанию во многих реплицируемых базах данных.

Асимметрия записи и фантомы

В предыдущих подразделах мы столкнулись с *«грязными» операциями чтения* и *потерянными обновлениями* — двумя разновидностями состояний гонки, возникающими при попытке конкурентной записи в одни объекты различными транзакциями.

Чтобы избежать нарушения целостности данных, необходимо предотвращать подобные состояния гонки — либо автоматически, базой данных, либо вручную, с помощью таких мер безопасности, как блокировки или атомарные операции записи.

Однако на этом список возможных состояний гонки, возникающих при конкурентных операциях записи, отнюдь не исчерпывается. В данном подразделе мы рассмотрим некоторые более сложные примеры конфликтов.

Для начала представьте, что вы создаете приложение для врачей, предназначенное для организации смен дежурств в больнице. Больницы обычно стараются, чтобы в каждый момент времени было по возможности несколько дежурных врачей, но точно не меньше одного. Врачи могут отказываться от дежурств (например, если сами заболеют) при условии, что хотя бы один их коллега остается на дежурстве [40, 41].

Теперь представьте, что Алиса и Боб — два дежурных доктора на конкретной смене. Оба плохо себя чувствуют, и оба решили попросить отгул. К сожалению, они нажали на кнопку запроса отгула примерно в одно время. Происходящее после этого показано на рис. 7.8.

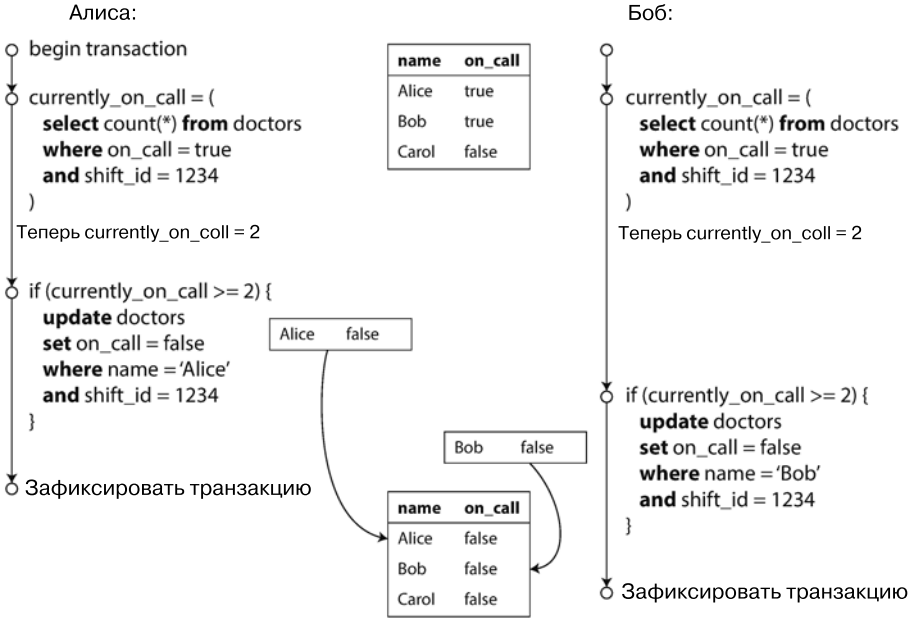


Рис. 7.8. Пример асимметрии записи, вызванной ошибкой в коде приложения

В каждой из транзакций приложение сначала проверяет наличие в данный момент двух или более дежурных врачей. При положительном результате оно считает, что можно безопасно дать одному из врачей отгул. Поскольку база данных использует изоляцию снимков состояния, обе проверки возвращают 2 и выполнение обеих

транзакций продолжается. Алиса успешно обновляет свой график и получает отгул. И Боб делает то же. Происходит фиксация обеих транзакций, после чего оказывается, что ни одного дежурного врача нет. Требование наличия хотя бы одного дежурного врача оказалось нарушено.

Характеристики асимметрии записи

Такая аномалия носит название *асимметрии записи* (write skew) [28]. Это не «грязная» операция записи и не потеря обновления, поскольку две наши транзакции обновляют два различных объекта (графики дежурств Алисы и Боба соответственно). Наличие конфликта тут менее заметно, но это, безусловно, состояние гонки: если две транзакции выполнялись бы одна за другой, то второй врач не получил бы отгула. Аномальное поведение стало возможно только потому, что транзакции выполнялись конкурентно.

Асимметрию записи можно рассматривать как обобщение проблемы потери обновлений. Эта асимметрия может происходить при чтении двумя транзакциями одних и тех же объектов с последующим обновлением некоторых из них (различные транзакции могут обновлять разные объекты). В частном случае, когда различные транзакции обновляют один объект, происходит «грязная» операция записи или потеря обновления (в зависимости от хронометража).

Мы видели выше, что существует множество различных способов предотвращения потери обновлений. В случае асимметрии записи возможностей оказывается меньше.

- Атомарные однообъектные операции не помогут, поскольку в транзакции участвует несколько объектов.
- К сожалению, не поможет и автоматическое обнаружение потерянных обновлений, встречающееся в некоторых реализациях изоляции снимков состояния: асимметрия записи автоматически не обнаруживается ни при воспроизводимом чтении в PostgreSQL или MySQL/InnoDB, ни при сериализуемых транзакциях Oracle, ни на уровне изоляции снимков состояния SQL Server [23]. Автоматическое предотвращение асимметрии записи требует настоящей сериализуемости (см. раздел 7.3).
- Некоторые базы данных позволяют создавать ограничения целостности, за соблюдением которых затем следит сама база (например, ограничения уникальности, ограничения внешних ключей или ограничения, накладываемые на конкретное значение). Однако для спецификации наличия хотя бы одного дежурного врача понадобилось бы ограничение, налагаемое на несколько объектов. В большинстве баз нет встроенной поддержки подобных ограничений, но их можно реализовать с помощью триггеров или материализованных представлений, в зависимости от базы данных [42].
- Если использовать уровень сериализуемости невозможно, то вторым лучшим решением будет, вероятно, явная блокировка строк, необходимых для

транзакции. В примере с врачами можно написать примерно следующие запросы:

```
BEGIN TRANSACTION;

SELECT * FROM doctors
  WHERE on_call = true
  AND shift_id = 1234 FOR UPDATE; (1)

UPDATE doctors
  SET on_call = false
  WHERE name = 'Alice'
  AND shift_id = 1234;

COMMIT;
```

(1) Предложение `FOR UPDATE`, как и выше, указывает базе установить блокировку на все возвращенные данным запросом строки.

Дополнительные примеры асимметрии записи

Асимметрия записи может показаться на первый взгляд запутанным вопросом, но немного разобравшись, вы научитесь замечать ситуации, чреватые ее возникновением. Вот несколько примеров.

- *Система бронирования конференц-залов.* Допустим, вам нужно обеспечить невозможность двух операций бронирования одного конференц-зала на одно время [43]. В этом случае при запросе бронирования сначала необходимо проверить наличие конфликтующих операций бронирования (то есть бронирования того же зала на пересекающийся промежуток времени). Если таковых не нашлось, то можно создавать свое бронирование (пример 7.2)[1].

Пример 7.2. Система бронирования конференц-залов пытается избежать двойных бронирований (небезопасно при уровне изоляции снимков состояния)

```
BEGIN TRANSACTION;

-- Проверяем наличие существующих операций бронирования на промежуток времени,
-- пересекающийся с промежутком времени 12:00–13:00
SELECT COUNT(*) FROM bookings
  WHERE room_id = 123 AND
    end_time > '2015-01-01 12:00' AND start_time < '2015-01-01 13:00';

-- Если предыдущий запрос вернул 0:
INSERT INTO bookings
  (room_id, start_time, end_time, user_id)
  VALUES (123, '2015-01-01 12:00', '2015-01-01 13:00', 666);

COMMIT;
```

[1] В базе данных PostgreSQL это можно реализовать более изящно, воспользовавшись диапазонными типами данных, но в других БД они поддерживаются редко.

К сожалению, изоляция снимков состояния не предотвращает конкурентные вставки конфликтующих операций бронирования другими пользователями. Чтобы гарантировать отсутствие конфликтов расписания, нам опять понадобится изоляция уровня сериализуемости.

- *Многопользовательские игры*. В примере 7.1 мы задействовали блокировку для предотвращения потери обновлений (то есть гарантировали, что два игрока не смогут передвинуть одну фигуру одновременно). Однако блокировка не препятствует игрокам перемещать две различные фигуры на одну позицию на доске или совершать другой нарушающий правила игры ход. В зависимости от вида воплощаемого правила вы можете применять ограничения уникальности, в противном случае есть вероятность столкнуться с асимметрией записи.
- *Заявка на имя пользователя*. На сайте, где у каждого пользователя должно быть уникальное имя, два пользователя могут попытаться одновременно создать учетные записи с одинаковым именем. Можно применить транзакцию для проверки, занято ли данное имя, и в случае отрицательного ответа создавать соответствующую учетную запись. Однако, как и в предыдущем примере, на уровне изоляции снимков состояния это небезопасно. К счастью, есть простое решение проблемы: ограничение уникальности целостности (вторая транзакция, которая попытается зарегистрировать данное имя пользователя, будет прервана вследствие нарушения этого ограничения).
- *Предотвращение двойных списаний средств*. Сервис, позволяющий пользователям тратить деньги или баллы, должен проверять, чтобы пользователь не потратил больше, чем у него есть. Реализовать это можно с помощью вставки в счет пользователя предполагаемой статьи расходов, суммирования стоимостей всех статей расходов на счету и проверки неотрицательности суммы [44]. При асимметрии записи может произойти конкурентная вставка двух статей расходов, которые вместе приведут к перерасходу средств, однако ни одна из транзакций не увидит другую.

Асимметрия записи вследствие фантомов

Все вышеприведенные примеры следуют одной схеме.

1. Запрос `SELECT` проверяет некое требование путем поиска удовлетворяющих определенному условию строк (например, сейчас дежурят хотя бы два врача, на конкретное время нет уже существующего бронирования, в данной позиции на доске нет другой фигуры, имя пользователя не занято, на счету еще есть деньги).
2. В зависимости от результата первого запроса код приложения решает, что делать дальше (продолжать выполнение операции или известить пользователя об ошибке и прервать выполнение).
3. Если приложение решает продолжить выполнение, то производит операцию записи (`INSERT`, `UPDATE` или `DELETE`) в базу данных и фиксирует транзакцию.

Результат данной операции записи изменяет входные условия принимаемого на шаге 2 решения. Другими словами, если бы пришлось повторить запрос SELECT из шага 1 после фиксации операции записи, то результат мог бы оказаться другим, поскольку эта операция изменила множество строк, удовлетворяющих условию поиска (на дежурстве теперь на одного врача меньше, конференц-зал забронирован на указанное время, позиция на доске уже занята передвинутой фигурой, имя пользователя занято, на счету стало меньше денег).

Эти шаги могут выполняться и в другом порядке. Например, вы могли сначала выполнить операцию записи, затем запрос SELECT и, наконец, решить, прерывать транзакцию или фиксировать, в зависимости от результата запроса.

В примере с дежурными врачами измененная на шаге 3 строка была одной из строк, возвращенных на шаге 1, так что можно было обезопасить транзакцию и избежать асимметрии записи, блокируя строки на шаге 1 (SELECT FOR UPDATE). Однако остальные четыре примера отличаются от него: они выполняют проверку на *отсутствие* удовлетворяющих определенному условию поиска строк, а операция записи *вставляет* соответствующую этому же условию строку. Если запрос на шаге 1 не вернет ничего, то SELECT FOR UPDATE не на что будет устанавливать блокировки.

Такой эффект, при котором операция записи в одной транзакции меняет результат запроса на поиск в другой, называется *фантомом* (phantom) [3]. Изоляция снимков состояния предотвращает возникновение фантомов в запросах только для чтения. Но в транзакциях, выполняющих чтение и запись данных, в таких как в обсуждавшихся выше примерах, фантомы могут приводить к особенно запутанным случаям асимметрии записи.

Материализация конфликтов

Раз уж проблема с фантомами заключается в отсутствии объекта, на который можно было бы установить блокировку, вероятно, имеет смысл искусственно создать объект для блокировки в базе данных?

Например, в случае с бронированием конференц-залов можно было бы создать таблицу отрезков времени и залов. Каждая строка в ней соответствовала бы конкретному залу на определенный период времени (допустим, 15 минут). Можно было бы создать строки для всех вероятных сочетаний залов и отрезков времени заранее, например, на следующие шесть месяцев.

Тогда транзакция перед созданием бронирования могла бы блокировать (SELECT FOR UPDATE) строки в этой таблице, соответствующие нужным залу и отрезку времени. После установки блокировок можно проверить пересечения операций бронирования и вставить новую бронь, как и раньше. Обратите внимание: сама дополнительная таблица не служит для хранения информации о бронировании — это просто набор блокировок, используемый для предотвращения конкурентного изменения бронирования одного и того же зала на одно время.

Такой подход с превращением фантома в конфликт при блокировке на конкретном множестве существующих в БД строк носит название *материализации конфликтов* (materializing conflicts) [11]. К сожалению, материализация конфликтов непроста и подвержена ошибкам, а перетекание механизма управления конкурентным доступом в модель данных приложения выглядит не очень красиво. Поэтому материализацию конфликтов следует рассматривать как последнее средство, если нет никаких альтернатив. В большинстве случаев предпочтительнее использовать изоляцию уровня сериализуемости.

7.3. Сериализуемость

В этой главе мы рассмотрели несколько примеров транзакций, подверженных проблемам состояния гонки. Некоторые состояния можно предотвратить с помощью уровней чтения зафиксированных данных и изоляции снимков состояния, но не все. Мы видели несколько довольно запутанных примеров с асимметрией записи и фантомами. Ситуация, прямо скажем, печальная:

- уровни изоляции реализованы в разных базах данных по-разному, и разобраться в них непросто (например, смысл термина «воспроизводимое чтение» сильно варьируется);
- глядя на код приложения — особенно большого, в котором трудно охватить взглядом все конкурентные действия, — непросто сказать, безопасно ли его выполнять на данном уровне изоляции;
- не существует никаких удобных инструментов для обнаружения состояний гонки. В принципе, может оказаться полезен статический анализ [26], но эти научные методы пока еще неприменимы на практике. Тестирование едва ли позволит обнаружить проблемы конкурентного доступа, поскольку они обычно недетерминированы и возникают только в случае неудачного хронометража.

Эта проблема не нова — она известна с конца 1970-х годов, с момента появления слабых уровней изоляции [2]. И всегда исследователи давали один и тот же простой совет: используйте *сериализуемость*!

Сериализуемость (serializability) обычно считается самым сильным уровнем изоляции. Она гарантирует, что даже при конкурентном выполнении транзакций результат останется таким же, как и в случае их *последовательного* (по одной за раз) выполнения, без всякой конкурентности. Следовательно, база данных гарантирует, что правильно выполняющиеся последовательно транзакции будут столь же правильно выполняться конкурентно. Другими словами, база предотвращает *все* возможные состояния гонки.

Но если сериализуемость настолько лучше более слабых уровней изоляции, почему бы не использовать ее повсеместно? Для ответа на данный вопрос рассмотрим вероятные реализации сериализуемости и их производительность. Большинство

современных БД, обеспечивающих сериализуемость, применяют один из трех методов, которые мы рассмотрим в оставшейся части этой главы:

- действительно последовательное выполнение транзакций (см. подраздел «По-настоящему последовательное выполнение» текущего раздела);
- двухфазную блокировку (см. подраздел «Двухфазная блокировка (2PL)» данного раздела), на протяжении нескольких десятилетий бывшую единственным заслуживающим внимания вариантом;
- методы оптимистического управления конкурентным доступом, например сериализуемую изоляцию снимков состояния (см. подраздел «Сериализуемая изоляция снимков состояния (SSI)» текущего раздела).

Пока что мы рассмотрим эти методы, главным образом, применительно к одноузловым базам данных. В главе 9 изучим возможность их обобщения на многоузловые транзакции в распределенных системах.

Последовательное выполнение

Простейший способ избежать проблем с конкурентным доступом — вообще отказаться от него: выполнять одновременно только одну транзакцию, последовательно, в одном потоке выполнения. Тем самым мы полностью обходим задачу обнаружения и предотвращения конфликтов между транзакциями: подобная изоляция транзакций по определению является сериализуемой.

Хотя эта идея довольно очевидна, создатели баз данных только относительно недавно, в 2007 году, решили, что однопоточный цикл выполнения транзакций оправдывает себя [45]. Если целых 30 лет многопоточный конкурентный доступ считался необходимым для достижения хорошей производительности, то как случилось, что однопоточное выполнение стало считаться допустимым?

Этот пересмотр концепции был вызван двумя факторами.

- RAM стала достаточно дешевой для того, чтобы во многих сценариях использования можно было хранить в оперативной памяти весь активный набор данных (см. пункт «Храним все в памяти» подраздела «Другие индексные структуры» раздела 3.1). Транзакции выполняются гораздо быстрее, если все необходимые данные находятся в оперативной памяти и не нужно ждать их загрузки с жесткого диска.
- Создатели баз данных осознали, что OLTP-транзакции обычно короткие и выполняют лишь небольшое количество операций записи и чтения (см. раздел 3.2). Напротив, длительные аналитические запросы обычно только читают информацию, поэтому их можно выполнять на согласованном снимке состояния (на уровне изоляции снимков состояния), вне цикла последовательного выполнения.

Подход с последовательным выполнением транзакций реализован в таких СУБД, как VoltDB/H-Store, Redis и Datomic [46–48]. Нацеленные на однопоточное выполнение системы иногда работают быстрее поддерживающих конкурентность, поскольку не требуют затрат на блокировки. Однако их пропускная способность ограничивается пропускной способностью одного ядра CPU. Чтобы выжать из этого одного потока выполнения как можно больше, необходимо упорядочивать транзакции не так, как обычно.

Инкапсуляция транзакций в хранимых процедурах

В самом начале эры баз данных предполагалось, что транзакция БД должна охватывать весь поток действий пользователя. Например, бронирование авиабилета — многоэтапный процесс (поиск маршрутов, цен и свободных мест; выбор маршрута; бронирование мест на каждом из участков маршрута; ввод данных пассажира; оплата). Создатели баз предполагали: было бы удобно сделать весь этот процесс одной транзакцией, чтобы фиксировать ее атомарно в случае надобности.

К сожалению, люди не так быстро реагируют и принимают решения. База, транзакции которой ожидают ввода данных пользователем, должна поддерживать потенциально огромное количество конкурентных транзакций, основная часть из которых простаивает. Большинство БД не способно реализовать это эффективно, так что почти все OLTP-приложения стараются сократить длительность транзакций за счет отказа от интерактивного ожидания ответа пользователя внутри транзакции. Применительно к веб-технологиям это означает, что транзакции фиксируются в том же HTTP-запросе, который их открыл, — транзакции не охватывают несколько запросов сразу. Каждый новый HTTP-запрос открывает новую транзакцию.

Несмотря на исключение людей из этого критического маршрута, транзакции продолжали выполняться в интерактивном режиме «клиент/сервер», по одному оператору за раз.

Приложение выполняет запрос, читает результат, возможно, выполняет еще один запрос в зависимости от результата первого и т. д. Запросы и их результаты путешествуют между кодом приложения (выполняемым на одной машине) и сервером базы данных (на другой).

При таком интерактивном режиме выполнения транзакций огромное количество времени тратится на обмен сетевыми сообщениями между приложением и базой данных. Пропускная способность в случае отключения конкурентного доступа в БД и обработки одной транзакции за раз оказалась бы ужасной, поскольку база тратила бы большую часть времени на ожидание генерации приложением следующего запроса для текущей транзакции. Чтобы достичь приемлемой производительности

в подобных базах данных, необходимо обрабатывать несколько транзакций одновременно.

Поэтому в системах с однопоточным последовательным выполнением транзакций интерактивные многооператорные транзакции запрещены. Вместо них приложение заранее отправляет весь код транзакции в базу данных в виде *хранимой процедуры* (stored procedure). Различие между этими подходами показано на рис. 7.9. Хранимая процедура выполняется очень быстро при условии, что все необходимые транзакции данные находятся в оперативной памяти, так как ей не приходится ожидать завершения сетевых или дисковых операций ввода/вывода.

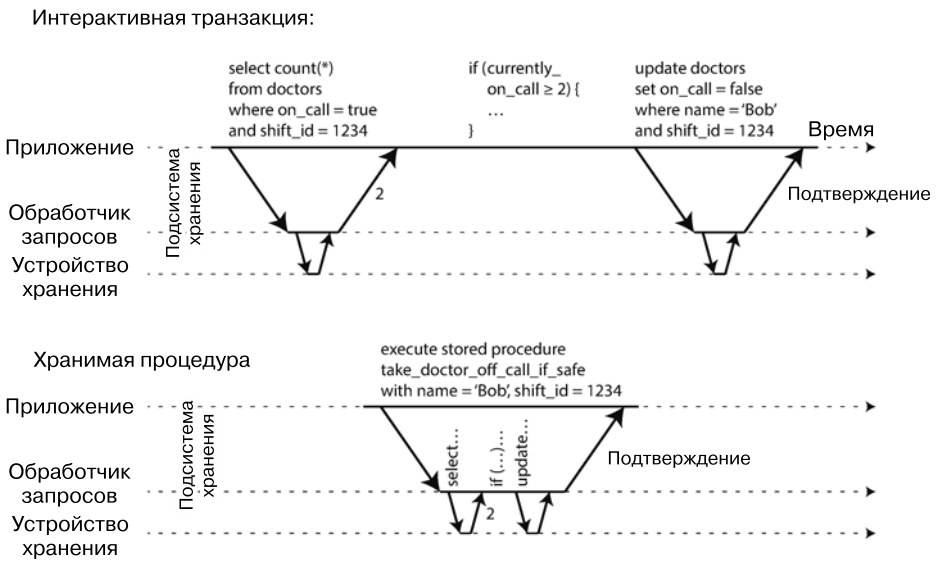


Рис. 7.9. Различие между интерактивной транзакцией и хранимой процедурой (на примере транзакции с рис. 7.8)

Достоинства и недостатки хранимых процедур

Хранимые процедуры уже существовали в реляционных базах данных довольно долго и были частью стандарта языка SQL (SQL/PSM) с 1999 года. Однако по разным причинам они завоевали не самую лучшую репутацию.

- У всех производителей СУБД свои языки написания хранимых процедур (PL/SQL у Oracle, T-SQL у SQL Server, PL/pgSQL у PostgreSQL и т. д.). Эти языки не успевают за новшествами неспециализированных языков программирования и выглядят довольно уродливо и «замшело» в нынешних реалиях. Кроме того, им недостает экосистемы библиотек, имеющейся у большинства языков программирования.

- Выполняемый в базе данных код труднее структурировать: по сравнению с сервером приложений его сложнее отлаживать, осуществлять контроль версий и развертывание и интегрировать с системами сбора метрик для мониторинга.
- Базы данных часто более чувствительны к производительности, чем серверы приложений, поскольку один экземпляр базы часто используется несколькими серверами приложений. Неудачно написанная хранимая процедура (например, применяется много памяти или процессорного времени) в БД может доставить гораздо больше неприятностей, чем столь же неудачно написанный код в сервере приложений.

Однако с этими проблемами можно справиться. Современные реализации хранимых процедур используют не PL/SQL, а существующие универсальные языки программирования: VoltDB задействует Java или Groovy, Datomic — Java или Clojure, а Redis — Lua.

Благодаря хранимым процедурам и применению данных в оперативной памяти становится возможным выполнение всех транзакций в одном потоке. В этом случае достигается вполне приличная производительность за счет того, что нет необходимости ждать завершения операций ввода/вывода и отсутствуют затраты на механизмы управления конкурентным доступом.

VoltDB также использует хранимые процедуры для репликации: вместо дублирования операций записи транзакций из одного узла в другой она выполняет одни и те же хранимые процедуры во всех репликах. Следовательно, для VoltDB необходимо, чтобы хранимые процедуры были *детерминированными* (возвращали одни и те же результаты при выполнении в различных узлах). Транзакция, использующая текущие дату и время, должна делать это через специальные детерминистичные API.

Секционирование

Последовательное выполнение всех транзакций значительно упрощает контроль версий, но ограничивает пропускную способность базы данных по транзакциям скоростью отдельного ядра процессора конкретной машины. Транзакции только для чтения могут выполняться где угодно, с помощью изоляции снимков состояния, но для приложений с высокой нагрузкой по записи однопотоковая обработка транзакций может оказаться узким местом.

Для масштабирования на несколько ядер CPU и несколько узлов можно секционировать данные (см. главу 6) — VoltDB это позволяет. Если разбить набор данных так, чтобы каждой транзакции необходимо было читать и записывать данные только в пределах отдельной секции, то появляется возможность отдельного потока обработки транзакций для каждой секции, независимо от других. В этом

случае можно поставить каждому из ядер CPU в соответствие свою секцию, что обеспечит масштабирование пропускной способности линейно по отношению к числу ядер CPU [47].

Однако база данных должна согласовать выполнение любой обращающейся к нескольким секциям транзакции по всем затрагиваемым ею секциям. Хранимые процедуры необходимо выполнять во всех секциях строго синхронно, чтобы обеспечить сериализуемость в масштабе всей системы.

В силу дополнительных затрат на согласование многосекционные транзакции работают намного медленнее, чем транзакции, имеющие дело только с одной секцией. VoltDB заявляет о пропускной способности примерно 1000 записей в секунду в случае транзакций, имеющих дело с несколькими секциями. Это на порядки меньше пропускной способности односекционных транзакций, и увеличить данный показатель с помощью добавления дополнительных машин не получится [49].

Можно ли сделать транзакцию односекционной — очень зависит от структуры используемых приложением данных. Простые данные типа «ключ — значение» часто очень легко секционировать, но данные с вторичными индексами обычно требуют серьезного межсекционного согласования (см. раздел 6.3).

Краткое резюме последовательного выполнения

Последовательное выполнение транзакций стало одним из перспективных способов достижения изоляции снимков состояния, с некоторыми ограничениями.

- Все транзакции должны быть маленькими и быстрыми, поскольку одна-единственная медленная транзакция может застопорить всю работу.
- Набор данных должен помещаться в памяти целиком. Потенциально можно переместить на диск редко используемые данные, но если понадобится обратиться к ним в однопоточной транзакции, система способна очень сильно замедлиться[1].
- Пропускная способность по записи должна быть достаточно низкой для обработки на одном ядре CPU, иначе придется секционировать транзакции и отказаться от согласования многосекционных транзакций.
- Многосекционные транзакции возможны, но их сфера применения жестко ограничена.

---

[1] Если транзакции понадобились данные, находящиеся не в оперативной памяти, то лучшим решением будет прервать ее, асинхронно считать данные в оперативную память, параллельно с обработкой других транзакций, и перезапустить исходную транзакцию после загрузки данных. Такой подход называется антикэшированием (anti-caching), как уже упоминалось в пункте «Храним все в памяти» подраздела «Другие индексные структуры» раздела 3.1.

Двухфазная блокировка (2PL)

В течение почти 30 лет в базах данных широко использовался только один алгоритм сериализуемости: *двухфазная блокировка* (two-phase locking, 2PL)[1].



2PL не 2PC

Обратите внимание, что хотя название двухфазной блокировки (2PL) очень схоже с названием двухфазной фиксации транзакций (2PC), это две совершенно разные вещи. Мы обсудим 2PC в главе 9.

Ранее мы уже видели, что блокировки часто используются для предотвращения «грязных» операций записи (см. пункт «Никаких “грязных” операций записи» подраздела «Чтение зафиксированных данных» раздела 7.2): при конкурентной записи одного объекта двумя транзакциями блокировка гарантирует, что второй из записывающих транзакций придется подождать завершения первой (ее прерывания или фиксации).

Двухфазная блокировка напоминает обычную, но существенно усиливает требования. Допускается конкурентное чтение одного объекта несколькими транзакциями при условии, что никто его не записывает. Но для выполнения операции записи этого объекта (изменения или удаления) требуется монопольный доступ.

- Если транзакция A читает объект, а транзакция B хочет выполнить запись этого объекта, то B должна дождаться фиксации или прерывания транзакции A, прежде чем продолжить работу (это гарантирует, что B внезапно не поменяет данный объект «за спиной» транзакции A).
- Если транзакция A записывает объект, а транзакция B хочет его прочитать, то должна дождаться фиксации или прерывания транзакции A, прежде чем продолжить работу (чтение старой версии объекта, аналогично рис. 7.1, недопустимо при 2PL).

При двухфазной блокировке записывающие транзакции блокируют не просто другие записывающие транзакции, но и читающие и наоборот. Правило изоляции снимков состояния — *читающие транзакции никогда не блокируют записывающие, а записывающие никогда не блокируют читающие* (см. пункт «Реализация изоляции снимков состояния» подраздела «Изоляция снимков состояния и воспроизводимое чтение» раздела 7.2) — демонстрирует ключевое отличие между изоляцией снимков состояния и двухфазной блокировкой. С другой стороны, двухфазная блокировка, обеспечивая сериализуемость, защищает от всех обсуждавшихся выше состояний гонки, включая потери обновлений и асимметрию записи.

[1] Иногда называемая сильной строгой двухфазной блокировкой (strong strict two-phase locking, SS2PL), чтобы отличать ее от других вариантов 2PL.

Реализация двухфазной блокировки

2PL используется на уровне сериализуемых транзакций СУБД MySQL (InnoDB) и SQL Server и на уровне воспроизводимого чтения СУБД DB2 [23, 36].

Чтобы реализовать блокировку читающих и записывающих транзакций, на каждом объекте в базе данных имеется блокировка, которая может находиться или в *разделяемом* (shared mode), или в *монопольном режиме* (exclusive mode). Эти блокировки применяются следующим образом.

- Перед чтением объекта транзакция должна сначала установить блокировку в разделяемом режиме. Допускается удержание блокировки в разделяемом режиме несколькими транзакциями одновременно, но если какая-либо еще транзакция уже установила монопольную блокировку на объект, то этой транзакции следует подождать снятия монопольной блокировки.
- Перед записью объекта транзакция сначала должна установить блокировку в монопольном режиме. Другим транзакциям не разрешается удерживать блокировку одновременно с ней (ни в разделяемом, ни в монопольном режиме), так что если на объект уже установлена блокировка, то ей придется подождать.
- При чтении объекта с последующей записью транзакция может повысить уровень блокировки с разделяемой до монопольной. Такое повышение уровня блокировки выполняется аналогично непосредственной установке монопольной блокировки.
- После установки блокировки транзакция должна продолжать удерживать ее вплоть до завершения (фиксации или прерывания). Отсюда и название «двухфазная»: первая фаза (во время выполнения транзакции) представляет собой получение блокировок, а вторая (в конце выполнения транзакции) — их освобождение.

Из-за столь большого количества блокировок часто встречается ситуация, когда транзакция A ждет снятия блокировки транзакции B и наоборот. Такая ситуация называется *взаимной блокировкой* (deadlock). База данных автоматически обнаруживает взаимные блокировки между транзакциями и прерывает одну из них, чтобы остальные могли продолжить работу. Приложению приходится повторно выполнять прерванную транзакцию.

Производительность двухфазной блокировки

Главный недостаток двухфазной блокировки и причина, по которой ее не использовали повсеместно с 1970-х годов, — ее низкая производительность: пропускная способность по транзакциям и время отклика запросов значительно хуже при двухфазной блокировке, чем при слабой изоляции.

Отчасти это происходит из-за затрат на установку и освобождение всех блокировок, но в большей степени — вследствие снижения степени конкурентности.

По определению, если две конкурентные транзакции пытаются произвести какие-либо действия, способные потенциально привести к состоянию гонки, одной из них придется ожидать завершения второй.

Традиционные реляционные БД не ограничивают длительность выполнения транзакций, поскольку они созданы в расчете на интерактивные приложения, ожидающие ввода данных пользователем. Следовательно, одна транзакция может ожидать другую неограниченное время. Даже если постараться максимально сократить длительность всех своих транзакций, когда нескольким транзакциям необходимо обратиться к одному объекту, может сформироваться очередь, поскольку одной транзакции, прежде чем выполнить какие-либо действия, придется ждать завершения нескольких других.

Поэтому время ожидания в использующих 2PL базах данных способно меняться в достаточно широких пределах, и на верхних процентилях они могут работать очень медленно (см. подраздел «Описание производительности» раздела 1.3) в случае конкуренции за ресурсы. Достаточно одной медленно работающей транзакции или одной транзакции, обращающейся к большому объему данных и устанавливающей множество блокировок, чтобы вся остальная система замедлилась вплоть до полной остановки. Подобная нестабильность нежелательна в случаях, когда требуется надежная работа.

Хотя взаимные блокировки могут происходить и на уровне изоляции чтения зафиксированных данных с использованием блокировок, значительно чаще они встречаются при сериализуемом уровне изоляции 2PL (в зависимости от паттернов доступа приложения). Такая ситуация способна повлечь дополнительные проблемы с производительностью: если транзакция прерывается из-за взаимной блокировки и выполняется повторно, то должна выполнить все действия заново. При часто происходящих взаимных блокировках это означает значительные ненужные затраты.

Предикатные блокировки

В предыдущем описании блокировок мы обошли стороной важный, но малозаметный нюанс. В пункте «Асимметрия записи вследствие фантомов» подраздела «Асимметрия записи и фантомы» раздела 7.2 мы обсуждали проблему *фантомов* — изменение одной транзакцией результатов запроса на поиск другой транзакции. База данных, воплощающая сериализуемость, обязана предотвращать возникновение фантомов.

В примере с бронированием конференц-залов это бы означало, что если одна транзакция выполнила поиск существующих операций бронирования зала на определенный промежуток времени (см. пример 7.2), то другой транзакции не разрешается конкурентно вставлять или обновлять другую операцию бронирования

того же зала на то же время (можно конкурентно вставлять бронирование на другие залы или на тот же зал на другое время).

Как все это реализовать? По сути, нам нужна *предикатная блокировка* (predicate lock) [3]. Она аналогична описанным ранее разделяемым/монопольным блокировкам, но относится не к конкретному объекту (например, одной строке в таблице), а ко всем объектам, удовлетворяющим какому-то условию отбора:

```
SELECT * FROM bookings
  WHERE room_id = 123 AND
    end_time   > '2018-01-01 12:00' AND
    start_time < '2018-01-01 13:00';
```

Предикатная блокировка ограничивает доступ следующим образом.

- Транзакция A для чтения каких-либо объектов, соответствующих определенному условию, как в вышеприведенном запросе `SELECT`, должна установить предикатную разделяемую блокировку в соответствии с условиями запроса. Если у другой транзакции B уже установлена монопольная блокировка на любой соответствующий этим условиям объект, то транзакции A придется дождаться освобождения блокировки транзакцией B и лишь затем выполнить свой запрос.
- Транзакция A для вставки, обновления или удаления каких-либо объектов должна сначала проверить, попадает ли старое или новое значение под условия какой-либо существующей предикатной блокировки. При наличии таковой у удерживаемой транзакции B транзакция A сможет продолжить работу только после фиксации или прерывания транзакции B.

Основная идея заключается в применимости предикатных блокировок даже к тем объектам, которые еще не существуют в базе данных, но могут там появиться в будущем (фантомы). Если двухфазная блокировка включает предикатные блокировки, то база данных предотвращает все формы асимметрии записи и других состояний гонки, так что ее изоляцию можно с полным правом назвать сериализуемостью.

Блокировки по диапазону значений индекса

К сожалению, производительность предикатных блокировок оставляет желать лучшего: в случае большого количества блокировок от активных транзакций проверка соответствия блокировок может занимать значительное время. Поэтому большинство баз данных с 2PL на самом деле реализуют *блокировку по диапазону значений индекса* (index-range lock), известную также под названием *блокировки следующего ключа* (next-key locking) и представляющую собой упрощенный аналог предикатной [41, 50].

Можно спокойно упростить предикат, чтобы он соответствовал более широкому множеству объектов. Например, расширить предикатную блокировку для бронирования конференц-зала номер 123 с полудня до часа дня, блокировав бронирование

зала 123 на любое время, или все залы (а не только 123) с полудня до часа дня. Это вполне безопасно, ведь любые подходящие под изначальный предикат операции записи определенно будут соответствовать также и расширенной версии.

В базе бронирования конференц-залов будет, вероятно, индекс по столбцу `room_id` и/или индексы по столбцам `start_time` и `end_time` (в противном случае вышеприведенный запрос станет работать на большой базе данных очень медленно).

- Допустим, у нас есть индекс по столбцу `room_id`, и база использует его для поиска существующих операций бронирования зала 123. БД может просто связать с этим элементом индекса разделяемую блокировку, служащую признаком поиска транзакцией операций бронирования зала 123.
- Если же база данных использует для поиска существующих операций бронирования индекс по времени, то может связать разделяемую блокировку с диапазоном значений в этом индексе, которая будет служить признаком поиска транзакцией операций бронирования, пересекающихся по времени с периодом от полудня до часа дня 1 января 2018 года.

В любом случае расширение условий поиска связывается с одним из индексов. Далее, если другой транзакции понадобится вставить, обновить или удалить бронирование для того же зала и/или пересекающегося промежутка времени, то ей придется обновить ту же часть индекса. В процессе этого она наткнется на разделяемую блокировку и будет вынуждена дождаться ее освобождения.

Такой подход обеспечивает эффективную защиту от фантомов и асимметрий записи. Блокировки по диапазону значений индекса не столь точны, как предикатные блокировки (они могут блокировать более широкий диапазон объектов, чем необходимо для поддержания сериализуемости), но это хороший компромисс в силу значительно более низких затрат.

Если подходящего индекса, с которым можно связать блокировку по диапазону, нет, база всегда способна вернуться к разделяемой блокировке на всю таблицу. Производительность пострадает, поскольку все другие транзакции будут вынуждены прекратить операции записи в данную таблицу, но это вполне безопасный запасной вариант.

Сериализуемая изоляция снимков состояния (SSI)

В этой главе мы нарисовали довольно мрачную картину управления конкурентным доступом в базах данных. С одной стороны, имеющиеся реализации сериализуемости или демонстрируют низкую производительность (двухфазная блокировка), или плохо масштабируются (последовательное выполнение). С другой — слабые уровни изоляции показывают высокую производительность, но подвержены различным состояниям гонки (потеря обновлений, асимметрия записи, фантомы и т. д.). Так что же, сериализуемая изоляция и хорошая производительность — вещи принципиально взаимоисключающие?

Возможно, нет: очень многообещающим представляется алгоритм под названием *«сериализуемая изоляция снимков состояния»* (serializable snapshot isolation, SSI). Он обеспечивает полную сериализуемость за счет лишь небольшого снижения производительности по сравнению с обычной изоляцией снимков состояния. SSI — относительно новый метод: он был впервые описан в 2008 году [40] и представляет собой предмет диссертации Майкла Кэхилла (Michael Cahill) на соискание ученой степени PhD [51].

Сегодня SSI используется в одноузловых базах данных (уровень изоляции SERIALAZABLE в PostgreSQL, начиная с версии 9.1 [41]) и распределенных (FoundationDB задействует схожий алгоритм). SSI, столь «юному» по сравнению с другими механизмами управления конкурентным доступом, все еще приходится доказывать свою производительность на практике, но он достаточно быстр, чтобы в будущем стать новым стандартом.

Пессимистическое и оптимистическое управление конкурентным доступом

Двухфазная блокировка представляет собой так называемый механизм пессимистического управления конкурентным доступом: он основан на следующем принципе: если что-то может потенциально пойти не так (о чем сигнализирует удерживаемая другой транзакцией блокировка), то лучше подождать нормализации ситуации, прежде чем выполнять какие-либо действия. Это напоминает *взаимоисключающие блокировки* (mutual exclusion), используемые для защиты данных в многопоточном программировании.

Последовательное выполнение в некотором смысле предельно пессимистично: по сути, оно эквивалентно тому, что каждая транзакция удерживает монопольную блокировку на всю базу данных (или всю секцию базы) на все время выполнения транзакции. Пессимистичность компенсируется максимальным ускорением отдельных транзакций, поэтому блокировки приходится удерживать только очень короткий период времени.

Напротив, сериализуемая изоляция снимков состояния представляет собой *оптимистический* метод управления конкурентным доступом. «Оптимистический» в этом контексте означает следующее: вместо блокировки в случае потенциально опасных действий транзакции просто продолжают выполняться в надежде, что все будет хорошо. При фиксации транзакции база данных проверяет, не случилось ли чего-то плохого (например, не была ли нарушена изоляция). Если да, то транзакция прерывается и ее выполнение приходится повторять еще раз. Допускается фиксация только выполненных сериализуемым образом транзакций.

Оптимистическое управление конкурентным доступом — довольно старая идея [52], о его достоинствах и недостатках спорят уже долгое время [53]. В случае сильной конкурентности (мнжество транзакций пытаются получить доступ к одним и тем же объектам) производительность его невысока, так как приходится прерывать большой

процент транзакций. Если система уже и так близка к своей максимальной пропускной способности, то нагрузка в виде дополнительных повторно выполняемых транзакций может ухудшить производительность.

Однако если резерв возможностей системы достаточно велик, а конкуренция между транзакциями не слишком высока, то методы оптимистического управления конкурентным доступом демонстрируют более высокую производительность, чем пессимистического. Конкуренцию можно понизить с помощью коммутативных атомарных операций: например, при конкурентном увеличении значения счетчика несколькими транзакциями порядок выполнения операций не играет роли (конечно, если не производится чтение счетчика в той же транзакции), так что они не конфликтуют.

Как ясно из названия, SSI основана на изоляции снимков состояния, то есть все операции чтения в пределах транзакции выполняются на основе согласованного снимка состояния базы данных (см. подраздел «Изоляция снимков состояния и воспроизводимое чтение» раздела 7.2). Это основное отличие между ранее существовавшими методами оптимистического управления конкурентным доступом. SSI добавляет поверх изоляции снимков состояния алгоритм для обнаружения конфликтов сериализации между операциями записи и принятия решения о том, какие транзакции прервать.

Решения, основанные на устаревших исходных условиях

При обсуждении выше асимметрии записи при изоляции снимков состояния (см. подраздел «Асимметрия записи и фантомы» раздела 7.2) мы регулярно наблюдали один и тот же паттерн: транзакция читает данные из базы, исследует результаты запроса и принимает решение о каком-либо действии (записи в базу) на основе прочитанных данных. Однако при изоляции снимков состояния полученные в результате исходного запроса данные могут оказаться уже неактуальными на момент фиксации транзакции, поскольку были изменены в промежутке.

Другими словами, транзакция предпринимает какие-либо действия на основе определенных *исходных условий* (фактов, истинных по состоянию на момент начала транзакция, например: «На дежурстве сейчас два врача»). Позднее, к моменту фиксации транзакции, первоначальные данные могли поменяться, и исходные условия более не соответствуют действительности.

База данных не знает, как приложение использует результаты выполняемых к ней запросов (например: «Сколько врачей сейчас дежурят?»). Ради безопасности она предполагает, что любые изменения в результатах запроса (исходных условиях) означают потенциальную неправильность выполненных текущей транзакцией операций записи. Другими словами, возможна причинно-следственная связь между запросами и выполняемыми транзакцией операциями записи. Для обеспечения сериализуемости БД должна выявлять ситуации, когда транзакция

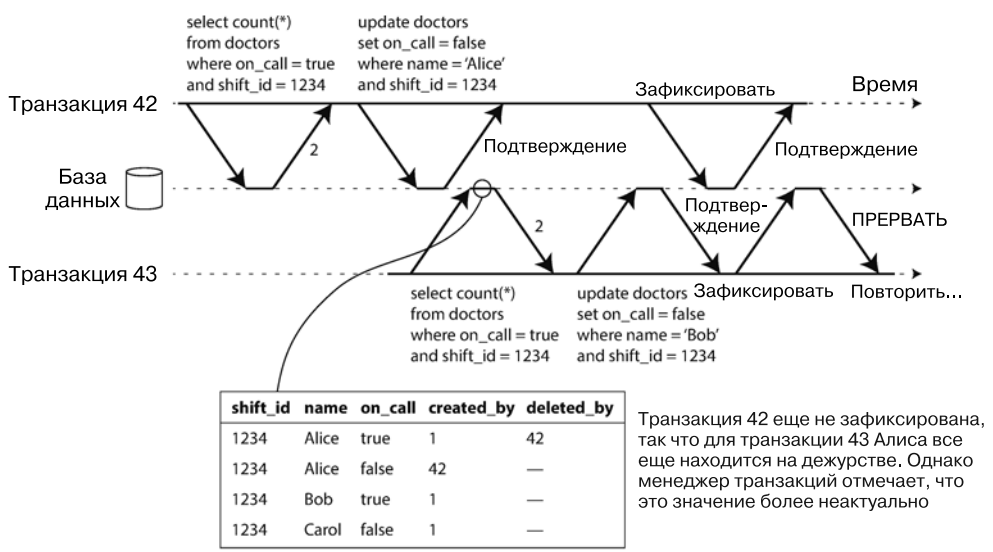
могла выполнять действия на основе устаревших исходных условий, и прерывать такие транзакции.

Но откуда база данных знает, изменились ли результаты запроса? Рассмотрим два случая:

- выявление операций чтения устаревших версий MVCC-объектов (перед чтением произошла незафиксированная операция записи);
- выявление операций записи, влияющих на предшествующие операции чтения (операция записи произошла после чтения).

Выявление операций чтения устаревших версий MVCC объектов

Как вы помните, изоляция снимков состояния обычно реализуется с помощью многоверсионного управления конкурентным доступом (MVCC) (рис. 7.10). Транзакция, читающая из согласованного снимка состояния в базе данных MVCC, игнорирует все операции записи, которые были выполнены транзакциями, еще не зафиксированными на момент получения снимка состояния. На рис. 7.10 транзакция 43 видит, что Алиса находится на дежурстве (`on_call = true`), поскольку транзакция 42 (изменившая статус Алисы) не зафиксирована. Однако на момент фиксации транзакции 43 транзакция 42 уже зафиксирована. Это значит, что операция записи, проигнорированная при чтении из согласованного снимка состояния, теперь уже вступила в силу и исходные условия транзакции 43 более не соответствуют действительности.



shift_id	name	on_call	created_by	deleted_by
1234	Alice	true	1	42
1234	Alice	false	42	—
1234	Bob	true	1	—
1234	Carol	false	1	—

Рис. 7.10. Выявление чтения транзакцией устаревших значений из снимка состояния MVCC

Для предотвращения этой аномалии база данных должна отслеживать случаи игнорирования транзакцией вследствие правил видимости MVCC операций записи других транзакций. При попытке фиксации транзакции база проверяет, были ли зафиксированы какие-либо из проигнорированных операций записи. Если да, то транзакцию необходимо прервать.

Почему необходимо ждать до момента фиксации? Почему не прервать транзакцию 43 сразу же после обнаружения устаревшей операции чтения? Если транзакция 43 выполняла только чтение, то ее не нужно прерывать, поскольку опасности асимметрии записи нет. На момент чтения данных транзакцией 43 база еще не знает, станет ли эта транзакция в дальнейшем записывать данные. Более того, возможно, что транзакцию 42 придется прервать или она еще не будет зафиксирована на момент фиксации транзакции 43, поэтому прочитанные данные все же окажутся неустаревшими. SSI благодаря отказу от ненужных прерываний транзакций сохраняет имеющуюся в изоляции снимков состояния поддержку длительных операций чтения данных из согласованного снимка состояния.

Выявление операций записи, влияющих на предшествующие операции чтения

Второй случай — когда другая транзакция модифицирует данные после их чтения. Этот случай показан на рис. 7.11.

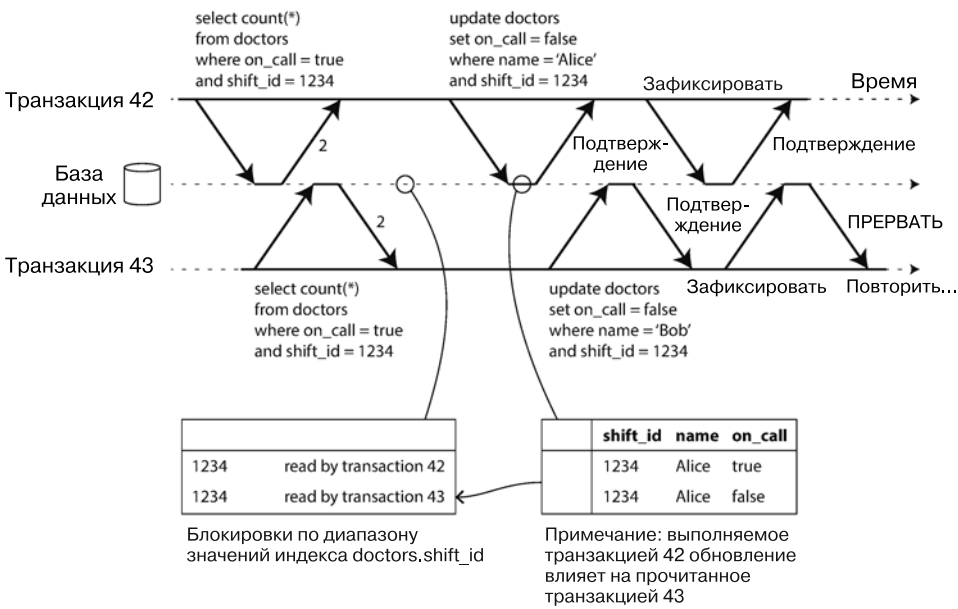


Рис. 7.11. Выявление при сериализуемой изоляции снимков состояния случаев модификации транзакцией данных, прочитанных другой транзакцией

В контексте двухфазной блокировки мы обсуждали блокировки по диапазону значений индекса (см. одноименный пункт подраздела «Двухфазная блокировка (2PL)» текущего раздела), которые позволяют базе данных блокировать доступ ко всем строкам, соответствующим некоему запросу на поиск, например `WHERE shift_id = 1234`. Можно воспользоваться аналогичным методом и тут, за исключением того, что SSI-блокировки не блокируют другие транзакции.

На рис. 7.11 обе транзакции, 42 и 43, выполняют поиск дежурящих в смену 1234 врачей. При наличии индекса по `shift_id` база может воспользоваться его записью 1234, чтобы отметить факт чтения транзакциями 42 и 43 этих данных (если индекса нет, информацию можно отслеживать на уровне таблицы). Сама информация требуется только временно: после завершения выполнения транзакции (ее фиксации или прерывания) и завершения всех конкурентных транзакций БД может спокойно «забыть» о том, какие данные читались этой транзакцией.

Транзакция, записывающая данные в базу, должна выполнить поиск по индексам всех остальных транзакций, которые недавно прочитали соответствующие данные. Этот процесс подобен установке блокировки на соответствующий диапазон ключей, но вместо блокировки до момента фиксации читающих эти данные транзакций блокировка служит в качестве «ловушки» — уведомляет транзакции, что прочитанные ими данные могут оказаться неактуальными.

На рис. 7.11 транзакция 43 уведомляет транзакцию 42, что прочитанные ею в предыдущей операции чтения данные устарели и наоборот. Первой выполняется фиксация транзакции 42, и она проходит успешно: хотя операция записи транзакции 43 влияет на 42, транзакция 43 пока еще не зафиксирована, поэтому данная операция записи еще не вступила в силу. Однако на момент фиксации транзакции 43 конфликтующая операция записи транзакции 42 уже была зафиксирована, так что транзакцию 43 придется прервать.

Производительность сериализуемой изоляции снимков состояния

Как всегда, на реальную производительность алгоритма влияет множество технических нюансов. Например, необходимо выбрать оптимальный уровень детализации при отслеживании операций чтения и записи транзакций. Более детальное отслеживание действий каждой транзакции приведет к высокой точности принятия решений о прерывании транзакций, но издержки этого могут оказаться существенными. Менее детальное отслеживание будет работать быстрее, но способно привести к прерыванию лишних транзакций, которые, строго говоря, прерывать не было нужды.

В некоторых случаях нет ничего страшного в чтении транзакцией данных, переписанных другой транзакцией: в зависимости от неких событий иногда может оказаться, что результат выполнения транзакции все равно остался сериализуемым.

Эта теория используется в PostgreSQL для снижения количества прерываемых транзакций [11, 41].

По сравнению с двухфазной блокировкой огромное преимущество сериализуемой изоляции снимков состояния — отсутствие необходимости приостанавливать выполнение одной транзакции до момента освобождения блокировок, удерживаемых другими транзакциями. Как и при обычной изоляции снимков состояния, записывающие транзакции не блокируют выполнение читающих транзакций и наоборот. Такая конструктивная особенность обеспечивает более высокую степень предсказуемости и стабильности времени ожидания запроса. В частности, запросы только для чтения могут выполняться на согласованном снимке состояния без необходимости в каких-либо блокировках, что очень удобно при значительной нагрузке по чтению.

По сравнению с последовательным выполнением сериализуемая изоляция снимков состояния не ограничивается пропускной способностью одного ядра процессора: FoundationDB распределяет задачу обнаружения конфликтов сериализации по нескольким машинам, обеспечивая масштабируемость до очень высоких уровней пропускной способности. Хотя данные можно секционировать по нескольким машинам, транзакции могут читать и записывать данные в различных секциях с сохранением гарантий сериализуемой изоляции [54].

Частота прерываний транзакций существенно влияет на общую производительность SSI. Например, читающая и записывающая данные на протяжении длительного времени транзакция, вероятнее всего, столкнется с конфликтами и будет прервана, поэтому SSI требует, чтобы транзакции чтения/записи были достаточно короткими (длительные транзакции только для чтения вполне допустимы). Однако SSI, скорее всего, не так чувствительна к медленным транзакциям, как двухфазная блокировка или последовательное выполнение.

7.4. Резюме

Транзакции представляют собой слой абстракции, благодаря которому приложение может не обращать внимания на отдельные проблемы конкурентного доступа и некоторые виды сбоев аппаратного и программного обеспечения. Широкий класс ошибок сводится к простому *прерыванию транзакции*, а приложению достаточно просто повторить выполнение транзакции.

В этой главе мы рассмотрели множество примеров проблем, которые можно предотвратить с помощью транзакций. Далеко не все приложения подвержены этим проблемам: приложение с очень простым паттерном доступа, например, читающее и добавляющее только одну запись, вероятно, может обойтись без транзакций. Однако при более сложных паттернах доступа транзакции способны существенно сократить количество возможных сценариев ошибок, которые приходится учитывать.

Без транзакций различные сценарии ошибок (сбои процессов, разрывы сети, отключения электропитания, переполнение диска, неожиданная конкурентность и т. д.) способны привести к несогласованности данных. Например, денормализованные данные легко могут оказаться не согласованными с исходными. Без транзакций было бы очень сложно предугадать влияние, оказываемое на БД в результате запутанных интерактивных обращений.

В этой главе мы рассмотрели в деталях вопрос конкурентного доступа. Мы обсудили несколько широко используемых уровней изоляции, в частности *чтение зафиксированных данных*, *изоляцию снимков состояния*[1] и *сериализуемость*. Мы изучили характеристики этих уровней изоляции на разнообразных примерах состояний гонки.

- *«Грязные» операции чтения*. Клиент читает записанные другим клиентом данные до их фиксации. Уровень изоляции чтения зафиксированных данных и более сильные предотвращают «грязные» операции чтения.
- *«Грязные» операции записи*. Клиент перезаписывает данные, которые другой клиент записал, но еще не зафиксировал. Практически все реализации транзакций предотвращают «грязные» операции записи.
- *Асимметрия чтения (невоспроизводимое чтение)*. Клиент видит различные части базы данных по состоянию на разные моменты времени. Чаще всего такую проблему предотвращают с помощью изоляции снимков состояния, при которой транзакция читает данные из согласованного снимка состояния, соответствующего определенному моменту времени. Обычно это реализуется благодаря *многоверсионному управлению конкурентным доступом* (MVCC).
- *Потерянные обновления*. Два клиента выполняют в конкурентном режиме цикл чтения — изменения — записи. Один переписывает записанные другим данные без учета внесенных им изменений, так что данные оказываются потеряны. Некоторые реализации изоляции снимков состояния предотвращают эту аномалию автоматически, а в других требуется установка блокировки вручную (`SELECT FOR UPDATE`).
- *Асимметрия записи*. Транзакция читает какие-либо данные, принимает на основе прочитанного значения решение о дальнейших действиях и выполняет операцию записи в базу данных. Однако на момент ее выполнения исходные условия, на основе которых принималось решение, более не соответствуют действительности. Эту аномалию предотвращает только сериализуемость.
- *Фантомные чтения*. Транзакция читает объекты, соответствующие определенному условию поиска. Другой клиент выполняет операцию записи, которая каким-то образом влияет на результаты этого поиска. Изоляция снимков состояния предотвращает непосредственно фантомные чтения, но фантомы в контексте асимметрии записи требуют отдельной обработки, например блокировок по диапазону значений индекса.

[1] Иногда называемую также воспроизводимым чтением (repeatable read).

Слабые уровни изоляции защищают от некоторых из этих аномалий, но другие вам, как разработчику приложения, придется обрабатывать вручную (например, с помощью явной блокировки). Только изоляция уровня сериализуемых транзакций предотвращает все подобные проблемы. Мы обсудили три различных подхода к реализации сериализуемых транзакций.

- *По-настоящему последовательное выполнение транзакций*. Если вы можете сделать отдельные транзакции очень быстрыми, причем количество транзакций, обрабатываемых за единицу времени на одном ядре CPU, достаточно невелико, то для обработки этот вариант окажется простым и эффективным.
- *Двухфазная блокировка*. На протяжении десятилетий она была стандартным способом обеспечения сериализуемости, но многие приложения стараются ее не использовать из-за плохих показателей производительности.
- *Сериализуемая изоляция снимков состояния (SSI)*. Довольно свежий алгоритм, лишенный практически всех недостатков предыдущих подходов. В нем используется оптимистический подход, благодаря чему транзакции выполняются без блокировок. Перед фиксацией транзакции выполняется проверка, и если выполнение было несериализуемым, то транзакция прерывается без фиксации.

В примерах из этой главы применялась реляционная модель данных. Однако, как обсуждалось в пункте «Востребованность многообъектных транзакций» подраздела «Однообъектные и многообъектные операции» раздела 7.1, транзакции — полезная возможность баз данных вне зависимости от используемой модели.

В этой главе мы исследовали идеи и алгоритмы в основном в контексте работающих на одной машине баз данных. Выполнение транзакций в распределенных базах представляет собой набор новых интересных и трудных задач, которые мы обсудим в следующих двух главах.

7.5. Библиография

1. *Chamberlin D. D., Astrahan M. M., Blasgen M. W., et al.* A History and Evaluation of System R // Communications of the ACM, volume 24, number 10, pages 632–646, October 1981 [Электронный ресурс]. — Режим доступа: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.348&rep=rep1&type=pdf.
2. *Gray J. N., Lorie R. A., Putzolu G. R., Traiger I. L.* Granularity of Locks and Degrees of Consistency in a Shared Data Base // Modelling in Data Base Management Systems: Proceedings of the IFIP Working Conference on Modelling in Data Base Management Systems, edited by G. M. Nijssen, pages 364– 394, Elsevier/North Holland Publishing, 1976. Also in Readings in Database Systems, 4th edition, edited by Joseph M. Hellerstein and Michael Stonebraker, MIT Press, 2005. http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.92.8248&rep=rep1&type=pdf.

3. *Eswaran K. P., Gray J. N., Lorie R. A., Traiger I. L.* The Notions of Consistency and Predicate Locks in a Database System // Communications of the ACM, volume 19, number 11, pages 624–633, November 1976 [Электронный ресурс]. — Режим доступа: http://jimgray.azurewebsites.net/papers/on%20the%20notions%20of%20consistency%20and%20predicate%20locks%20in%20a%20database%20system%20cacm.pdf.
4. ACID Transactions Are Incredibly Helpful // FoundationDB, LLC, 2013 [Электронный ресурс]. — Режим доступа: http://web.archive.org/web/20150320053809/https://foundationdb.com/acid-claims.
5. *Cook J. D.* ACID Versus BASE for Database Transactions. July 6, 2009 [Электронный ресурс]. — Режим доступа: https://www.johndcook.com/blog/2009/07/06/brewer-cap-theorem-base/.
6. *Clarke G.* NoSQL's CAP Theorem Busters: We Don't Drop ACID. November 22, 2012 [Электронный ресурс]. — Режим доступа: http://www.theregister.co.uk/2012/11/22/foundationdb_fear_of_cap_theorem/.
7. *Härder T., Reuter A.* Principles of Transaction-Oriented Database Recovery // ACM Computing Surveys, volume 15, number 4, pages 287–317, December 1983 [Электронный ресурс]. — Режим доступа: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.2812&rep=rep1&type=pdf.
8. *Bailis P., Fekete A., Ghodsi A., et al.* HAT, not CAP: Towards Highly Available Transactions // 14th USENIX Workshop on Hot Topics in Operating Systems (HotOS), May 2013 [Электронный ресурс]. — Режим доступа: http://www.bailis.org/papers/hat-hotos2013.pdf.
9. *Fox A., Gribble S. D., Chawathe Y., et al.* Cluster-Based Scalable Network Services // at 16th ACM Symposium on Operating Systems Principles (SOSP), October 1997 [Электронный ресурс]. — Режим доступа: https://people.eecs.berkeley.edu/~brewer/cs262b/TACC.pdf.
10. *Bernstein P. A., Hadzilacos V., Goodman N.* Concurrency Control and Recovery in Database Systems. — Addison-Wesley, 1987. https://www.microsoft.com/en-us/research/people/philbe/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fpeople%2Fphilbe%2Fccontrol.aspx.
11. *Fekete A., Liarokapis D., O'Neil E., et al.* Making Snapshot Isolation Serializable // ACM Transactions on Database Systems, volume 30, number 2, pages 492–528, June 2005 [Электронный ресурс]. — Режим доступа: https://www.cse.iitb.ac.in/infolab/Data/Courses/CS632/2009/Papers/p492-fekete.pdf.
12. *Zheng M., Tucek J., Qin F., Lillibridge M.* Understanding the Robustness of SSDs Under Power Fault // 11th USENIX Conference on File and Storage Technologies (FAST), February 2013 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/system/files/conference/fast13/fast13-final80.pdf.
13. *Denness L.* SSDs: A Gift and a Curse. June 2, 2015 [Электронный ресурс]. — Режим доступа: https://laur.ie/blog/2015/06/ssds-a-gift-and-a-curse/.

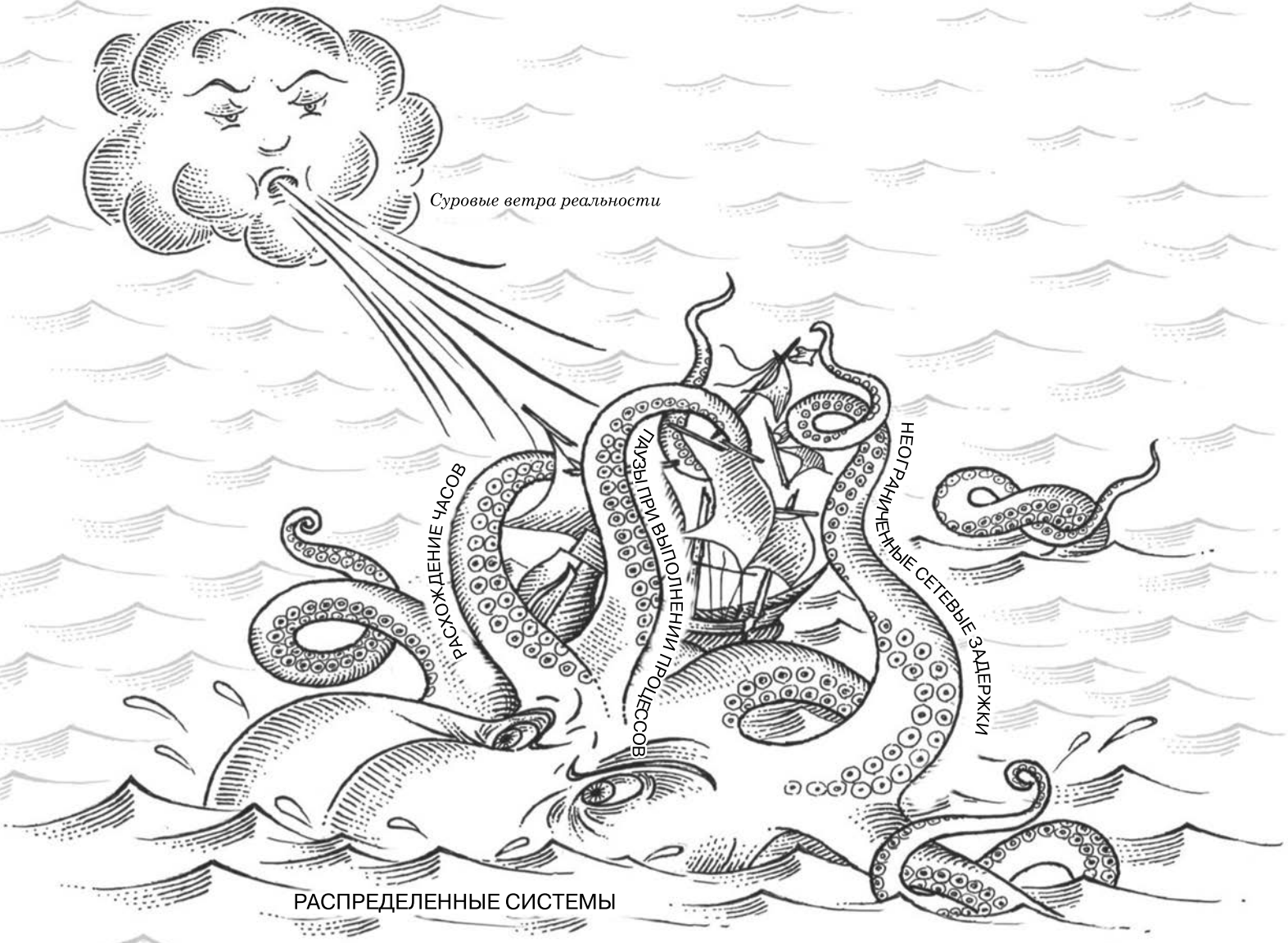
14. *Surak A.* When Solid State Drives Are Not That Solid. June 15, 2015 [Электронный ресурс]. — Режим доступа: https://blog.algolia.com/when-solid-state-drives-are-not-that-solid/.
15. *Pillai T. S., Chidambaram V., Alagappan R., et al.* All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications // 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), October 2014 [Электронный ресурс]. — Режим доступа: http://research.cs.wisc.edu/wind/Publications/alice-osdi14.pdf.
16. *Siebenmann C.* Unix's File Durability Problem. April 14, 2016 [Электронный ресурс]. — Режим доступа: https://utcc.utoronto.ca/~cks/space/blog/unix/FileSyncProblem.
17. *Bairavasundaram L. N., Goodson G. R., Schroeder B., et al.* An Analysis of Data Corruption in the Storage Stack // 6th USENIX Conference on File and Storage Technologies (FAST), February 2008 [Электронный ресурс]. — Режим доступа: http://research.cs.wisc.edu/adsl/Publications/corruption-fast08.pdf.
18. *Schroeder B., Lagisetty R., Merchant A.* Flash Reliability in Production: The Expected and the Unexpected // 14th USENIX Conference on File and Storage Technologies (FAST), February 2016 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/conference/fast16/technical-sessions/presentation/schroeder.
19. *Allison D.* SSD Storage — Ignorance of Technology Is No Excuse. March 24, 2015. [Электронный ресурс]. — Режим доступа: https://blog.korelogic.com/blog/2015/03/24.
20. *Scherer D.* Those Are Not Transactions (Cassandra 2.0). September 6, 2013 [Электронный ресурс]. — Режим доступа: http://web.archive.org/web/20150526065247/http://blog.foundationdb.com/those-are-not-transactions-cassandra-2-0.
21. *Kingsbury K.* Call Me Maybe: Cassandra. September 24, 2013 [Электронный ресурс]. — Режим доступа: https://aphyr.com/posts/294-call-me-maybe-cassandra/.
22. ACID Support in Aerospike // Aerospike, Inc., June 2014 [Электронный ресурс]. — Режим доступа: https://www.aerospike.com/docs/architecture/assets/AerospikeACIDSupport.pdf.
23. *Kleppmann M.* Hermitage: Testing the 'I' in ACID. November 25, 2014 [Электронный ресурс]. — Режим доступа: http://martin.kleppmann.com/2014/11/25/hermitage-testing-the-i-in-acid.html.
24. *D'Agosta T.* BTC Stolen from Poloniex. March 4, 2014 [Электронный ресурс]. — Режим доступа: https://bitcointalk.org/index.php?topic=499580.
25. bitcointhief2: How I Stole Roughly 100 BTC from an Exchange and How I Could Have Stolen More! February 2, 2014 [Электронный ресурс]. — Режим доступа: https://www.reddit.com/r/Bitcoin/comments/1wtbiu/how_i_stole_roughly_100_btc_from_an_exchange_and/.
26. *Jorwekar S., Fekete A., Ramamritham K., Sudarshan S.* Automating the Detection of Snapshot Isolation Anomalies // 33rd International Conference on Very Large Data

Bases (VLDB), September 2007 [Электронный ресурс]. — Режим доступа: http://www.vldb.org/conf/2007/papers/industrial/p1263-jorwekar.pdf.

27. *Melanson M.* Transactions: The Limits of Isolation. March 20, 2014 [Электронный ресурс]. — Режим доступа: http://www.michaelmelanson.net/2014/03/20/transactions/.
28. *Berenson H., Bernstein P. A., Gray J. N., et al.* A Critique of ANSI SQL Isolation Levels // ACM International Conference on Management of Data (SIGMOD), May 1995 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/a-critique-of-ansi-sql-isolation-levels/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F69541%2Ftr-95-51.pdf.
29. *Adya A.* Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions // PhD Thesis, Massachusetts Institute of Technology, March 1999 [Электронный ресурс]. — Режим доступа: http://pmg.csail.mit.edu/papers/adya-phd.pdf.
30. *Bailis P., Davidson A., Fekete A., et al.* Highly Available Transactions: Virtues and Limitations (Extended Version) // 40th International Conference on Very Large Data Bases (VLDB), September 2014 [Электронный ресурс]. — Режим доступа: https://arxiv.org/pdf/1302.0309.pdf.
31. *Momjian B.* MVCC Unmasked. July 2014 [Электронный ресурс]. — Режим доступа: http://momjian.us/main/presentations/internals.html#mvcc.
32. *Gurusami A.* Repeatable Read Isolation Level in InnoDB — How Consistent Read View Works. January 15, 2013 [Электронный ресурс]. — Режим доступа: https://blogs.oracle.com/mysqlinnodb/entry/repeatable_read_isolation_level_in.
33. *Prokopov N.* Unofficial Guide to Datomic Internals. May 6, 2014 [Электронный ресурс]. — Режим доступа: http://tonsky.me/blog/unofficial-guide-to-datomic-internals/.
34. *Schwartz B.* Immutability, MVCC, and Garbage Collection. December 28, 2013 [Электронный ресурс]. — Режим доступа: https://www.xaprb.com/blog/2013/12/28/immutability-mvcc-and-garbage-collection/.
35. *Anderson J. C., Lehnardt J., Slater N.* CouchDB: The Definitive Guide. — O'Reilly Media, 2010.
36. *Mukherjee R.* Isolation in DB2 (Repeatable Read, Read Stability, Cursor Stability, Uncommitted Read) with Examples. July 4, 2013 [Электронный ресурс]. — Режим доступа: http://mframes.blogspot.com.by/2013/07/isolation-in-cursor.html.
37. *Hilker S.* Cursor Stability (CS) — IBM DB2 Community. March 14, 2013 [Электронный ресурс]. — Режим доступа: http://www.toadworld.com/platforms/ibmdb2/w/wiki/6661.cursor-stability-cs.
38. *Wiger N.* An Atomic Rant. February 18, 2010 [Электронный ресурс]. — Режим доступа: http://www.nateware.com/an-atomic-rant.html#.We8bK3ZLfIU.

39. *Jacobson J.* Riak 2.0: Data Types. March 23, 2014 [Электронный ресурс]. — Режим доступа: http://blog.joeljacobson.com/riak-2-0-data-types/.
40. *Cahill M. J., Röhm U., Fekete A.* Serializable Isolation for Snapshot Databases // ACM International Conference on Management of Data (SIGMOD), June 2008 [Электронный ресурс]. — Режим доступа: http://www.cs.nyu.edu/courses/fall12/CSCI-GA.2434-001/p729-cahill.pdf.
41. *Ports D. R. K., Grittner K.* Serializable Snapshot Isolation in PostgreSQL // 38th International Conference on Very Large Databases (VLDB), August 2012 [Электронный ресурс]. — Режим доступа: https://drkp.net/papers/ssi-vldb12.pdf.
42. *Andrews T.* Enforcing Complex Constraints in Oracle. October 15, 2004. [Электронный ресурс]. — Режим доступа: http://tonyandrews.blogspot.com.by/2004/10/enforcing-complex-constraints-in.html.
43. *Terry D. B., Theimer M. M., Petersen K., et al.* Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System // 15th ACM Symposium on Operating Systems Principles (SOSP), December 1995 [Электронный ресурс]. — Режим доступа: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.141.7889&rep=rep1&type=pdf.
44. *Fredericks G.* Postgres Serializability Bug. September 2015 [Электронный ресурс]. — Режим доступа: https://github.com/gfredericks/pg-serializability-bug.
45. *Stonebraker M., Madden S., Abadi D. J., et al.* The End of an Architectural Era (It's Time for a Complete Rewrite) // 33rd International Conference on Very Large Data Bases (VLDB), September 2007 [Электронный ресурс]. — Режим доступа: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.3697&rep=rep1&type=pdf.
46. *Hugg J.* H-Store/VoltDB Architecture vs. CEP Systems and Newer Streaming Architectures // Data @Scale Boston, November 2014 [Электронный ресурс]. — Режим доступа: https://www.youtube.com/watch?v=hD5M4a1UVz8.
47. *Kallman R., Kimura H., Natkins J., et al.* H-Store: A High-Performance, Distributed Main Memory Transaction Processing System // Proceedings of the VLDB Endowment, volume 1, number 2, pages 1496–1499, August 2008 [Электронный ресурс]. — Режим доступа: http://www.vldb.org/pvldb/1/1454211.pdf.
48. *Hickey R.* The Architecture of Datomic. November 2, 2012 [Электронный ресурс]. — Режим доступа: https://www.infoq.com/articles/Architecture-Datomic.
49. *Hugg J.* Debunking Myths About the VoltDB In-Memory Database. May 12, 2014. [Электронный ресурс]. — Режим доступа: https://www.voltdb.com/blog/.
50. *Hellerstein J. M., Stonebraker M., Hamilton J.* Architecture of a Database System // Foundations and Trends in Databases, volume 1, number 2, pages 141–259, November 2007 [Электронный ресурс]. — Режим доступа: http://db.cs.berkeley.edu/papers/fntdb07-architecture.pdf.
51. *Cahill M. J.* Serializable Isolation for Snapshot Databases // PhD Thesis, University of Sydney, July 2009 [Электронный ресурс]. — Режим доступа: http://cahill.net.au/wp-content/uploads/2010/02/cahill-thesis.pdf.

52. *Badal D. Z.* Correctness of Concurrency Control and Implications in Distributed Databases // 3rd International IEEE Computer Software and Applications Conference (COMPSAC), November 1979 [Электронный ресурс]. — Режим доступа: http://ieeexplore.ieee.org/abstract/document/762563/?reload=true.
53. *Agrawal R., Carey M. J., Livny M.* Concurrency Control Performance Modeling: Alternatives and Implications // ACM Transactions on Database Systems (TODS), volume 12, number 4, pages 609–654, December 1987 [Электронный ресурс]. — Режим доступа: https://people.eecs.berkeley.edu/~brewer/cs262/ConcControl.pdf.
54. *Rosenthal D.* Databases at 14.4MHz. December 10, 2014 [Электронный ресурс]. — Режим доступа: http://web.archive.org/web/20150427041746/http://blog.foundationdb.com/databases-at-14.4mhz.


Суровые ветра реальности
РАСХОЖДЕНИЕ ЧАСОВ
ПАУЗЫ ПРИ ВЫПОЛНЕНИИ ПРОЦЕССОВ
НЕОГРАНИЧЕННЫЕ СЕТЕВЫЕ ЗАДЕРЖКИ
РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ

8 Проблемы распределенных систем

Привет, мы только что встретились
Запаздывание сети ужасное
Но вот мои данные
Сохрани их, что ли

Кайл Кингсбери.
Карли Рэй Джепсен и опасности
сетевого секционирования (2013)

В нескольких последних главах постоянно звучала тема действий в ситуациях, когда что-то пошло не так. Например, мы обсуждали восстановление реплик после отказов (см. подраздел «Перебои в обслуживании узлов» раздела 5.1), задержку репликации (см. раздел 5.2) и конкурентный доступ для транзакций (см. раздел 7.2). Чем лучше мы разберемся в различных граничных случаях, встречающихся в реальных системах, тем лучше сможем их обрабатывать.

Однако, хотя мы немало говорили о сбоях, последние несколько глав все равно были слишком оптимистичны. Реальность еще суровее. Поэтому сейчас доведем наш пессимизм до предела и предположим, что *все потенциальные проблемы обязательно произойдут*[1]. (Опытные сисадмины подтвердят разумность этого предположения. Если вы очень попросите, то они даже расскажут вам несколько жутких историй и покажут «боевые шрамы былых битв».)

Работа с распределенными системами принципиально отличается от написания программного обеспечения для отдельного компьютера — и главное отличие

[1] За одним исключением: мы предполагаем, что эти сбои не византийские (см. подраздел «Византийские сбои» раздела 8.4).

состоит во множестве новых и захватывающих проблем [1, 2]. В этой главе мы ознакомимся с возникающими на практике проблемами и разберемся, на что можно рассчитывать, а на что — нет.

В конце концов, наша задача как инженеров — создавать системы, выполняющие свое предназначение (то есть предоставляют ожидаемые пользователями гарантии), несмотря на все возможные проблемы. В главе 9 мы рассмотрим некоторые примеры алгоритмов для обеспечения подобных гарантий в распределенной системе. Но сначала, в данной главе, необходимо разобраться, с какими задачами нам придется столкнуться.

Эта глава представляет собой совершенно пессимистический и депрессивный обзор всех проблем, какие только могут возникнуть в распределенной системе. Мы рассмотрим сетевые проблемы (см. раздел 8.2), проблемы с часами и хронометражем (см. раздел 8.3) и обсудим, насколько их можно избежать. В последствиях всех этих проблем достаточно сложно разобраться, так что мы изучим вопрос о том, как оценивать состояние распределенной системы и судить о происходящем в ней (см. раздел 8.4).

8.1. Сбои и частичные отказы

Программа, написанная для работы на одном компьютере, обычно ведет себя довольно предсказуемым образом: или работает, или нет. Программы с ошибками приводят к тому, что компьютер как будто «встал не с той ноги» (проблема, которую часто можно исправить его перезагрузкой), хотя на самом деле это просто последствия работы плохо написанного ПО.

Нет никаких фундаментальных причин, по которым программное обеспечение на отдельном компьютере может быть ненадежным: если аппаратное обеспечение работает нормально, то одни и те же операции всегда возвращают одни результаты (они *детерминированы*). Сбой аппаратного обеспечения (например, порча содержимого оперативной памяти или плохой контакт) обычно приводит к полному отказу всей системы (например, к сбою ядра операционной системы, «синему экрану смерти», невозможности запуска системы). Отдельный компьютер с нормальным программным обеспечением, как правило, или полностью функционирует, или полностью сломан.

Конструкторы компьютеров сделали это умышленно: в случае внутреннего сбоя лучше, чтобы компьютер отказал полностью, а не возвращал неправильные результаты, которые только запутывали и усложняли бы ситуацию. Следовательно, компьютеры инкапсулируют свою запутанную физическую реализацию за идеализированной системной моделью, работающей с математической точностью. Инструкции CPU всегда работают одинаково; записанные в оперативную память или на диск данные остаются неизменными, а не портятся беспорядочно. Такая особенность конструкции ведет свое начало еще с самого первого цифрового компьютера [3].

При создании программного обеспечения, работающего на нескольких компьютерах, соединенных сетью, ситуация кардинально отличается. В распределенных системах мы более не имеем дела с идеализированной системной моделью — у нас нет другого выхода, кроме как встретиться лицом к лицу с запутанной реальностью физического мира. А в нем очень много чего может пойти не так, как предполагалось, как иллюстрирует следующий рассказ [4].

> При всей ограниченности моего опыта, я сталкивался с длительными нарушениями связности сети в отдельном ЦОДе (DC), отказами PDU (распределительных щитов электропитания), сетевых коммутаторов, случайными циклами включения/выключения целых стоек, отказами опорной сети в масштабах всего ЦОДа, а также водителем с гипогликемией, который врезался на своем пикапе Ford в систему HVAC (отопления, вентиляции и кондиционирования воздуха) ЦОДа. А я даже не сисадмин.
>
> *Кода Хейл*

В распределенной системе некоторые части системы могут перестать работать совершенно непредсказуемым образом, хотя остальные функционируют без проблем. Такая ситуация называется *частичным отказом* (partial failure). Трудность состоит в их *недетерминистичности*: любые действия, включающие использование сети и нескольких узлов, могут иногда выполняться нормально, а иногда — причем совершенно неожиданно — нет. Как мы увидим, не всегда *непонятно* даже, успешно ли выполнилась какая-либо операция, так как время движения сообщения по сети тоже недетерминистично!

Именно подобный недетерминизм и вероятность частичных отказов делают работу с распределенными системами столь сложной [5].

Облачные вычисления и организация вычислений на суперкомпьютерах. Существует широкий спектр подходов к организации крупномасштабных вычислительных систем.

- С одной стороны этого спектра — сфера *высокопроизводительных вычислений* (high-performance computing, HPC). Суперкомпьютеры с тысячами процессоров обычно используются для требующих большого объема вычислений научных задач, например прогнозов погоды или молекулярной динамики (моделирования движения атомов и молекул).
- На другом конце спектра располагаются *облачные вычисления* (cloud computing), четкое определение которых отсутствует [6]. Они обычно ассоциируются с мультиарендными ЦОДами, соединенными IP-сетью (обычно Ethernet) серийными компьютерами, эластичным/по требованию распределением ресурсов и автоматизированными системами учета и расчетов.
- Традиционные корпоративные центры обработки данных располагаются примерно посередине между этими предельными случаями.

В каждом из описанных подходов используются совершенно разные методы обработки сбоев. В суперкомпьютерах задания обычно время от времени создают в долговременном хранилище контрольные точки состояния вычислений. В случае сбоя одного из узлов обычно полностью останавливается работа всего кластера. А после починки сбойного узла вычисления продолжаются, начиная с последней контрольной точки [7, 8]. Этим суперкомпьютер больше напоминает одноузловой компьютер, чем распределенную систему: проблема частичных отказов решается путем эскалации их до полных отказов — в случае сбоя любой части система целиком перестает работать (аналогично сбою ядра операционной системы на отдельной машине).

В этой книге мы сосредоточим наше внимание на реализации интернет-сервисов, которые обычно довольно сильно отличаются от суперкомпьютеров.

- Множество интернет-приложений должны работать *онлайн* в том смысле, что от них в любой момент времени требуется низкое время ожидания при обслуживании пользователей. Недоступность сервиса — например, вследствие остановки кластера для ремонта — неприемлема. Напротив, последствия остановки и перезапуска офлайн-заданий (пакетных заданий), допустим, заданий моделирования погоды, относительно невелики.
- Суперкомпьютеры обычно создаются на основе специализированного аппаратного обеспечения для гарантии высокой надежности каждого из узлов, а обмен сообщениями узлов происходит с помощью разделяемой памяти и удаленного прямого доступа к памяти (remote direct memory access, RDMA). С другой стороны, узлы в облачных сервисах строятся на базе серийных машин с эквивалентной производительностью при более низкой стоимости, но и более высокой частоте отказов.
- Сети больших ЦОДов часто основываются на IP и Ethernet и используют топологию Клоза, чтобы предоставить широкую бисекционную полосу пропускания [9]. Суперкомпьютеры часто применяют специализированные сетевые топологии, например многомерные сетки и торы [10], которые обеспечивают более высокую производительность для HPC-нагрузок с известными паттернами обмена сообщениями.
- Чем больше система, тем выше вероятность поломки одного из ее компонентов. С течением времени сбойные компоненты постепенно чинят, а новые — ломаются, но вполне логично предположить: в системе из тысяч узлов *что-то* не работает всегда [7]. Если стратегия обработки ошибок состоит просто в отключении всей системы, то большая система будет проводить почти все свое время в восстановлении после сбоев, а не выполнять полезную работу [8].
- Способность системы продолжать работать в целом при сбоях отдельных узлов — очень полезная возможность для эксплуатации и обслуживания: например, она позволяет выполнять плавающие обновления (см. главу 4), перезапуская по одному узлу за раз, в то время как сервис в целом продолжает

обслуживать пользователей. В облачных средах при плохом функционировании одной из виртуальных машин можно просто ее уничтожить и запросить новую (в надежде, что она будет работать быстрее).

- При географически распределенном развертывании (когда данные хранятся поближе к пользователям ради снижения времени доступа) обмен сообщениями, вероятнее всего, будет выполняться через Интернет — медленный и ненадежный по сравнению с локальными сетями. При работе суперкомпьютеров обычно предполагается, что все их узлы расположены рядом друг с другом.

Чтобы обеспечить работу распределенных систем, нужно допустить возможность частичных отказов и встроить в программное обеспечение механизмы обеспечения отказоустойчивости. Другими словами, следует построить надежную систему из ненадежных компонентов. (Как обсуждалось в разделе 1.2, абсолютной надежности не существует в природе, поэтому необходимо осознавать пределы того, что можно реально обещать.)

Даже в маленьких системах, состоящих всего из нескольких узлов, важно учитывать возможность частичного отказа. В такой системе вполне вероятна нормальная работа большинства компонентов бóльшую часть времени. Однако рано или поздно в некоторых частях системы *возникнут* сбои и программному обеспечению придется их как-то обрабатывать. Обработка сбоев должна быть неотъемлемой частью архитектуры программного обеспечения, а вам (как оператору этого ПО) следует знать, чего ожидать от него в случае сбоя.

Было бы неразумно предполагать, что ошибки — редкое явление, и просто надеяться на лучшее. Важно рассматривать широкий круг возможных сбоев — даже очень маловероятных — и искусственно создавать подобные ситуации при тестировании, выясняя, что произойдет. В распределенных системах подозрительность, пессимизм и паранойя окупаются сполна.

Создание надежной системы из ненадежных компонентов

Вы, вероятно, удивляетесь, как это вообще имеет смысл, — интуитивно кажется, что система надежна настолько, насколько надежен самый ненадежный из ее компонентов (самое *слабое* ее *звено*). Это не так: на самом деле идея создания более надежной системы из менее надежных компонентов весьма давно витает в сфере вычислительной техники [11]. Приведу примеры.

- Коды с коррекцией ошибок позволяют верно передавать цифровые данные по каналам связи, в которых отдельные биты иногда оказываются перепутаны, например, вследствие радиопомех в беспроводных сетях [12].
- IP (Internet protocol, интернет-протокол) ненадежен: возможны потери, задержки, дублирования пакетов и доставка их в неправильном порядке. TCP (transmission control protocol, протокол управления передачей) обеспечивает более надежный

транспортный слой поверх IP: он гарантирует повторную передачу потерянных пакетов, удаление дубликатов пакетов и повторное упорядочение их так, как они были отправлены.

Хотя система и может быть надежнее, чем ее составные части, этой надежности всегда есть предел. Например, коды с коррекцией ошибок способны помочь лишь при небольшом количестве ошибок в отдельных битах, но если сигнал сильно подавляется помехами, то у количества данных, которые можно передать через такой канал связи, существует фундаментальная граница [13]. TCP способен скрыть потерю пакетов, их дублирование и перемешивание, но не может, как по волшебству, ликвидировать сетевые задержки.

Несмотря на то что более надежные системы более высокого уровня и не идеальны, они приносят немалую пользу — берут на себя часть сложных низкоуровневых сбоев, упрощая тем самым выявление и обработку оставшихся. Мы подробнее исследуем этот вопрос в пункте «Сквозной аргумент» подраздела «Сквозные аргументы в базе данных» раздела 12.3.

8.2. Ненадежные сети

Как обсуждалось во введении к части II данной книги, рассматриваемые здесь распределенные системы относятся к *системам без разделения ресурсов* (shared-nothing systems), то есть просто соединенным сетью группам машин. Сеть — единственное средство связи между ними — мы предполагаем, что у каждой из машин есть своя оперативная память и жесткий диск и машины не могут обращаться к дискам и памяти друг друга (разве что выполняя запросы к сервису по сети).

Архитектура без разделения ресурсов — не единственная системная архитектура, но в создании интернет-сервисов она доминирует по нескольким причинам: относительная дешевизна и отсутствие необходимости в специальном аппаратном обеспечении, потенциальность использования широко доступных облачных вычислительных сервисов, возможность достижения высокой надежности за счет избыточности в рамках настройки географически распределенных ЦОДов.

Интернет, как и большинство внутренних сетей ЦОДов (обычно Ethernet), представляет собой *асинхронные сети пакетной передачи данных* (asynchronous packet networks). В подобных сетях узел может отправлять сообщения (пакеты) другим узлам, но сеть не дает гарантий, когда он будет доставлен и будет ли доставлен вообще. При отправке пакета с ожиданием ответа есть вероятность возникновения множества проблем (некоторые из них показаны на рис. 8.1).

1. Запрос может быть потерян (например, потому, что кто-то выдернул сетевой кабель).
2. Запрос может попасть в очередь и быть доставленным позднее (например, из-за перегруженности сети или получателя).

3. Может произойти отказ удаленного узла (например, из-за фатального сбоя или отключения электропитания).
4. Удаленный узел может временно перестать отвечать на запросы (например, из-за длительной паузы на сборку мусора; см. подраздел «Паузы при выполнении процессов» раздела 8.3), но продолжит отвечать позднее.
5. Удаленный узел, возможно, обработал запрос, но ответ был потерян при передаче по сети (например, из-за неправильной настройки сетевого коммутатора).
6. Удаленный узел, вероятно, обработал запрос, но ответ задержался и будет доставлен позднее (например, из-за перегруженности сети или вашей собственной машины).

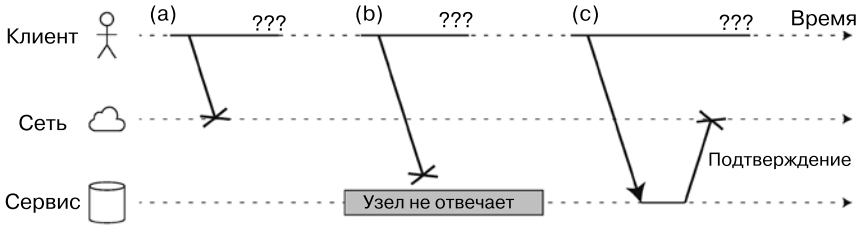


Рис. 8.1. В случае отправки запроса и неполучения ответа невозможно определить: *а)* был ли потерян запрос; *б)* работает ли удаленный узел; *в)* не был ли потерян ответ

Отправитель даже не знает, был ли доставлен пакет: для него единственный вариант узнать — отправляемое получателем ответное сообщение, которое, в свою очередь, может тоже потеряться или задержаться. Эти проблемы неразличимы в асинхронной сети: вы знаете только, что до сих пор не получили ответа. Сказать, почему вы не получили ответа на отправленный другому узлу запрос, *невозможно*.

Обычный способ решения этой проблемы — установить определенное *время ожидания*, по истечении которого вы перестаете ждать и считаете, что ответ вообще не будет получен. Однако в случае его превышения по-прежнему остается неизвестным, получил ли удаленный узел запрос (и если запрос все еще находится где-то в очереди, то он может быть доставлен получателю, несмотря на то что отправитель уже потерял надежду).

Сетевые сбои на практике

Компьютерные сети создаются уже десятилетиями, и, казалось бы, можно надеяться, что уже известно, как делать их надежными. Однако похоже, мы в этом еще не совсем преуспели.

Существуют как системные исследования, так и множество разрозненных фактов, показывающих, что сетевые проблемы встречаются поразительно часто, даже в контролируемых условиях находящегося в ведении одной компании ЦОДа [14].

В одном исследовании, проведенном в средних размеров ЦОДе, было обнаружено около 12 сетевых сбоев в месяц, из которых половина привела к отсоединению от сети одной машины, а половина — целой стойки [15]. В другом исследовании проводилась оценка частоты отказов таких компонентов, как стоечные коммутаторы верхнего уровня, агрегирующие коммутаторы и балансировщики нагрузки [16]. Оказалось, что добавление избыточного сетевого оборудования не снижает частоту отказов настолько, насколько бы хотелось, поскольку не защищает от человеческого фактора (например, ошибок неправильной настройки коммутаторов) — главной причины перебоев в обслуживании.

Общедоступные облачные сервисы, например EC2, печально известны частыми кратковременными сбоями [14], так что хорошо управляемые сети частных ЦОДов зачастую работают стабильнее. Тем не менее никто не застрахован от сетевых проблем: например, проблема, возникшая во время обновления программного обеспечения коммутатора, может вызвать перестройку топологии сети с задержкой доставки сетевых пакетов более чем на минуту [17]. Акула способна укусить подводный кабель и повредить его [18]. Среди других неожиданных сбоев сетевой интерфейс, периодически теряющий все входящие пакеты, но успешно отправляющий исходящие [19], — то, что сетевое подключение работает в одном направлении, не гарантирует его работу в противоположном.



Нарушение связности сети

Ситуация, в которой часть сети оказывается отрезанной от остальной вследствие сетевого сбоя, иногда называется нарушением связности сети (network partition) или фрагментацией сети (netsplit). В этой книге я буду обычно использовать более общий термин «сетевой сбой» (network fault) во избежание путаницы с секциями (partitions, shards) в системах хранения, обсуждавшимися в главе 6.

Даже если в вашей среде сетевые сбои случаются редко, сам факт *возможности* их возникновения означает, что приложения должны уметь с ними справляться. Любой сетевой обмен сообщениями подвержен сбоям — ничего с этим не поделаешь.

Если обработка ошибок для сетевых сбоев отсутствует или не протестирована должным образом, то порой могут случаться достаточно неприятные вещи: например, возникнут взаимные блокировки в кластере, вследствие чего он перестанет обслуживать запросы даже после восстановления работы сети [20] или вообще удалит все ваши данные [21]. Программное обеспечение, столкнувшееся с непредусмотренной разработчиком ситуацией, способно на самые неожиданные действия.

Обработка сетевых сбоев не означает, что с ними нужно *справляться молча*: если сеть обычно достаточно надежна, то разумно будет просто отображать пользовате-

лям сообщение об ошибке во время проблем с ней. Однако следует четко понимать, как ваше программное обеспечение реагирует на сетевые проблемы, и убедиться, что система сумеет восстановиться после них. Имеет смысл умышленно спровоцировать возникновение сетевых проблем и проверить реакцию системы (эта идея лежит в основе утилиты Chaos Monkey; см. раздел 1.2).

Обнаружение сбоев

Во многих системах необходимо автоматическое обнаружение сбойных узлов. Например:

- балансировщик нагрузки должен прекращать отправлять запросы узлам, которые не отвечают (то есть выводить их *из эксплуатации*);
- в распределенной базе данных, где используется репликация с одним ведущим узлом, в случае его сбоя нужно «повысить в ранге» один из ведомых узлов до нового ведущего (см. подраздел «Перебои в обслуживании узлов» раздела 5.1).

К сожалению, из-за вносимой сетью неопределенности сложно выяснить, работает ли узел. При некоторых особых обстоятельствах можно получить обратную связь, явным образом указывающую на сбой.

- Если машина, на которой работает узел, доступна, но ни один процесс не прослушивает целевой порт (например, вследствие фатального сбоя процесса), то операционная система будет — что весьма удачно — закрывать TCP-соединения или отказывать их устанавливать путем отправки в ответ пакета `RST` или `FIN`. Однако в случае сбоя удаленного узла во время обработки запроса никакого способа узнать, сколько данных было фактически им обработано, нет [22].
- Если удаленный узел потерпел сбой (или был уничтожен администратором), но операционная система узла по-прежнему функционирует, то можно использовать сценарий для оповещения других узлов о фатальном сбое, чтобы быстро делегировать выполнение другому узлу и не ждать истечения времени ожидания. Например, подобным образом работает HBase [23].
- При наличии доступа к административному интерфейсу сетевых коммутаторов ЦОДа можно их опросить на предмет отказов сетевых подключений на аппаратном уровне (например, отключения удаленной машины). Эта возможность исключается при соединении по Интернету, или в совместно используемом ЦОДе, где у вас нет доступа к самим коммутаторам, или при невозможности применять административный интерфейс из-за проблем с сетью.
- Если маршрутизатор уверен в недоступности нужного вам IP-адреса, то может сообщить об этом с помощью ICMP пакета Destination Unreachable («Адресат недоступен»). Однако у маршрутизатора нет «волшебного» способа обнаружения отказов — он ограничен в своих возможностях, как и другие участники работы сети.

Быстрое получение отклика о нерабочем состоянии удаленного узла полезно, но рассчитывать на него не стоит. Даже если TCP подтверждает доставку пакета, приложение могло отказать до завершения его обработки. Чтобы убедиться в успешном выполнении запроса, необходима позитивная реакция самого приложения [24].

С другой стороны, в случае возникновения неких проблем вы можете получить отклик об ошибке на каком-либо уровне стека, но в целом приходится допускать, что никакого отклика не будет. Можно попробовать повторить отправку несколько раз (протокол TCP делает это прозрачным для приложения образом, но стоит попробовать сделать это и на уровне приложения), подождать истечения времени ожидания и в случае отсутствия отклика объявить, что узел не функционирует.

Время ожидания и неограниченные задержки

Если ограничение времени ожидания — единственный надежный способ обнаружения сбоев, то насколько коротким оно должно быть? К сожалению, простого ответа на этот вопрос не существует.

Длительное время ожидания означает, что нужно долго ждать, прежде чем объявить узел вышедшим из строя (и все это время пользователи должны ждать или видеть сообщение об ошибке). Более короткое время ожидания позволяет быстрее обнаруживать сбои, но и влечет более высокий риск ошибочного объявления узла вышедшим из строя, в то время как он просто временно замедлил темп работы (например, вследствие пика нагрузки в узле или в сети).

Преждевременное объявление узла неработающим способно потенциально привести к проблемам: если он на самом деле работает и как раз выполняет какое-либо задание (например, отправляет сообщение электронной почты) и это задание будет передано другому узлу, то оно может оказаться выполненным дважды. Мы обсудим этот вопрос подробнее в разделе 8.4, а также в главах 9 и 11.

При объявлении узла нефункционирующим необходимо делегировать его обязанности другим узлам, что увеличивает нагрузку на них и на сеть. Если нагрузка на систему уже и так была высока, преждевременное объявление узлов неработающими только ухудшит ситуацию. В частности, может оказаться, что узел на самом деле работал, но медленно отвечал вследствие перегруженности. Передача его обязанностей другим узлам в подобном случае способна привести к каскадному сбою (в наихудшем случае все узлы объявят друг друга неработающими и вся система перестанет функционировать).

Представьте воображаемую систему с сетью, обеспечивающей гарантированную максимальную длительность задержки пакетов, — каждый пакет или доставляется за некоторое время d или теряется, но доставка никогда не занимает дольше d. Более того, допустим, мы можем гарантировать обработку запроса любым несбойным

узлом за некое время r. В этом случае гарантируем: на любой успешный запрос будет возвращен ответ за время $2d + r$, вследствие чего при его отсутствии за это время можно быть уверенными, что или узел, или сеть не работают. В подобной ситуации было бы разумно использовать $2d + r$ в качестве времени ожидания.

К сожалению, большинство систем, с которыми мы сталкиваемся, не могут гарантировать ни того ни другого: у асинхронных сетей *задержки не ограничены* (то есть они пытаются доставить пакеты как можно быстрее, но верхнего предела времени доставки пакета нет), а большинство реализаций серверов не гарантируют обработку запросов за определенное время (см. пункт «Гарантии времени отклика» подраздела «Паузы при выполнении процессов» раздела 8.3). Для обнаружения отказов недостаточно, чтобы система работала быстро большую часть времени: при коротком времени ожидания для вывода системы из строя достаточно лишь временного пика длительностей циклов отправки — отклика.

Перегруженность сети и очереди. При управлении автомобилем время путешествия по дорожной сети часто варьируется именно из-за перегруженности трафика. Аналогично изменение длительности задержек пакетов в компьютерных сетях чаще всего происходит из-за очередей [25].

- Если несколько различных узлов одновременно пытаются отправить пакеты одному получателю, то сетевому коммутатору придется выстроить их в очередь и отправлять на целевое сетевое подключение по одному (как показано на рис. 8.2). При занятом сетевом подключении пакету может понадобиться подождать некоторое время (это называется *перегруженностью сети*). При поступлении данных в количестве, вызывающем переполнение очереди, пакет удаляется и его приходится отправлять заново, несмотря на нормальную работу сети.
- Если на момент достижения пакетом машины получателя все ядра CPU оказываются заняты, то операционная система отправляет входящий сетевой запрос в очередь до тех пор, пока приложение не сможет его обработать. В зависимости от загруженности машины на это уйдет произвольное количество времени.
- В виртуализированных средах операционная система часто приостанавливает работу на несколько десятков миллисекунд, пока другая виртуальная машина использует ядро CPU. На протяжении этого отрезка времени виртуальная машина не может принимать из сети никакие данные, так что входящие данные накапливаются в очереди (буферизируются) гипервизором виртуальных машин [26], что делает длительность сетевых задержек еще более непредсказуемой.
- TCP осуществляет управление потоком данных (называемое также *предотвращением перегруженности* (congestion avoidance) или *контролем обратного потока* (backpressure)), при котором узел ограничивает свою частоту отправки сообщений во избежание перегрузки сетевого подключения или узла-получателя [27]. Это требует организации дополнительной очереди в узле-отправителе, еще до попадания данных в сеть.

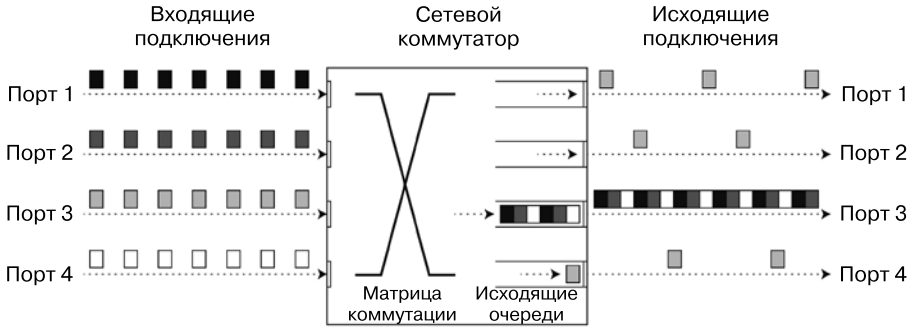


Рис. 8.2. В случае отправки несколькими машинами сетевого трафика одному получателю может произойти переполнение очереди его коммутатора. В данном случае порты 1, 2 и 4 пытаются отправить пакеты на порт 3

Более того, TCP считает пакет потерянным после отсутствия ответа по истечении некоторого времени ожидания (рассчитываемого на основе наблюдаемой длительности циклов отправки — отклика) и автоматически повторяет пересылку потерянных пакетов. Хотя приложению эта потеря и повторная отправка пакета незаметны, зато хорошо заметна происходящая из-за них задержка (ожидание, пока истечет время, после чего ожидание подтверждения получения повторно отправленного пакета).

TCP и UDP

Некоторые чувствительные к задержке программы, например приложения для видеоконференций и IP-телефонии (Voice over IP, VoIP) используют UDP вместо TCP. Это своеобразный компромисс между надежностью и непостоянством задержек: за счет отсутствия в UDP управления потоком данных и повторной отправки потерянных пакетов в этом протоколе удается избежать части причин для широкой вариабельности времени задержки (хотя в нем все равно возможны очереди на коммутаторах и задержки из-за диспетчеризации).

UDP имеет смысл выбирать в случаях, когда запоздавшие данные теряют всякую ценность. Например, при звонке по VoIP нет времени на повторную передачу пакета, прежде чем его данные должны быть воспроизведены через динамики. В таком случае смысла повторять отправку пакета нет — вместо этого приложению следует заполнить соответствующий пакету промежуток времени тишиной (что приведет к краткому прерыванию звука) и продолжить воспроизведение потока данных. Повтор происходит на уровне разговаривающих людей («Не могли бы вы повторить, что вы сказали? Звук только что прервался на секунду».)

Все эти факторы приводят к сильной вариабельности времени сетевых задержек. Задержки из-за очередей варьируются в особенно широком диапазоне в случаях, когда система работает на пределе возможностей: система, у которой еще есть

солидный резерв вычислительных мощностей, с легкостью способна исчерпать очереди, в то время как в сильно загруженной системе длинные очереди могут образовываться очень быстро.

В общедоступных облаках и мультиарендных ЦОДах ресурсы разделяются между множеством потребителей: сетевые подключения и коммутаторы и даже сетевые интерфейсы машин и процессоры (при работе на виртуальных машинах) применяются совместно. Пакетные источники нагрузки, например MapReduce (см. главу 10), с легкостью способны перегружать сетевые подключения. А поскольку вы не можете никак контролировать или знать, как другие пользователи задействуют разделяемые ресурсы, сетевые задержки могут меняться в очень широких пределах, если кто-то рядом с вами *(шумный сосед)* использует много ресурсов [28, 29].

В подобных средах время ожидания подбирается только экспериментальным образом: оценивать распределение длительности циклов отправки — отклика за большой промежуток времени и на множестве машин, чтобы определить ожидаемый разброс задержек. После этого можно найти — с учетом характеристик вашего приложения — подходящий компромисс между задержкой обнаружения отказа и риском преждевременного объявления узлов неработающими.

Или даже лучше: вместо того чтобы задействовать заранее заданную длительность времени ожидания, система может непрерывно измерять время отклика и его разброс (*фазовое дрожание* (jitter)) и автоматически подстраивать время ожидания к наблюдаемому распределению времени отклика. Для этого подойдет средство обнаружения отказов Phi Accrual [30], используемое в качестве примера в Akka и Cassandra [31]. Время ожидания для повторной отправки в протоколе TCP тоже определяется схожим образом [27].

Асинхронные и синхронные сети

Распределенные системы были бы намного проще, если бы можно было гарантировать доставку пакетов по сети в пределах фиксированного времени задержки без потери пакетов. Почему же нельзя решить данную проблему на аппаратном уровне и сделать сети надежными, чтобы программному обеспечению не приходилось об этом заботиться?

Чтобы ответить на этот вопрос, будет интересно сравнить сеть ЦОДа с обычной кабельной телефонной сетью (не сотовой и не VoIP), характеризующейся исключительной надежностью: задержки аудиокадров и обрывы вызовов случаются чрезвычайно редко. Телефонные звонки требуют постоянно низкого сквозного времени ожидания и достаточной для передачи голосовых аудиоданных ширины канала. Разве не замечательно было бы иметь подобный уровень надежности и предсказуемости в компьютерных сетях?

При звонке по телефонной сети формируется *канал вызова* (call circuit): фиксированная гарантированная часть полосы пропускания, выделенная для данного звонка, на протяжении всего маршрута между собеседниками. Этот канал сохраняется вплоть до окончания вызова [32]. Например, сеть ISDN работает с фиксированной скоростью 4000 кадров/с. При организации звонка внутри каждого кадра выделяется 16 бит (в каждом направлении). Следовательно, на протяжении всего звонка каждая из сторон гарантированно может отправить ровно 16 бит аудиоданных каждые 250 микросекунд [33, 34].

Подобные сети *синхронны*: данные, даже проходя через несколько маршрутизаторов, не попадают в очереди, поскольку 16 бит выделенного канала для звонка заранее зарезервировано на следующем транзитном участке сети. А поскольку очередей нет, максимальное сквозное время задержки сети фиксировано. Это называется *ограниченной задержкой* (bounded delay).

Нельзя ли просто сделать сетевые задержки предсказуемыми? Обратите внимание: канал в телефонных сетях сильно отличается от TCP-соединения. Канал представляет собой зарезервированный фиксированный объем полосы пропускания, недоступный для использования другими звонками в это время, тогда как пакеты TCP соединения задействует при возможности доступную полосу пропускания сети. TCP при получении блока данных переменного размера (например, сообщения электронной почты или веб-страницы) будет стараться передать его за кратчайшее возможное время. Во время простоя TCP-соединения полоса пропускания не применяется[1].

Если бы сети ЦОДов и Интернет являлись сетями с коммутацией каналов, можно было бы гарантировать конкретную максимальную длительность циклов отправки — отклика при установке канала. Однако они не такие: Ethernet и IP — протоколы с коммутацией пакетов, чреватые возникновением очередей, а следовательно, и неограниченностью задержек в сети. В этих сетях нет понятия канала вызова.

Почему же в сетях ЦОДов и Интернете используется коммутация пакетов? А потому, что они оптимизированы для *трафика с периодической пиковой нагрузкой* (bursty traffic). Установка канала вызова хорошо подходит для аудио- и видеозвонков, при которых приходится передавать примерно одинаковое количество бит в секунду на протяжении всего звонка. С другой стороны, запрос веб-страницы, отправка сообщения электронной почты или передача файла не выдвигает конкретных требований к ширине полосы пропускания — необходимо лишь передать их настолько быстро, насколько возможно.

Для передачи файла при установленном канале вызова пришлось бы предугадывать, какая ширина полосы пропускания понадобится. В случае заниженного значения передача будет выполняться слишком медленно, а возможности сети — использоваться не полностью. В случае завышенного — канал не удастся создать

[1] За исключением периодических keepalive-пакетов в случае включения соответствующего режима TCP.

(поскольку сеть не позволит создать его, если нет возможности гарантировать выделение нужной полосы пропускания). Следовательно, применение каналов для трафика с периодической пиковой нагрузкой лишь приводит к пустой трате ресурсов сети и замедляет передачу данных. Напротив, протокол TCP динамически подстраивает скорость передачи данных к имеющимся ресурсам сети.

Были предприняты попытки создания гибридных сетей, поддерживающих коммутацию как каналов, так и пакетов, например ATM[1]. Несколько схож с ними InfiniBand [35], в котором реализовано сквозное управление потоком на уровне сетевых подключений, снижающее необходимость организации в сети очередей, хотя перегруженность все равно оказывает на него свое влияние [36]. С помощью рационального использования *QoS*[2] (назначения приоритетов и планирования пакетов) и *управления допуском* (admission control, ограничения скорости передачи) можно эмулировать коммутацию пакетов в сетях с коммутацией пакетов или обеспечить статистически ограниченные задержки [25, 32].

Время ожидания и использование ресурсов

Вообще говоря, можно рассматривать переменные задержки как последствия динамического разделения ресурсов.

Допустим, два телефонных коммутатора соединены кабелем с пропускной способностью, достаточной для 10 000 одновременных звонков. Каждый переключаемый на этот кабель канал занимает один из слотов звонков. Следовательно, кабель можно рассматривать как ресурс, который совместно задействует вплоть до 10 000 пользователей. Этот ресурс разбивается *статически*: даже если по кабелю в настоящий момент осуществляется только один звонок, а все 9999 остальных слотов не применяются, канал все равно занимает ту же самую ширину полосы пропускания, что и при полном использовании ресурса кабеля.

Напротив, Интернет разделяет полосу пропускания сети *динамически*. Отправители борются друг с другом за возможность максимально быстро передать свои пакеты по кабелю, а сетевые коммутаторы выбирают, какой из них отправить (то есть выделяют ресурсы сети). Недостатком такого подхода является образование очередей, а достоинством — применение кабеля по максимуму. Стоимость использования кабеля фиксирована, так что чем полнее вы его задействуете, тем дешевле обходится отправка каждого байта.

---

[1] Асинхронный режим передачи данных (asynchronous transfer mode, ATM) — конкурент Ethernet в 1980-х годах, который не приобрел особой популярности, если не считать центральных коммутаторов телефонных станций. Несмотря на такую же аббревиатуру, не имеет ничего общего с банкоматами (automatic teller machines). Возможно, что в какой-нибудь параллельной вселенной Интернет создан на основе чего-то вроде ATM, — в ней видеозвонки через Интернет, вероятно, гораздо более надежны, чем в нашей, поскольку не страдают от потери и задержек пакетов.

[2] От англ. quality of service — «качество обслуживания».

Ситуация с CPU схожа: если каждое ядро CPU используется совместно несколькими потоками выполнения, каждому из них периодически приходится ждать в очереди выполнения операционной системы во время работы других потоков, так что его выполнение может приостанавливаться на различные промежутки времени. Однако при этом аппаратное обеспечение задействуется более полно, чем при выделении каждому потоку фиксированного количества циклов CPU (см. пункт «Гарантии времени отклика» подраздела «Паузы при выполнении процессов» раздела 8.3). Одной из причин для применения виртуальных машин является в том числе и более полное использование аппаратного обеспечения.

В некоторых средах при статическом разделении ресурсов (например, выделенном оборудовании и монопольном выделении полосы пропускания) реально дать гарантии времени задержки. Однако это происходит за счет снижения полноты использования — иными словами, более затратно. С другой стороны, мультиарендность с динамическим разделением ресурсов обеспечивает большую полноту применения, но за счет более широкого варьирования задержек.

Широкое варьирование задержек в сетях отнюдь не закон природы, оно возникает в результате компромисса между затратами и получаемыми результатами.

Однако возможности QoS в настоящий момент не предоставляются в мультиарендных ЦОДах и общедоступных облаках и при взаимодействии через Интернет[1]. Используемая в настоящий момент технология не позволяет гарантировать величину задержек или надежность сети: приходится допускать вероятность перегруженности сети, неограниченных задержек и возникновения очередей. В результате не существует «правильного» значения длительности времени ожидания — его приходится определять экспериментально.

8.3. Ненадежные часы

Часы и измерение времени чрезвычайно важны. Приложениям требуются часы, чтобы отвечать на следующие вопросы.

1. Истекло ли время ожидания запроса?
2. Чему равен 99-й процентиль времени отклика данного сервиса?
3. Сколько запросов в секунду в среднем обрабатывал данный сервис за последние пять минут?
4. Сколько времени пользователь провел на сайте?

[1] Соглашения между интернет-провайдерами об обмене трафиком и формирование маршрутов с помощью протокола пограничной маршрутизации (border gateway protocol, BGP) сильнее напоминает коммутацию каналов, чем сам IP. На этом уровне имеется возможность приобретать выделенную полосу пропускания. Однако интернет-маршрутизация работает на уровне сетей, а не отдельных соединений между узлами, причем в совсем других временных рамках.

5. Когда была опубликована статья?
6. Когда (дата, время) нужно отправить сообщение электронной почты с напоминанием?
7. Когда истекает срок хранения записи кэша?
8. Какова метка даты/времени сообщения об ошибке в журнале?

В примерах 1–4 речь идет о *длительности* (то есть интервалах времени между отправкой запроса и получением ответа), а примеры 5–8 описывают *моменты времени* (события, произошедшие в конкретные день и время).

В распределенной системе время — дело тонкое, поскольку взаимодействие происходит не мгновенно: для передачи сообщения по сети от одной машины к другой требуется время. Сообщение доставляется всегда позже, чем было отправлено, но вследствие меняющихся задержек в сети мы не знаем, насколько именно. Этот факт иногда затрудняет определение конкретной последовательности событий в случае нескольких машин.

Более того, у каждой машины в сети свои часы, представляющие собой аппаратное устройство: обычно генератор с кварцевой стабилизацией частоты. Точность этих устройств оставляет желать лучшего, так что у каждой машины собственное представление о времени, вероятно отстающем или опережающем время других машин. Часы можно синхронизировать до некоторой степени: самый распространенный механизм — сетевой протокол времени (network time protocol, NTP), позволяющий компьютерам подстраивать свои часы в соответствии с временем, передаваемым группой серверов [37]. Эти серверы, в свою очередь, получают время от еще более точного источника, например от GPS-приемника.

Монотонные часы и часы истинного времени

В современных компьютерах существует как минимум две разновидности часов: *часы истинного времени* (time-of-day clock) и *монотонные часы* (monotonic clock). Хотя и те и другие измеряют время, необходимо их различать, поскольку они служат разным целям.

Часы истинного времени

Часы истинного времени делают именно то, что и следует ожидать от часов: возвращают текущие дату и время в соответствии с каким-либо календарем (так называемое *физическое время*). Например, функция `clock_gettime(CLOCK_REALTIME)` в операционной системе Linux[1] и `System.currentTimeMillis()` в языке программирования

[1] Хотя их называют часами реального времени, они не имеют никакого отношения к операционным системам реального времени (см. пункт «Гарантии времени отклика» подраздела «Паузы при выполнении процессов» данного раздела).

Java возвращают количество секунд (или миллисекунд), прошедших с начала *эры UNIX* — полуночи 1 января 1970 года по UTC в соответствии с григорианским календарем, не считая секунд координации. В некоторых системах используются в качестве точки отсчета другие даты.

Часы истинного времени обычно синхронизированы с помощью NTP, вследствие чего метка даты/времени с одной машины (идеально) соответствует такой же метке на другой. Однако у часов истинного времени есть некоторые странности, как описано ниже. В частности, если локальное время слишком сильно отличается от времени NTP-сервера, то локальные часы могут быть принудительно сброшены и как будто перепрыгнут на предшествующий момент времени. Эти прыжки, а также игнорирование секунд координации делают часы истинного времени неподходящими для измерения прошедшего промежутка времени [38].

Разрешение часов истинного времени, так исторически сложилось, довольно грубое, например, в старых Windows-системах они идут интервалами 10 мс [39]. В современных системах этой проблемы практически нет.

Монотонные часы

Монотонные часы подходят для измерения продолжительности интервалов времени, таких как время ожидания или отклика сервиса: например, функции `clock_gettime(CLOCK_MONOTONIC)` в операционной системе Linux и `System.nanoTime()` в языке программирования Java представляют собой монотонные часы. Они получили свое название потому, что гарантированно движутся вперед, в отличие от часов истинного времени, которые могут перепрыгивать назад во времени.

Можно посмотреть показания монотонных часов в какой-то момент времени, выполнить какие-либо действия, а затем посмотреть их показания снова. *Разница* между двумя полученными значениями соответствует времени, прошедшему между двумя проверками. Однако *абсолютные* показания монотонных часов смысла не имеют: это может быть количество наносекунд с момента загрузки компьютера или что-то столь же случайное. В частности, не имеет смысла сравнивать показания монотонных часов с двух разных компьютеров, поскольку их значения различаются.

На сервере с несколькими CPU у каждого процессора может быть отдельный таймер, не обязательно синхронизированный с другими процессорами. Операционная система сглаживает все несоответствия и обеспечивает по возможности монотонное представление часов потокам выполнения приложения, даже выполняемых различными CPU. Однако не помешает относиться к этим гарантиям монотонности немного скептически [40].

NTP умеет подстраивать частоту хода монотонных часов (это называется *подкручиванием* (slewing) часов), если обнаруживает, что локальные часы компьютера

отстают или спешат по сравнению с NTP-сервером. По умолчанию NTP допускает отставание или ускорение часов не более чем на 0,05 %, но NTP не может заставить монотонные часы перескочить вперед или назад. Их разрешение обычно достаточно хорошее: в большинстве систем такие часы способны отмерять микросекундные интервалы или даже еще более мелкие.

Использование монотонных часов для измерения прошедшего времени в распределенной системе (например, времени ожидания) обычно допустимо, поскольку не предполагает какой-либо синхронизации между часами различных узлов и они нечувствительны к небольшим неточностям измерения.

Синхронизация часов и их точность

Монотонным часам не требуется синхронизация, но часы истинного времени, чтобы нести какую-то пользу, должны устанавливаться в соответствии с NTP-сервером или каким-либо другим внешним источником времени. К сожалению, имеющиеся у нас методы получения точного времени далеко не столь надежны или точны, как хотелось бы: аппаратные часы и NTP могут быть вещами весьма ненадежными. Вот несколько примеров.

- Кварцевые часы компьютера не слишком точны: они *расходятся* с правильным временем (спешат или отстают). Расхождение часов (clock drift) варьируется в зависимости от температуры машины. Google допускает для своих серверов расхождение часов на 200 миллионных долей [41], что эквивалентно 6 миллисекундам для часов, синхронизируемых с сервером каждые 30 секунд, или 17 секундам — для синхронизируемых ежедневно. Такое расхождение ограничивает максимально достижимую точность даже в случае правильного функционирования системы.
- Если часы компьютера слишком сильно расходятся с NTP-сервером, то он может отказать в синхронизации или принудительно сбросить значение локальных часов [37]. Все приложения, наблюдавшие по ним время до и после такого сброса, обнаружат внезапный сдвиг времени назад или вперед.
- Ошибка конфигурации, при которой узел случайно оказывается отрезанным брандмауэром от NTP-серверов, может остаться незамеченной в течение некоторого времени. Бытует мнение, что это случалось на практике.
- Точность NTP-синхронизации ограничена сетевой задержкой, что особенно чувствуется в перегруженной сети с переменной задержкой доставки пакетов. В соответствии с одним экспериментом можно достичь минимальной ошибки 35 миллисекунд при синхронизации по Интернету [42], хотя периодические пики сетевых задержек приводят к ошибке около секунды. В зависимости от настроек клиент может вообще отказаться от дальнейших попыток синхронизации при длительной сетевой задержке.

- Некоторые NTP-серверы неточны или неправильно сконфигурированы и возвращают время, отличающееся от точного, на часы [43, 44]. NTP-клиенты опрашивают несколько серверов и игнорируют аномальные значения, так что они достаточно ошибкоустойчивы. Тем не менее не хотелось бы основывать точное время своей системы на данных, получаемых от неизвестно кого в Интернете.
- Секунды координации приводят к минутам длиной 59 или 61 секунда, нарушающим все допущения о хронометраже в тех системах, при проектировании которых они не были учтены [45]. Фатальные отказы множества крупных систем вследствие появления секунд координации [38, 46] показывают, насколько легко неправильное время становится системным. Оптимальный способ обработки секунд координации — сделать так, чтобы NTP-серверы «врали» и вносили поправку на секунду координации по частям, в течение дня (так называемое *размазывание*) [47, 48], хотя на практике NTP-серверы ведут себя по-разному [49].
- В виртуальных машинах аппаратные часы тоже виртуализируются, что создает дополнительные трудности для требующих точного учета времени приложений [50]. При разделении ядра CPU между несколькими виртуальными машинами каждая из них приостанавливается на несколько десятков миллисекунд, пока работают другие. С точки зрения приложения это выглядит как неожиданный прыжок часов вперед во времени [26].
- Если ваше программное обеспечение работает на устройствах, находящихся вне вашего полного контроля (например, мобильных или встроенных), то вы, вероятно, вообще не можете доверять их аппаратным часам. Некоторые пользователи умышленно устанавливают свои аппаратные часы на неправильные дату и время, например, чтобы обойти ограничения хронометража в играх. В результате часы могут отставать или спешить на произвольное время.

Если время играет достаточно большую роль в том, чтобы вложить в обеспечение точности часов значительные средства, то можно добиться неплохих результатов. Например, проект регламента Евросоюза для финансовых учреждений MiFID II требует, чтобы все участники высокочастотного трейдинга синхронизировали свои часы в пределах 100 мс по UTC ради выявления и устранения аномалий рынка, таких как «мгновенные обвалы», и обнаружения рыночных злоупотреблений [51].

Подобной точности можно добиться с помощью GPS-приемников, протокола точного времени (precision time protocol, PTP) [52] и тщательного развертывания и мониторинга. Однако это требует дополнительных усилий и знаний, и при синхронизации часов вероятна масса проблем. Расхождение часов способно быстро вырасти при неправильной конфигурации NTP-демона или блокировке NTP-трафика брандмауэром.

Ненадежность синхронизированных часов

Проблема с часами состоит в том, что при кажущихся простоте и удобстве использования они насчитывают немало подводных камней: день может состоять не точно из 86 400 секунд, время одного узла — сильно отличаться от времени другого, а часы истинного времени могут перескакивать назад во времени.

Ранее в этой главе мы обсуждали потерю и произвольную задержку пакетов сетями. И хотя большую часть времени сети работают достаточно надежно, необходимо проектировать программное обеспечение с учетом возможности их периодических сбоев, требующих должной обработки. То же самое справедливо и для часов: хотя большую часть времени они работают достаточно хорошо, ошибкоустойчивому ПО следует быть готовым к работе с неточными часами.

Часть проблемы состоит в том, что неточность часов легко может остаться незамеченной. Неисправность CPU машины или неправильная конфигурация сети часто приводят к полному прекращению работы, которое сразу же будет замечено и исправлено. С другой стороны, если кварцевые часы не работают или NTP-клиент настроен неправильно, то практически все выглядит нормально функционирующим, хотя часы все дальше и дальше расходятся с реальностью. При зависимости какой-то из частей программного обеспечения от точной синхронизации часов результатом этого станет, скорее всего, не эффектный отказ системы, а скрытая и малозаметная потеря данных [53, 54].

Следовательно, при использовании требующего синхронизированных часов ПО необходимо тщательно следить за расхождениями показаний часов на всех машинах. Любой узел, часы которого расходятся слишком сильно с часами остальных узлов, следует объявить неработающим и исключить из кластера. Подобный мониторинг гарантирует обнаружение испорченных часов до того, как они нанесут слишком большой ущерб.

Метки даты/времени и упорядочение событий

Рассмотрим конкретную ситуацию, в которой выглядит очень заманчивой, хотя и рискованной, надежда на точность часов, — упорядочение событий в нескольких узлах. Например, кто из двух клиентов, записывающих данные в распределенную базу, окажется первым? Чьи данные свежее?

На рис. 8.3 проиллюстрировано рискованное использование часов истинного времени в базе данных с репликацией с несколькими ведущими узлами (пример схож с показанным на рис. 5.9). Клиент A записывает $x = 1$ в узле 1, операция записи реплицируется в узел 3; клиент B увеличивает на единицу значение x в узле 3 (там теперь $x = 2$); и наконец, обе операции записи реплицируются в узел 2.

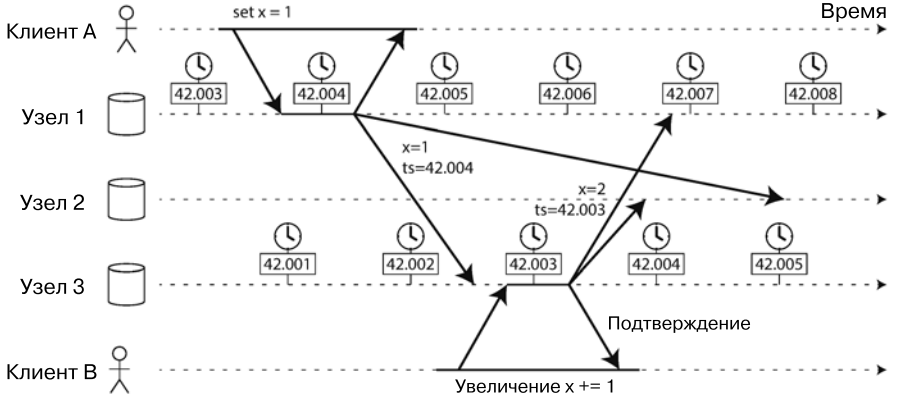


Рис. 8.3. Операция записи, выполняемая клиентом B, причинно-следственно зависит от операции записи клиента A, но метка даты/времени первой — более ранняя

При репликации операции записи в другие узлы на рис. 8.3 она помечается меткой даты/времени в соответствии с часами истинного времени на инициировавшем ее узле. Синхронизация часов в этом примере весьма точна: расхождение между узлами 1 и 3 не превышает 3 миллисекунд — на практике, вероятно, будет хуже.

Тем не менее на рис. 8.3 не получается упорядочить должным образом события с помощью меток даты/времени: эта метка операции записи $x = 1$ равна 42,004 секунды, а операции записи $x = 2$ — 42,003, несмотря на однозначно более позднее выполнение записи $x = 2$. При получении этих двух событий узел 2 ошибочно сочтет, что значение $x = 1$ — более позднее, и отбросит операцию записи $x = 2$. В результате операция увеличения значения x, выполненная клиентом B, будет потеряна.

Эта стратегия разрешения конфликтов называется *«выигрывает последний»* (last write wins, LWW) и широко используется в базах данных с репликацией как с несколькими ведущими узлами, так и без ведущего узла, например, Cassandra [53] и Riak [54] (см. пункт «“Выигрывает последний” (отбраковка конкурентных операций записи)» подраздела «Обнаружение конкурентных операций записи» раздела 5.4). В некоторых реализациях метки даты/времени генерируются на клиенте, а не на сервере, но это не решает фундаментальных проблем LWW.

- Операции записи в базу данных могут пропадать загадочным образом: узел с отстающими часами не способен перезаписывать значения, записанные ранее узлом со спешащими часами, пока не пройдет время, на которое расходятся часы этих узлов [54, 55]. При таком сценарии произвольные объемы данных могут отбрасываться без какого-либо уведомления приложения об ошибке.
- LWW не способна различать операции записи, выполняемые последовательно и быстро одна за другой (на рис. 8.3 увеличение значения клиентом B определенно происходит *после* операции записи клиента A), а также подлинно конкурентные операции записи (когда ни один из записывающих клиентов не знает

о другом). Чтобы предотвратить нарушение причинно-следственной связи, приходится использовать дополнительные механизмы отслеживания причинности, например векторы версий (см. подраздел «Обнаружение конкурентных операций записи» раздела 5.4).

- Два узла могут независимо генерировать операции записи с одинаковыми метками даты/времени, особенно когда разрешение часов определяется лишь миллисекундами. Для разрешения подобных конфликтов требуется специальное дополнительное значение (например, просто большое случайное число), но этот подход тоже способен привести к нарушению причинно-следственных связей [53].

Следовательно, хотя разрешение конфликтов с помощью сохранения «последнего» значения и отбрасывания остальных и кажется заманчивым, важно не забывать, что определение «последнего» значения зависит от локальных часов истинного времени, которые вполне могут идти неправильно. Даже при четко синхронизированных по протоколу NTP часах остается вероятность отправить пакет с меткой даты/времени 100 миллисекунд (по часам отправителя) и получить его в момент, соответствующий метке даты/времени 99 миллисекунд (по часам получателя), — как будто пакет прибыл до отправки, что невозможно.

Можно ли добиться точности NTP-синхронизации, достаточной для исключения подобного неправильного упорядочения? Вероятно, нет, поскольку она сама по себе ограничена длительностью циклов отправки-отклика сети, помимо других источников ошибок, например расхождения кварцевых часов. Для правильного упорядочения необходимо, чтобы источник времени был существенно точнее измеряемого значения (а именно, сетевой задержки).

Так называемые *логические часы* (logical clocks) [56, 57], в основе которых лежит наращивание счетчика, а не колебательный кварцевый генератор, — более надежная альтернатива для упорядочения событий (см. подраздел «Обнаружение конкурентных операций записи» раздела 5.4). Логические часы измеряют не время суток или количество прошедших секунд, а только относительную упорядоченность событий (одно произошло до другого или после). В отличие от них часы истинного времени и монотонные часы, измеряющие фактически прошедшее время, называются *физическими часами*. Мы рассмотрим упорядоченность подробнее в разделе 9.3.

Доверительный интервал показаний часов

Вероятно, вы способны снимать показания часов истинного времени с разрешением порядка микросекунд или даже наносекунд. Но и в подобном случае отнюдь не факт, что точность полученного значения столь же высока. На деле, вероятнее всего, это не так — как уже упоминалось выше, расхождение неточных кварцевых часов может легко достигать нескольких миллисекунд даже при синхронизации с NTP-сервером в локальной сети каждую минуту. Если NTP-сервер находится в общедоступном Интернете, то максимально возможная точность будет порядка

десятков миллисекунд, а ошибка легко способна достигать 100 миллисекунд в случае перегруженности сети [57].

Следовательно, не имеет смысла рассматривать показания часов как момент времени — это скорее промежуток времени в пределах доверительного интервала: например, система может быть на 95 % уверена, что текущее время находится между 10,3 и 10,5 секунд после минутной отметки, но ничего точнее сказать нельзя [58]. Если время известно только с точностью ±100 миллисекунд, то относящиеся к микросекундам цифры в метке даты/времени, по существу, не имеют смысла.

По источнику времени можно вычислить границы погрешности. Для подключенного непосредственно к вашему компьютеру GPS-приемника или атомных (цезиевых) часов ожидаемый диапазон погрешности указывается их производителем. При получении времени от сервера погрешность зависит от ожидаемого расхождения кварцевого генератора с момента последней синхронизации с сервером, плюс погрешность NTP-сервера, плюс длительность цикла отправки — отклика на сервер в сети (в первом приближении и при условии, что вы доверяете серверу).

К сожалению, большинство систем не сообщают величину этой погрешности: например, при вызове функции `clock_gettime()` в возвращаемом значении не указывается ожидаемая ошибка метки даты/времени, так что вам неизвестен ее доверительный интервал: 5 миллисекунд или 5 лет.

Интересным исключением является API *TrueTime* Google в СУБД Spanner [41], который явным образом сообщает доверительный интервал локальных часов. При запросе текущего времени вам возвращаются два значения: [*earliest*, *latest*], представляющие собой *самую раннюю* и *самую позднюю из возможных* меток даты/времени. Основываясь на расчетах погрешности, часы знают, что фактическое текущее время находится где-то внутри этого интервала. Ширина его зависит в том числе от времени, прошедшего с момента последней синхронизации кварцевых часов с более точным источником.

Синхронизация часов для глобальных снимков состояния

В подразделе «Изоляция снимков состояния и воспроизводимое чтение» раздела 7.2 мы обсуждали *изоляцию снимков состояния* — очень удобную возможность баз данных, необходимую для поддержки как маленьких, быстрых транзакций, предназначенных для чтения и записи, так и больших, длительных транзакций только для чтения (например, для целей создания резервных копий и аналитики). Благодаря ей транзакции только для чтения видят базу в согласованном на определенный момент времени состоянии, без блокировки и создания помех для работы транзакций, предназначенных для чтения и записи.

Наиболее распространенным реализациям изоляции снимков состояния необходим монотонно возрастающий идентификатор транзакций. Если операция записи была выполнена после снятия снимка состояния (то есть идентификатор транзакции

операции записи превышает идентификатор транзакции снимка состояния), то она «невидима» для транзакции снимка состояния. На одноузловых базах для генерации идентификаторов транзакций достаточно простого счетчика.

Однако в случае распределенной по множеству машин базы данных (вероятно, в нескольких ЦОДах) сгенерировать глобальный, монотонно возрастающий идентификатор транзакций (для всех секций) не так уж просто и требует согласования действий. Идентификатору транзакций следует отражать причинно-следственные связи: если транзакция B читает записанное транзакцией A значение, то идентификатор транзакции B должен быть больше идентификатора транзакции A — в противном случае снимок состояния не будет согласованным.

При большом количестве маленьких быстрых транзакций создание идентификаторов транзакций в распределенной системе становится необоснованным узким местом[1].

Можно ли использовать метки даты/времени синхронизированных часов истинного времени в качестве идентификаторов транзакций? Если удастся обеспечить достаточно хорошую синхронизацию, то их характеристики вполне подойдут: у более поздних транзакций будет больший идентификатор. Проблема, конечно, в погрешности часов.

СУБД Spanner реализует подобным образом изоляцию снимков состояния на нескольких ЦОДах [59, 60]. Он использует возвращаемый API TrueTime доверительный интервал часов и основывается на следующем наблюдении. При двух данных доверительных интервалах, каждый из которых состоит из самой ранней и самой поздней из возможных меток даты/времени ($A = [A_{min}, A_{max}]$ и $B = [B_{min}, B_{max}]$), причем эти два интервала не пересекаются (то есть $A_{min} < A_{max} < B_{min} < B_{max}$), событие B определенно произошло после события A — в этом никаких сомнений быть не может. Сомнения в очередности событий A и B могут возникнуть только при пересечении этих интервалов.

Чтобы метки даты/времени транзакций отражали причинно-следственные связи, СУБД Spanner умышленно выжидает время, соответствующее длительности доверительного интервала перед фиксацией транзакции для чтения и записи. Таким образом он гарантирует, что любая читающая эти данные транзакция происходит достаточно позже и, как следствие, их доверительные интервалы не пересекаются. В целях максимального сокращения времени такого ожидания Spanner

[1] Существуют распределенные генераторы последовательных чисел, например Snowflake в социальной сети Twitter, которые генерируют почти монотонно возрастающие уникальные идентификаторы масштабируемым образом (например, путем выделения блоков пространства для идентификаторов в различных узлах). Однако они обычно не способны обеспечить согласованную причинно-следственную упорядоченность, поскольку масштаб времени присваивания блоков идентификаторов больше, чем масштаб времени операций записи и чтения базы данных. См. также раздел 9.3.

приходится минимизировать погрешность часов; для этого Google устанавливает GPS-приемник или атомные часы в каждом ЦОДе, благодаря чему часы синхронизируются с точностью в пределах 7 миллисекунд [41].

Использование синхронизации часов для распределенных транзакций — сфера активных научных исследований [57, 61, 62]. Эти идеи представляют интерес, но пока еще не воплощены в основных БД, помимо баз данных компании Google.

Паузы при выполнении процессов

Рассмотрим еще один пример рискованного использования часов в распределенной системе. Допустим, у вас есть база данных с одним ведущим узлом для каждой секции. Принимать операции записи способен только ведущий узел. Откуда он знает, что все еще является ведущим (а не был объявлен остальными узлами неработающим) и может спокойно принимать операции записи?

Одна из возможностей для ведущего узла — получить от других узлов *договор аренды* (lease) — некое подобие блокировки с заданным временем ожидания [63]. Быть владельцем аренды одновременно может только один узел — следовательно, при получении договора узел знает, что стал ведущим на некоторое время, до истечения аренды. Чтобы продолжать оставаться ведущим, узел обязан периодически возобновлять договор перед истечением срока его действия. В случае отказа узел прекращает возобновлять договор аренды, так что по его истечении функции ведущего передаются другому узлу.

Цикл обработки запроса будет при этом выглядеть примерно следующим образом:

```
while (true) {
    request = getIncomingRequest();

    // Убедиться, что до истечения срока аренды остается не менее 10 секунд
    if (lease.expiryTimeMillis - System.currentTimeMillis() < 10000) {
        lease = lease.renew();
    }

    if (lease.isValid()) {
        process(request);
    }
}
```

Что не так с этим кодом? Во-первых, он зависит от синхронизации часов: время истечения договора аренды задается другой машиной (где он может рассчитываться как текущее время плюс 30 секунд, например), а сверяется с локальными системными часами. Если часы рассинхронизированы более чем на несколько секунд, то данный код начнет вытворять странные вещи.

Во-вторых, даже если мы поменяем протокол с целью использовать только локальные монотонные часы, есть другая проблема: этот код предполагает, что промежуток между моментом проверки времени (`System.currentTimeMillis()`) и мо-

ментом обработки запроса (`process(request)`) очень невелик. Обычно данный код выполняется очень быстро, поэтому 10-секундного буфера более чем достаточно для гарантии того, что договор аренды не истечет посередине обработки запроса.

Однако каково развитие событий в случае непредвиденной паузы в выполнении программы? Например, представьте, что поток выполнения был приостановлен в районе строки `lease.isValid()`. В этом случае вполне вероятно, что договор аренды истечет ко времени обработки запроса и ведущим будет уже другой узел. Однако ничто в коде не указывает потоку, что он был приостановлен так надолго, поэтому код заметит истечение срока договора аренды только в момент следующего прохода цикла. А к этому времени код уже вполне мог сделать что-то рискованное при обработке запроса.

Так ли безумно предположение о подобной длительной паузе потока выполнения? К сожалению, нет. Вот несколько возможных причин подобной паузы.

- В средах выполнения многих языков программирования (например, на виртуальной машине Java) есть *сборщик мусора* (garbage collector, GC), периодически приостанавливающий все работающие потоки. Эти *«всеобъемлющие» паузы GC* иногда длятся по нескольку минут [64]! Даже так называемые конкурентные сборщики мусора, например HotSpot CMS виртуальной машины Java, не могут работать полностью параллельно с кодом приложения — даже им приходится останавливать все время от времени [65]. Хотя длительность данных пауз зачастую можно снизить за счет изменения паттернов выделения ресурсов или настройки GC [66], необходимо предполагать худшее, чтобы гарантировать отказоустойчивость.
- В виртуализированных средах возможна *приостановка* (suspend) виртуальных машин (приостановка выполнения всех процессов и сохранение содержимого памяти на жесткий диск) с последующим *возобновлением* выполнения (resume) (восстановлением содержимого памяти и продолжением выполнения). Такая пауза может произойти в любой момент выполнения процесса и продолжаться произвольный период времени. Это свойство иногда используется для *динамической миграции* (live migration) виртуальных машин с одного хоста на другой без перезагрузки. В данном случае длительность паузы зависит от скорости записи процессов в память [67].
- На устройствах конечных пользователей, например ноутбуках, выполнение также может приостанавливаться и возобновляться в произвольные моменты времени, к примеру, при закрытии крышки ноутбука.
- При переключении контекста операционной системы на другой поток выполнения или переключении гипервизора на другую виртуальную машину (при работе в виртуальной машине) выполняемый в настоящий момент поток можно приостановить в любой произвольной точке кода. В случае виртуальной машины время процессора, потраченное на другие виртуальные машины, называется *украденным временем* (steal time). Если нагрузка машины высока (имеется длинная очередь ожидающих выполнения потоков), то может пройти немало времени, прежде чем приостановленные потоки снова будут запущены.

- Если приложение осуществляет синхронный доступ к диску, то можно приостановить поток, чтобы подождать завершения медленной операции дискового ввода/вывода [68]. Во многих языках программирования доступ к диску может производиться в неожиданные моменты времени, даже если код не упоминает его явным образом — например, загрузчик классов языка Java выполняет отложенную загрузку файлов классов при первом использовании, которое может произойти в любой момент выполнения программы. Паузы из-за ввода/вывода и сборки мусора способны даже накладываться друг на друга [69]. Если диск фактически представляет собой сетевую файловую систему или сетевое блочное устройство (например, EBS компании Amazon), то время ожидания ввода/вывода зависит еще и от сетевых задержек [29].
- Если настройки операционной системы допускают *свопинг на диск (постраничную подкачку)*, то простое обращение к оперативной памяти может привести к страничной ошибке, требующей загрузки страницы с диска в оперативную память. Поток приостанавливается на время выполнения данной медленной операции ввода/вывода. В случае сильного дефицита памяти это, в свою очередь, способно потребовать сброса еще одной страницы на диск. В экстремальных случаях операционная система может тратить большую часть своего времени на перемещение страниц в оперативную память и из нее, а не на выполнение фактической работы (это называется *пробуксовкой* (thrashing)). Чтобы избежать данной проблемы, на серверных машинах постраничная подкачка часто отключается (лучше прервать выполнение процесса для освобождения памяти, чем рисковать возникновением «пробуксовки»).
- В операционной системе Unix возможна приостановка выполнения процессов с помощью отправки им сигнала `SIGSTOP`, например, путем нажатия сочетания клавиш Ctrl+Z в командной оболочке. Этот сигнал тут же останавливает получение процессом циклов CPU, вплоть до момента возобновления его через сигнал `SIGCONT`, после чего выполнение процесса продолжается с того места, на котором он был остановлен. Даже если в вашей среде обычно не применяется сигнал `SIGSTOP`, всегда существует вероятность отправки его случайно кем-то из обслуживающего персонала.

Каждое из описанных событий способно *вытеснить* работающий поток в любой момент и возобновить его выполнение позднее, причем совершенно незаметно для данного потока. Эта проблема напоминает обеспечение выполнения многопоточного кода на одной машине: нет никакой уверенности в хронометраже в силу параллелизма и возможности произвольных переключений контекста.

Существует немало инструментов для обеспечения безопасного выполнения многопоточного кода на отдельной машине: взаимные исключения (mutex), семафоры, атомарные счетчики, безблокировочные структуры данных, блокирующие очереди и т. д. К сожалению, эти утилиты нельзя непосредственно использовать для распределенных систем, поскольку в последних нет разделяемой памяти, а есть только отправляемые по ненадежной сети сообщения.

Узел в распределенной системе обязан учитывать возможность приостановки в любой момент на длительное время, даже посередине выполнения функции. Во время этой паузы все остальное вокруг продолжает работать, приостановленный узел даже может быть объявлен неработающим, поскольку не отвечает. В конце концов приостановленный узел возобновляет работу, вероятно даже не замечая данной паузы до момента проверки часов.

Гарантии времени отклика

Как уже обсуждалось, во многих языках программирования и операционных системах процессы и потоки выполнения могут приостанавливаться на неограниченное время. Если постараться, то вполне *можно* исключить причины подобных пауз.

Некоторое программное обеспечение работает в таких сферах, где отсутствие ответа в течение заданного времени способно привести к серьезному ущербу: компьютеры, управляющие самолетами, ракетами, роботами и автомобилями и другими физическими объектами, должны реагировать на входные сигналы датчиков быстро и предсказуемо. В таких системах устанавливается специальный *дедлайн* для отклика программного обеспечения: если приложение не укладывается в срок, то возможен крах всей системы. Такие системы называются *системами жесткого реального времени* (hard real-time systems).



Реально ли реальное время?

Во встроенных системах термин «реальное время» означает следующее: система тщательно спроектирована и протестирована так, чтобы соблюдать гарантии хронометража при всех обстоятельствах. Это «реальное время» отличается от одноименного, но более расплывчатого термина, используемого в веб-приложениях, где он означает отправку серверами данных клиентам и потоковую обработку без каких-либо жестких ограничений времени отклика (см. главу 11).

Например, если бортовые датчики вашего автомобиля обнаружили, что вы попали в аварию, то вряд ли вас устроит отсрочка выброса подушки безопасности из-за несвоевременной паузы для сборки мусора в системе выброса.

Обеспечение жестких гарантий работы в реальном времени требует поддержки на всех уровнях стека программного обеспечения: *операционной системы реального времени* (real-time operating system, RTOS), предоставляющей возможность планирования процессов с гарантированным выделением времени CPU в заданные промежутки времени; документированного наихудшего времени выполнения библиотечных функций; ограничения или полного запрета динамического выделения памяти (существуют сборщики мусора реального времени, но приложение все равно обязано гарантировать, что их объем работ не слишком велик). Кроме того,

необходимо выполнить колоссальный объем измерений и работ по тестированию для уверенности в предоставлении этих гарантий.

Все указанное требует массива дополнительной работы и серьезно ограничивает диапазон возможных языков программирования, библиотек и утилит (поскольку большинство языков и утилит не предоставляют гарантий работы в реальном времени). Поэтому разработка систем реального времени является чрезвычайно дорогостоящей, и они чаще всего используются в критически важных для безопасности встроенных устройствах. Более того, «реальное время» не синонимично «высокой производительности» — на самом деле пропускная способность систем реального времени может быть ниже, поскольку максимальный приоритет в них устанавливается для своевременности откликов (см. врезку «Время ожидания и использование ресурсов» в пункте «Нельзя ли просто сделать сетевые задержки предсказуемыми?» подраздела «Асинхронные и синхронные сети» раздела 8.2).

Для большинства серверных систем обработки данных гарантии работы в реальном времени экономически невыгодны. Следовательно, им приходится мириться с паузами и нестабильной работой часов вследствие работы в среде без реального времени.

Ограничение последствий сборки мусора

Отрицательные эффекты приостановки процессов можно смягчить, не прибегая к дорогостоящим гарантиям работы в реальном времени. Среды выполнения языков программирования проявляют некоторую гибкость в вопросе планирования сборки мусора, поскольку могут отслеживать частоту выделения памяти для объектов и оставшуюся свободную память.

Возникает идея рассматривать паузы сборщика мусора как короткие запланированные перебои в обслуживании узла и делегировать обработку запросов другим узлам на время сборки мусора на узле. Если бы среда выполнения уведомляла приложение о планируемой вскоре GC-паузе, то приложение могло бы прекратить отправку данному узлу новых запросов, подождать завершения ожидающих выполнения запросов, после чего произвести сборку мусора во время, когда запросы не обрабатываются. Описанный трюк скрывает от клиентов GC-паузы и снижает высокие процентили времени отклика [70, 71]. Такой подход используется в некоторых чувствительных к времени ожидания системах торговли на биржах [72].

Разновидность этой идеи — применять сборщик мусора только для короткоживущих объектов (которые можно обработать быстро) и периодического перезапуска процессов до того, как они накопят столько долгоживущих объектов, что понадобится полная сборка мусора долгоживущих объектов [65, 73]. За один раз можно перезапустить один узел, перенаправив с него трафик до запланированного перезапуска аналогично выполнению плавающего обновления (см. главу 4).

Эти меры не способны полностью предотвратить паузы на сборку мусора, но могут снизить их влияние на приложение.

8.4. Знание, истина и ложь

До сих пор в этой главе мы рассмотрели различия распределенных систем и программ, выполняемых на одном компьютере: отсутствие разделяемой памяти, передача сообщений только через ненадежную сеть с задержками переменной длительности и подверженность системы частичным отказам, ненадежности часов и паузам при обработке.

Если у вас нет опыта работы с распределенными системами, то последствия этих проблем могут оказаться весьма дезориентирующими. Узел сети ничего *не знает* наверняка — он способен только делать предположения на основе получаемых (или не получаемых) им по сети сообщений. Один узел в силе узнать состояние другого узла (какие данные на нем хранятся, правильно ли он работает), только обмениваясь с ним сообщениями. Если удаленный узел не отвечает, то нет никакого способа выяснить его состояние, поскольку невозможно отличить сетевые проблемы от проблем в узле.

Обсуждения этих систем граничат с философией: что в нашей системе правда, а что ложь? Можем ли мы полагаться на это знание, если механизмы познания и измерения ненадежны? Должны ли программные системы подчиняться законам физического мира, например закону причины и следствия?

К счастью, нам не требуется искать смысл жизни. Для распределенной системы можно описать допущения относительно поведения *(модель системы)* и спроектировать ее так, чтобы она соответствовала этим допущениям. Существует возможность проверки корректной работы алгоритмов в рамках определенной модели системы. А значит, надежность вполне достижима, даже если лежащая в основе системы модель обеспечивает очень мало гарантий.

Однако хотя можно обеспечить должную работу системы при ненадежной модели, сделать это непросто. В оставшейся части главы мы обсудим понятия знания и истины в распределенных системах, что поможет нам разобраться с нужными допущениями и гарантиями. В главе 9 мы перейдем к рассмотрению ряда примеров распределенных систем и алгоритмов, которые обеспечивают конкретные гарантии при конкретных допущениях.

Истина определяется большинством

Представьте сеть с асимметричным сбоем: узел получает все отправляемые ему сообщения, но все его исходящие сообщения задерживаются или вообще отбрасываются [19]. И хотя он отлично работает и получает запросы от других узлов, эти другие не получают его ответов. Как следствие, по истечении некоторого времени ожидания другие узлы объявляют его неработающим. Ситуация превращается в какой-то кошмар: наполовину отключенный от сети узел насильно тащат «на

кладбище», а он отбивается и кричит: «Я жив!». Но, поскольку никто его криков не слышит, похоронная процессия неуклонно продолжает движение.

В чуть-чуть менее кошмарном сценарии наполовину отключенный от сети узел может заметить отсутствие подтверждений доставки своих сообщений от других узлов и понимает, что имеет место сбой сети. Тем не менее другие узлы ошибочно объявляют наполовину отключенный узел неработающим, а он не способен ничего с этим поделать.

В качестве третьего сценария представьте узел, приостановленный на долгое время в связи с длительной всеобъемлющей сборкой мусора. Все его потоки вытесняются из памяти процессом сборки мусора и приостанавливаются на минуту, и, следовательно, запросы не обрабатываются, а ответы не отправляются. Другие узлы ждут, повторяют отправку, теряют терпение, в конце концов объявляют узел неработающим и «отправляют его в катафалк». Наконец процесс сборки мусора завершает работу и потоки узла возобновляют работу как ни в чем не бывало. Остальные узлы удивляются «воскрешению» в полном здравии объявленного «мертвым» узла и начинают радостно болтать с очевидцами этого. Сначала данный узел даже не понимает, что прошла целая минута и он был объявлен «мертвым», — с его точки зрения с момента последнего обмена сообщениями с другими узлами прошло едва ли не мгновение.

Мораль этих историй: узел не может полагаться только на свое собственное мнение о ситуации. Распределенная система не может всецело полагаться на один узел, поскольку он способен отказать в любой момент, приведя к отказу и невозможности восстановления системы. Вместо этого многие распределенные алгоритмы основывают свою работу на *кворуме*, то есть решении большинства узлов (см. пункт «Операции записи и чтения по кворуму» подраздела «Запись в базу данных при отказе одного из узлов» раздела 5.4): принятие решений требует определенного минимального количества «голосов» от нескольких узлов. Данное условие позволяет снизить зависимость от любого одного конкретного узла.

Это включает принятие решений об объявлении узлов неработающими. Если кворум узлов объявляет другой узел неработающим, то он считается таковым, пусть и прекрасно работает в то же время. Отдельные узлы обязаны подчиняться решениям кворума.

Обычно кворум представляет собой абсолютное большинство от более чем половины узлов (хотя существуют и другие разновидности кворумов). Кворум по большинству позволяет системе работать в случае сбоев отдельных узлов (при трех узлах допустим отказ одного, при пяти — отказ двух). Такой способ безопасен, поскольку система может насчитывать только одно большинство — два большинства одновременно, чьи решения конфликтуют, невозможны. Мы обсудим использование кворумов подробнее, когда доберемся до *алгоритмов консенсуса* в главе 9.

Ведущий узел и блокировки

Зачастую системе необходимо наличие только одного экземпляра чего-либо. Например:

- только один узел может быть ведущим для секции базы данных, чтобы избежать ситуации разделения вычислительных мощностей (см. подраздел «Перебои в обслуживании узлов» раздела 5.1);
- только одна транзакция или один клиент может удерживать блокировку на конкретный ресурс или объект, чтобы предотвратить конкурентную запись в него и его порчу;
- только один пользователь может зарегистрировать конкретное имя пользователя, поскольку оно должно идентифицировать пользователя уникальным образом.

Реализация этого в распределенной системе требует осторожности: даже если узел уверен в своей «избранности» (ведущий узел секции, владелец блокировки, обработчик запроса, успешно завладевшего именем пользователя), совсем не факт, что с этим согласен кворум остальных узлов! Узел ранее мог быть ведущим, но если затем другие узлы объявили его неработающим (например, вследствие разрыва сети или паузы на сборку мусора), то его вполне могли «понизить в должности» и выбрать другой ведущий узел.

Случай, когда узел продолжает вести себя как «избранный», несмотря на то, что кворум остальных узлов объявил его неработающим, может привести к проблемам в недостаточно тщательно спроектированной системе. Подобный узел способен отправлять сообщения другим узлам в качестве самозваного «избранного», и если другие узлы с этим согласятся, то система в целом может начать работать неправильно.

Например, на рис. 8.4 показана ошибка с порчей данных вследствие неправильной реализации блокировки (это отнюдь не теоретическая ошибка: она часто возникает в СУБД HBase [74, 75]). Допустим, мы хотели бы убедиться, что одновременно обратиться к находящемуся в сервисе хранения файлу может только один клиент, поскольку, если несколько клиентов сразу попытаются выполнить в него запись, файл будет испорчен. Мы попытаемся реализовать это с помощью обязательного получения клиентом договора аренды от сервиса блокировок до обращения к файлу.

Описанная проблема представляет собой пример того, что мы обсуждали в подразделе «Паузы при выполнении процессов» раздела 8.3: если арендующий клиент приостанавливается слишком надолго, то срок его договора аренды истекает. После этого другой клиент может получить договор аренды на тот же файл и начать запись в него данных. После возобновления работы приостановленный клиент верит

(ошибочно), что у него все еще есть действующий договор аренды, и переходит к записи в файл. В результате операции записи двух клиентов перемешиваются и файл портится.

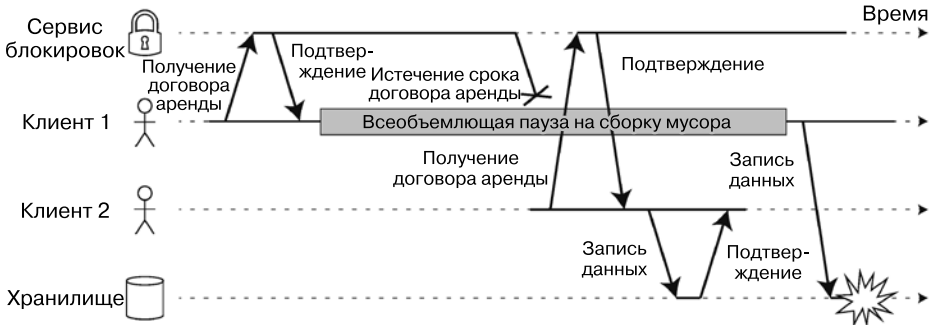


Рис. 8.4. Неправильная реализация распределенной блокировки: клиент 1 уверен, что у него все еще есть действующий договор аренды, на самом деле истекший, и вследствие этого портит файл в хранилище

Ограждающие маркеры

При использовании блокировки или договора аренды для защиты доступа к какому-либо ресурсу, например файловому хранилищу на рис. 8.4, необходимо убедиться, что узел, ошибочно считающий себя «избранным», не нарушит работу всей остальной системы. Для этой цели существует достаточно простой метод, показанный на рис. 8.5. Он называется *ограждением* (fencing).

Представим, что при каждом предоставлении блокировки или договора аренды сервер блокировок также возвращает *ограждающий маркер* (fencing token), представляющий собой номер, наращиваемый при каждом предоставлении блокировки (например, его может наращивать сервис блокировок). Кроме того, потребуем, чтобы клиент при каждой отправке запроса на запись сервису хранения включал в текущий запрос такой маркер.

На рис. 8.5 клиент 1 получает договор аренды с маркером 33, после чего надолго приостанавливается, и срок договора аренды истекает. Клиент 2 получает договор аренды с маркером 34 (номер монотонно возрастает), после чего отправляет запрос на запись сервису хранения, включая в запрос этот маркер. Позднее клиент 1 возобновляет работу и отправляет сервису хранения операцию записи с маркером 33. Однако сервис хранения помнит, что уже обработал операцию записи с большим номером маркера (34) и отклоняет данный запрос.

При использовании в качестве сервиса блокировок ZooKeeper в роли ограждающего маркера можно задействовать идентификатор транзакции `zxid` или версию узла `cversion`. Поскольку ZooKeeper гарантирует их монотонное возрастание, они обладают нужными для этого свойствами [74].

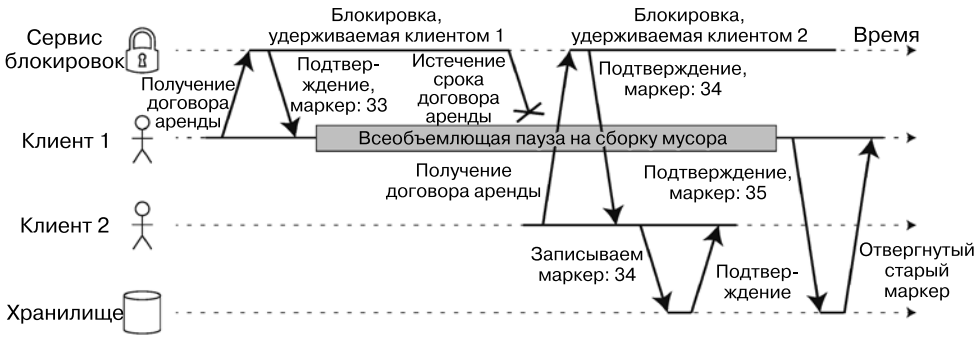


Рис. 8.5. Безопасный доступ к хранилищу благодаря тому, что операции записи разрешаются только в порядке возрастания ограждающих маркеров

Обратите внимание: этот механизм требует, чтобы сам ресурс принимал активное участие в проверке маркеров, отклоняя все операции записи с маркерами, более старыми, чем уже обработанные, — проверки статуса блокировки на самих клиентах недостаточно. Можно обойти ограничение и для ресурсов, не поддерживающих явным образом ограждающие маркеры (например, в случае сервиса хранения файлов ограждающий маркер включается в имя файла). Однако некоторая проверка все равно нужна, во избежание обработки запросов без защиты блокировкой.

Проверка маркеров на стороне сервера может показаться недостатком, но, вполне вероятно, это не так уж плохо: для сервиса было бы неразумно считать, что все его клиенты всегда «ведут себя хорошо», поскольку клиенты часто запускают люди совсем не с такими приоритетами, как у владельцев сервиса [76]. Следовательно, для любого сервиса будет хорошей идеей защищаться от непредумышленных неправильных действий со стороны клиентов.

Византийские сбои

Ограждающие маркеры способны обнаруживать и блокировать узел, *непреднамеренно* выполняющий ошибочные действия (например, потому, что еще не обнаружил истечение срока его договора аренды). Однако узел, умышленно желающий подорвать системные гарантии, может легко это сделать, отправив сообщение с поддельным маркером.

В этой книге мы предполагаем, что узлы ненадежны, но «добропорядочны»: они могут работать медленно или не отвечать вообще (вследствие сбоя), их состояние может быть устаревшим (из-за паузы на сборку мусора или сетевых задержек), но если узел *вообще* отвечает, то он «говорит правду» (соблюдает правила протокола в рамках имеющейся у него информации).

Проблемы распределенных систем значительно усугубляются при наличии риска, что узлы могут «лгать» (отправлять произвольные сбойные или поврежденные

ответы) — например, если узел может заявить о получении определенного сообщения, когда на самом деле этого не было. Подобное поведение носит название *византийского сбоя* (Byzantine fault), и задача достижения консенсуса в подобной, не заслуживающей доверия среде известна как *задача византийских генералов* (Byzantine generals problem) [77].

Задача византийских генералов

Задача византийских генералов представляет собой обобщение так называемой *задачи двух генералов* [78], описывающей ситуацию, в которой два командующих армиями генерала согласовывают между собой план сражения. Так как их лагеря находятся в двух разных местах, они могут общаться только через курьера, а курьеры иногда задерживаются или теряются (подобно пакетам в сети). Мы обсудим проблему *консенсуса* в главе 9.

В византийской версии данной задачи имеется n генералов, которым нужно согласовать между собой план, и их действиям мешает наличие среди них предателей. Большинство из генералов верны и посылают правдивые сообщения, но предатели могут попытаться обмануть и запутать остальных с помощью отправки поддельных или ложных сообщений (пытаясь при этом остаться нераскрытыми). Заранее неизвестно, кто предатели.

Византий — древний греческий город, позднее ставший Константинополем, на месте которого сейчас расположен Стамбул — столица Турции. Не существует никаких исторических свидетельств повышенной склонности византийских генералов к интригам и заговорам, по сравнению с остальными. Название происходит скорее от слова *«византийский»* в смысле «чрезмерно запутанный, бюрократический, неискренний», использовавшегося в политике задолго до эры компьютеров [79]. Автор этого названия Лэсли Лампорт (Lamport) хотел выбрать национальность, которая бы не обидела никого из читателей, и ему указали, что название *«задача албанских генералов»* будет не таким удачным [80].

Система защищена от византийских сбоев, если продолжает работать правильно даже в случае нарушения работоспособности некоторых узлов и не соблюдения ими протокола или при вмешательстве злоумышленников в работу сети. Это может быть важно в следующих обстоятельствах.

- В авиакосмической отрасли данные в памяти компьютеров или регистрах CPU могут портиться вследствие радиации, что приводит к отправке непредсказуемых ответов другим узлам. Поскольку отказ системы способен привести к катастрофическим последствиям (например, к падению самолета и гибели всех на борту или к столкновению ракеты с Международной космической станцией), системы управления полетом должны быть защищены от византийских сбоев [81, 82].
- Часть участников системы с множеством участвующих организаций могут попытаться сжульничать или обмануть остальных. В подобных обстоятельствах узлу не следует слепо верить сообщениям других узлов, поскольку они могут

быть отправлены со злым умыслом. Например, одноранговые сети, такие как Bitcoin и другие блочные цепи, можно рассматривать как способ для не доверяющих друг другу сторон договориться, произошла ли транзакция, не полагаясь на какой-либо центральный орган [83].

Однако в таких системах, которые мы обсуждаем в данной книге, обычно можно без опаски допустить, что византийских сбоев тут нет. В вашем ЦОДе все узлы контролируются вашей организацией (поэтому они, будем надеяться, доверенные), а уровни радиации достаточно низки для того, чтобы порча памяти не представляла серьезной проблемы. Протоколы создания защищенных от византийских сбоев систем достаточно сложны [84], и подобные встроенные системы требуют поддержки на аппаратном уровне [81]. В большинстве серверных информационных систем защищенные от византийских сбоев решения слишком дорого стоят, чтобы имело смысл их использовать.

В то же время веб-приложениям имеет смысл ожидать произвольного и злонамеренного поведения от находящихся под контролем конечных пользователей клиентов, например браузеров. Поэтому проверка вводимых данных, контроль корректности и экранирование вывода столь важны: например, для предотвращения внедрения SQL-кода (SQL injection) и межсайтового выполнения сценариев (cross-site scripting). Однако при этом обычно не применяются защищенные от византийских сбоев протоколы, а серверу просто делегируются полномочия принятия решения, допустимо ли поведение клиента. В одноранговых сетях, где подобного центрального органа нет, защищенность от византийских сбоев более уместна.

Можно рассматривать ошибки в программном обеспечении как византийские сбои, однако если одно и то же ПО используется во всех узлах, то алгоритмы защиты от византийских сбоев вас не спасут. Большинство таких алгоритмов требуют квалифицированного большинства из более чем двух третей нормально работающих узлов (то есть в случае четырех узлов может не работать максимум один). Чтобы решить проблему ошибок с помощью такого подхода, вам пришлось бы использовать четыре независимые реализации одного и того же программного обеспечения и надеяться на присутствие ошибки только в одной из них.

Аналогично кажется заманчивым, чтобы сам протокол защищал нас от уязвимостей, нарушений требований безопасности и злонамеренных действий. К сожалению, в большинстве систем это нереально. Если злоумышленник сумеет получить несанкционированный доступ к одному узлу, то очень вероятно, что сможет получить доступ и к остальным, поскольку на них, скорее всего, работает одно и то же программное обеспечение. Следовательно, традиционные механизмы (аутентификация, контроль доступа, шифрование, брандмауэры и т. д.) остаются основной защитой от атак.

Слабые формы «лжи». Хотя мы предполагаем, что узлы преимущественно «добропорядочны», имеет смысл добавить в ПО механизмы защиты от слабых форм «лжи» — например, некорректных сообщений вследствие аппаратных проблем,

ошибок в программном обеспечении и неправильных настроек. Подобные механизмы нельзя считать полномасштабной защитой от византийских сбоев, ведь они не смогут спасти от решительно настроенного злоумышленника, но это простые и практичные шаги по усилению надежности. Вот несколько примеров.

- Порча сетевых пакетов иногда происходит из-за проблем с аппаратным обеспечением или ошибок в операционной системе, драйверах, маршрутизаторах и т. п. Обычно поврежденные пакеты перехватываются при проверке контрольных сумм, встроенной в протоколы TCP и UDP, но иногда ускользают от проверки [85–87]. Для обнаружения порчи пакетов обычно достаточно простых мер, например контрольных сумм в протоколе уровня приложения.
- Открытые для общего доступа приложения должны тщательно контролировать корректность вводимых пользователями данных, например проверять, находятся ли значения в допустимом диапазоне, и ограничивать длину строк во избежание отказа в обслуживании вследствие выделения слишком больших объемов памяти. Для сервиса, применяемого внутри организации, за брандмауэром, достаточно менее строгих проверок вводимых данных, но некоторый простейший контроль значений (например, при синтаксическом разборе протокола [85]) не помешает.
- NTP-клиенты можно настроить для использования адресов нескольких серверов. Во время синхронизации клиент обращается ко всем из них, подсчитывает их ошибки и убеждается, что существует определенный промежуток времени, с которым согласно большинство серверов. Если большинство серверов функционируют нормально, то неправильно настроенный NTP-сервер, возвращающий ошибочное время, расценивается как аномалия и исключается из синхронизации [37]. Применение нескольких серверов повышает отказоустойчивость протокола NTP по сравнению с одним сервером.

Модели системы на практике

Для решения задач распределенных систем было разработано множество алгоритмов, например, мы рассмотрим решения для задачи консенсуса в главе 9. Чтобы принести какую-то пользу, эти алгоритмы должны уметь справляться с различными сбоями распределенных систем, которые мы обсуждали в этой главе.

Алгоритмы должны как можно меньше зависеть от особенностей аппаратного обеспечения и настроек ПО системы, на которой они работают. Это, в свою очередь, требует формализации типов вероятных сбоев. Для этого мы опишем *модель системы* — абстракцию, описывающую принимаемые алгоритмом допущения.

Что касается допущений по хронометражу, часто используются три системные модели.

- *Синхронная*. Предполагает ограниченность сетевых задержек, пауз процессов и расхождения часов. Это не подразумевает идеальной синхронизации часов или нулевой сетевой задержки, а просто означает, что длительность сетевых задержек, пауз процессов и расхождения часов никогда не превышает определенной

фиксированной верхней границы [88]. Синхронная модель нереалистична для большинства применяемых на практике систем, поскольку (как обсуждалось в данной главе) неограниченные задержки и паузы иногда случаются.

- *Частично синхронная.* Означает, что *большую часть времени* система ведет себя как синхронная, но иногда выходит за рамки заданной длительности сетевых задержек, пауз процессов и расхождения часов [88]. Это вполне реалистичная модель для большинства систем: значительную часть времени сети и процессы функционируют нормально — в противном случае вообще ничего бы не работало. Однако нужно учитывать такой факт: любые допущения относительно хронометража время от времени рассыпаются в прах. В подобном случае сетевые задержки, паузы и ошибки часов могут достигать произвольно больших значений.
- *Асинхронная.* В этой модели алгоритм не имеет права строить какие-либо временные допущения — на самом деле тут даже отсутствуют часы (так что нет и понятия времени ожидания). Некоторые алгоритмы можно приспособить для асинхронной модели, но она очень ограничивает разработчика.

Но, помимо проблем с хронометражем, следует учитывать и возможные отказы узлов. Вот три наиболее распространенные модели системы для узлов.

- *Модель «отказ — остановка».* Здесь алгоритм считает, что сбой узла может быть только фатальным. То есть узел способен перестать отвечать в любой момент, после чего его работа уже никогда не будет возобновлена.
- *Модель «отказ — восстановление».* Предполагается, что сбой узла может произойти в любой момент, но затем узел, вероятно, снова начнет отвечать через неизвестный период времени. В этой модели предполагается: у узлов имеется надежное хранилище (то есть энергонезависимый носитель информации), данные в котором не теряются при сбоях (в то время как находящиеся в оперативной памяти данные считаются потерянными).
- *Византийские (произвольные) сбои.* Узлы могут делать все, что им заблагорассудится, включая попытки обмана других узлов и умышленного введения их в заблуждение, как описано в последнем подразделе.

Для моделирования реальных систем обычно лучше всего подходит частично синхронная модель типа «сбой — восстановление». Но как с ней справляются распределенные алгоритмы?

Корректность алгоритмов

Чтобы дать определение *корректности* (correctness) алгоритма, я опишу его *свойства*. Например, у результатов работы алгоритма сортировки есть следующее свойство: для любых двух различных элементов выходного списка элемент слева меньше, чем элемент справа. Это просто формальный способ описания сортировки списка.

Аналогично формулируются свойства, которые требуются от корректного распределенного алгоритма. Например, при генерации ограждающих маркеров для блокировки

(см. пункт «Ограждающие маркеры» подраздела «Истина определяется большинством» текущего раздела) можно требовать от алгоритма следующие свойства:

- *уникальность* — никакие два запроса ограждающего маркера не приводят к возврату одинакового значения;
- *монотонное возрастание значений* — если запрос x возвращает маркер tx, а запрос y возвращает маркер ty, причем x завершается до начала выполнения y, то $tx < ty$;
- *доступность* — узел, с которым не произошел фатальный сбой, запросивший ограждающий маркер, в конце концов получает ответ на свой запрос.

Алгоритм корректен в некоторой модели системы при условии, что всегда удовлетворяет этим свойствам во всех ситуациях, способных, как мы предполагаем, возникнуть в данной модели системы. Но имеет ли это смысл? Если со всеми узлами произойдет фатальный сбой или все сетевые задержки внезапно затянутся до бесконечности, то никакой алгоритм ничего не сможет сделать.

Функциональная безопасность и живучесть

Для прояснения этой ситуации необходимо выделять два различных вида свойств: *функциональной безопасности* (safety) и *живучести* (liveness). В только что приведенном примере свойства *уникальности* и *монотонного возрастания значений* относятся к функциональной безопасности, а *доступность* — к живучести.

Чем отличаются эти две разновидности свойств? Отличительный признак: присутствие в определении свойств живучести фразы «в конце концов» (и да, вы совершенно правы: *конечная согласованность* — свойство живучести).

Функциональная безопасность часто неформально описывается фразой *«ничего плохого не произошло»*, а живучесть — *«со временем случится что-то хорошее»*. Однако лучше не увлекаться слишком сильно подобными неформальными определениями, поскольку слова «плохой» и «хороший» субъективны. Подлинные определения функциональной безопасности и живучести математически точны [90].

- Если свойство функциональной безопасности нарушается, то существует конкретный момент времени этого нарушения (например, при нарушении свойства уникальности можно определить конкретную операцию, при которой возвращен дублирующий маркер). При нарушении свойства функциональной безопасности ущерб уже был нанесен, ничего сделать с этим нельзя.
- Свойство живучести же, наоборот, может не быть привязанным к какому-либо моменту времени (например, узел мог отправить запрос, но все еще не получить ответа), но всегда есть надежда, что оно будет удовлетворено в дальнейшем (например, путем получения ответа).

Преимущество разделения на свойства функциональной безопасности и живучести — в упрощении работы со сложными моделями систем. В случае распределенных алгоритмов часто требуют, чтобы свойства функциональной безопасности

соблюдались *всегда*, во всех возможных ситуациях модели системы. То есть даже в случае фатального сбоя всех узлов или всей сети алгоритм должен гарантировать, что не вернет неправильный результат (то есть соблюдаются свойства функциональной безопасности).

Однако для свойств живучести вероятны уточнения: например, можно сказать, что ответ на запрос должен быть возвращен только в случае отсутствия фатального сбоя большинства узлов и если сеть в конце концов восстановилась после перебоя в обслуживании. Определение частично синхронной модели требует, чтобы система постепенно вернулась в синхронное состояние, то есть любой период прерывания работы сети длится только ограниченное время, после чего она восстанавливается.

Привязка моделей систем к реальному миру

Свойства функциональной безопасности и живучести, а также модели систем очень удобны для определения корректности распределенного алгоритма. Однако при реализации алгоритма на практике жестокая реальность вступает в свои права и становится ясно, что модель системы — лишь упрощенная абстракция реальности.

Например, алгоритмы в модели «фатальный сбой — восстановление» обычно предполагают: находящиеся в надежных хранилищах данные переживают фатальные сбои. Однако что будет в случае повреждения данных на диске или стирания данных из-за аппаратной ошибки или неправильной настройки [91]? А если в прошивке сервера содержится ошибка и он перестанет «видеть» свои жесткие диски после перезагрузки, хотя они и подключены к серверу должным образом [92]?

Алгоритмы кворума (см. пункт «Операции записи и чтения по кворуму» подраздела «Запись в базу данных при отказе одного из узлов» раздела 5.4) полагаются на то, что узлы запоминают данные, хранение которых декларировали. Возможность «амнезии» узла и забывания им ранее сохраненных данных нарушает условия кворума, а следовательно, и корректность алгоритма. Вероятно, нужна новая модель системы с допущением, что надежное хранилище в большинстве случаев переживает фатальные сбои, но иногда способно терять данные. Однако обосновать эту модель сложнее.

В теоретическом описании алгоритма может быть объявлено, что определенные вещи просто не должны случаться — и в невизантийских системах мы как раз и делаем допущения относительно того, какие сбои могут происходить, а какие — нет. Однако на практике в реализации иногда требуется включить код для обработки случая, когда произошло нечто, предполагавшееся невозможным, даже если эта обработка сводится к `printf("Тебе не повезло")` и `exit(666)`, то есть к тому, что разгребать все придется оператору-человеку [93]. (В этом, по мнению некоторых, заключается разница между компьютерными науками и программной инженерией.)

Это не значит, что теоретические, абстрактные системы ничего не стоят — как раз наоборот. Они исключительно полезны при извлечении из всей сложности реальной

системы приемлемого множества сбоев, которые можно рассмотреть с целью понять задачу и попытаться решить ее систематически. Доказать корректность алгоритма можно путем демонстрации того, что его свойства всегда соблюдаются в некой модели системы.

Доказательство корректности алгоритма не означает, что его *реализация* в реальной системе всегда будет вести себя корректно. Но это очень хороший первый шаг, поскольку теоретический анализ позволит выявить проблемы в алгоритме, которые могут оставаться скрытыми в реальной системе и проявить себя только в случае краха допущений (например, относительно хронометража) вследствие каких-то необычных обстоятельств. Теоретический анализ и эмпирическая проверка одинаково важны.

8.5. Резюме

В этой главе мы обсудили широкий спектр проблем, возникающих в распределенных системах.

- Вероятность потери или задержки на произвольное время отправленного по сети пакета. Аналогично может потеряться или задержаться ответ, так что при его отсутствии вы не будете знать, дошло ли ваше сообщение до адресата.
- Часы узла могут оказаться существенно рассинхронизированными с другими узлами (несмотря на все ваши старания по настройке NTP), неожиданно перескакивать вперед или назад во времени, и полагаться на них опасно, ведь вы не знаете их интервала погрешности.
- Выполнение процесса может быть приостановлено в любой момент на существенное время (например, вследствие всеобъемлющей паузы на сборку мусора), его могут объявить неработающим другие узлы, причем он способен потом возобновить работу, понятия не имея, что останавливался.

Возможность подобных *частичных отказов* (partial failures) — определяющая характеристика распределенных систем. При всяком действии программного обеспечения, вовлекающем использование других узлов, существует вероятность сбоя, или замедления работы, или того, что узел вообще перестанет отвечать (и постепенно время ожидания истечет). В распределенных системах мы пытаемся сделать ПО устойчивым к частичным отказам, чтобы система в целом продолжала функционировать даже при отказе некоторых ее элементов.

Для обеспечения устойчивости к отказам нужно сначала их *обнаружить*, но даже это непросто. В большинстве систем нет безошибочных механизмов обнаружения сбоев узлов, так что большинство распределенных алгоритмов определяет доступность удаленного узла по времени ожидания. Однако данный метод не позволяет различать сетевые отказы и отказы узлов, а неоднородность сетевых задержек иногда приводит к ошибочному предположению о фатальном сбое узла. Более того, иногда узел может работать хуже, чем обычно: например, гигабитное сете-

вое подключение способно внезапно снизить пропускную способность до 1 Кб/с вследствие ошибки в драйвере [94]. С подобным «хромающим», но продолжающим функционировать узлом иметь дело еще сложнее, чем с явно отказавшим.

Даже когда сбой обнаружен, справиться с ним — непростая задача для системы: нет ни глобальных переменных, ни разделяемой памяти, ни общих знаний, ни какого-либо другого разделяемого всеми машинами состояния. Узлы не могут даже согласовать время, не говоря уже о чем-то более существенном. Единственный способ передачи информации между узлами — отправка по ненадежной сети. Один узел не может принять никаких серьезных решений, так что необходимы протоколы, вовлекающие другие узлы и ищущие кворум для согласования решения.

Допустим, вы привыкли писать программное обеспечение в идеальном мире одного компьютера, где одна операция всегда детерминистически возвращает тот же самый результат. Тогда переход к запутанной реальности распределенных систем может оказаться для вас шоком. Напротив, разработчики этих систем зачастую считают задачу тривиальной, при условии, что ее можно решить на отдельном компьютере [5], а отдельный компьютер в наши дни способен на многое [95]. Если вы в состоянии не открывать ящик Пандоры и держать все на одной машине, то определенно есть смысл это сделать.

Однако, как обсуждалось во введении к данной части, масштабируемость не единственная причина, по которой может понадобиться распределенная система. Отказоустойчивость и низкое время ожидания (благодаря размещению данных географически близко к пользователям) — задачи одинаково важные, а их невозможно решить с помощью одного узла.

В этой главе мы также вскользь затронули вопрос о том, является ли ненадежность сетей, часов и процессов неизбежным законом природы. Мы увидели, что обеспечить жесткие гарантии реального времени ответа и ограниченные задержки сетей возможно, но очень дорогостояще и приводит к менее полному использованию имеющихся аппаратных ресурсов. Большинство не являющихся критически важными для безопасности систем выбирают дешевизну и ненадежность вместо дороговизны и надежности.

Кроме того, мы затронули вопрос суперкомпьютеров, предполагающих надежные компоненты, а следовательно, требующих полной остановки и перезапуска, если компонент все же откажет. Напротив, распределенные системы способны работать неограниченно долго без прерывания на уровне сервиса, поскольку все сбои и техническое обслуживание можно выполнить на уровне отдельных узлов — по крайней мере теоретически (на практике если «плохие» изменения конфигурации будут развернуты во всех узлах, то распределенная система все равно не сможет работать).

Эта глава была посвящена проблемам и обрисовала нам не очень-то радужные перспективы. В следующей главе мы перейдем к решениям и обсудим некоторые алгоритмы, разработанные специально для устранения всех проблем распределенных систем.

8.6. Библиография

1. *Cavage M.* There's Just No Getting Around It: You're Building a Distributed System // ACM Queue, volume 11, number 4, pages 80-89, April 2013 [Электронный ресурс]. — Режим доступа: http://queue.acm.org/detail.cfm?id=2482856.
2. *Kreps J.* Getting Real About Distributed System Reliability. March 19, 2012 [Электронный ресурс]. — Режим доступа: http://blog.empathybox.com/post/19574936361/getting-real-about-distributed-system-reliability.
3. *Padua S.* The Thrilling Adventures of Lovelace and Babbage: The (Mostly) True Story of the First Computer // Particular Books, April 2015.
4. *Hale C.* You Can't Sacrifice Partition Tolerance. October 7, 2010 [Электронный ресурс]. — Режим доступа: https://codahale.com/you-cant-sacrifice-partition-tolerance/.
5. *Hodges J.* Notes on Distributed Systems for Young Bloods. January 14, 2013 [Электронный ресурс]. — Режим доступа: https://www.somethingsimilar.com/2013/01/14/notes-on-distributed-systems-for-young-bloods/.
6. *Regalado A.* Who Coined 'Cloud Computing'? // technologyreview.com, October 31, 2011 [Электронный ресурс]. — Режим доступа: https://www.technologyreview.com/s/425970/who-coined-cloud-computing/.
7. *Barroso L. A., Clidaras J., Hölzle U.* The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition // Synthesis Lectures on Computer Architecture, volume 8, number 3, Morgan & Claypool Publishers, July 2013. [Электронный ресурс]. — Режим доступа: http://www.morganclaypool.com/doi/abs/10.2200/S00516ED2V01Y201306CAC024.
8. *Fiala D., Mueller F., Engelmann C., et al.* Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing // International Conference for High Performance Computing, Networking, Storage and Analysis (SC12), November 2012 [Электронный ресурс]. — Режим доступа: http://moss.csc.ncsu.edu/~mueller/ftp/pub/mueller/papers/sc12.pdf.
9. *Singh A., Ong J., Agarwal A., et al.* Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network // at Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), August 2015 [Электронный ресурс]. — Режим доступа: http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/p183.pdf.
10. *Lockwood G. K.* Hadoop's Uncomfortable Fit in HPC. May 16, 2014 [Электронный ресурс]. — Режим доступа: https://glennklockwood.blogspot.com.by/2014/05/hadoops-uncomfortable-fit-in-hpc.html.
11. *Neumann von J.* Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components // In Automata Studies (AM-34), edited by Claude E. Shannon and John McCarthy, Princeton University Press, 1956 [Электронный ресурс]. — Режим доступа: https://ece.uwaterloo.ca/~ssundara/courses/prob_logics.pdf.

12. *Hamming R. W.* The Art of Doing Science and Engineering. —Taylor & Francis, 1997.
13. *Shannon C. E.* A Mathematical Theory of Communication // The Bell System Technical Journal, volume 27, number 3, pages 379–423 and 623–656, July 1948 [Электронный ресурс]. — Режим доступа: https://cs.brynmawr.edu/Courses/cs380/fall2012/shannon1948.pdf.
14. *Bailis P., Kingsbury K.* The Network Is Reliable // ACM Queue, volume 12, number 7, pages 48-55, July 2014 [Электронный ресурс]. — Режим доступа: https://queue.acm.org/detail.cfm?id=2655736.
15. *Leners J. B., Gupta T., Aguilera M. K., Walfish M.* Taming Uncertainty in Distributed Systems with Help from the Network // 10th European Conference on Computer Systems (EuroSys), April 2015 [Электронный ресурс]. — Режим доступа: http://www.cs.nyu.edu/~mwalfish/papers/albatross-eurosys15.pdf.
16. *Gill P., Jain N., Nagappan N.* Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications // ACM SIGCOMM Conference, August 2011 [Электронный ресурс]. — Режим доступа: http://conferences.sigcomm.org/sigcomm/2011/papers/sigcomm/p350.pdf.
17. *Imbriaco M.* Downtime Last Saturday. December 26, 2012 [Электронный ресурс]. — Режим доступа: https://github.com/blog/1364-downtime-last-saturday.
18. *Oremus W.* The Global Internet Is Being Attacked by Sharks, Google Confirms. August 15, 2014 [Электронный ресурс]. — Режим доступа: http://www.slate.com/blogs/future_tense/2014/08/15/shark_attacks_threaten_google_s_undersea_internet_cables_video.html.
19. *Donges M. A.* Re: bnx2 cards Intermittantly Going Offline // Message to Linux netdev mailing list, September 13, 2012 [Электронный ресурс]. — Режим доступа: http://www.spinics.net/lists/netdev/msg210485.html.
20. *Kingsbury K.* Call Me Maybe: Elasticsearch. June 15, 2014 [Электронный ресурс]. — Режим доступа: https://aphyr.com/posts/317-call-me-maybe-elasticsearch.
21. *Sanfilippo S.* A Few Arguments About Redis Sentinel Properties and Fail Scenarios. October 21, 2014 [Электронный ресурс]. — Режим доступа: http://antirez.com/news/80.
22. *Hubert B.* The Ultimate SO_LINGER Page, or: Why Is My TCP Not Reliable. January 18, 2009 [Электронный ресурс]. — Режим доступа: https://blog.netherlabs.nl/articles/2009/01/18/the-ultimate-so_linger-page-or-why-is-my-tcp-not-reliable.
23. *Liochon N.* CAP: If All You Have Is a Timeout, Everything Looks Like a Partition. May 25, 2015 [Электронный ресурс]. — Режим доступа: http://blog.thislongrun.com/2015/05/CAP-theorem-partition-timeout-zookeeper.html.
24. *Saltzer J. H., Reed D. P., Clark D. D.* End-To-End Arguments in System Design // ACM Transactions on Computer Systems, volume 2, number 4, pages 277–288, November 1984 [Электронный ресурс]. — Режим доступа: http://www.ece.drexel.edu/courses/ECE-C631-501/SalRee1984.pdf.

25. *Grosvenor M. P., Schwarzkopf M., Gog I., et al.* Queues Don't Matter When You Can JUMP Them! // 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI), May 2015 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-grosvenor_update.pdf.
26. *Wang G., Ng T. S. E.* The Impact of Virtualization on Network Performance of Amazon EC2 Data Center // 29th IEEE International Conference on Computer Communications (INFOCOM), March 2010 [Электронный ресурс]. — Режим доступа: https://www.cs.rice.edu/~eugeneng/papers/INFOCOM10-ec2.pdf.
27. *Jacobson V.* Congestion Avoidance and Control // ACM Symposium on Communications Architectures and Protocols (SIGCOMM), August 1988 [Электронный ресурс]. — Режим доступа: https://www.cs.usask.ca/ftp/pub/discus/seminars2002-2003/p314-jacobson.pdf.
28. *Philips B.* etcd: Distributed Locking and Service Discovery // Strange Loop, September 2014 [Электронный ресурс]. — Режим доступа: https://www.youtube.com/watch?v=HJIjTTHWYnE.
29. *Newman S.* A Systematic Look at EC2 I/O. October 16, 2012 [Электронный ресурс]. — Режим доступа: http://blog.scalyr.com/2012/10/a-systematic-look-at-ec2-io/.
30. *Hayashibara N., Défago X., Yared R., Katayama T.* The φ Accrual Failure Detector // Japan Advanced Institute of Science and Technology, School of Information Science, Technical Report IS-RR-2004-010, May 2004 [Электронный ресурс]. — Режим доступа: https://dspace.jaist.ac.jp/dspace/handle/10119/4784.
31. *Wang J.* Phi Accrual Failure Detector. August 11, 2013 [Электронный ресурс]. — Режим доступа: http://ternarysearch.blogspot.com.by/2013/08/phi-accrual-failure-detector.html.
32. *Keshav S.* An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network. — Addison-Wesley Professional, May 1997.
33. Cisco, Integrated Services Digital Network [Электронный ресурс]. — Режим доступа: http://docwiki.cisco.com/wiki/Integrated_Services_Digital_Network.
34. *Kyas O.* ATM Networks. — International Thomson Publishing, 1995.
35. InfiniBand FAQ // Mellanox Technologies, December 22, 2014 [Электронный ресурс]. — Режим доступа: http://www.mellanox.com/related-docs/whitepapers/InfiniBandFAQ_FQ_100.pdf.
36. *Santos J. R., Turner Y., Janakiraman G. (J.)* End-to-End Congestion Control for InfiniBand // 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), April 2003. Also published by HP Laboratories Palo Alto, Tech Report HPL-2002-359 [Электронный ресурс]. — Режим доступа: http://www.hpl.hp.com/techreports/2002/HPL-2002-359.pdf.
37. *Windl U., Dalton D., Martinec M., Worley D. R.* The NTP FAQ and HOWTO. November 2006 [Электронный ресурс]. — Режим доступа: http://www.ntp.org/ntpfaq/NTP-a-faq.htm.

38. *Graham-Cumming J.* How and why the leap second affected Cloudflare DNS. January 1, 2017 [Электронный ресурс]. — Режим доступа: https://blog.cloudflare.com/how-and-why-the-leap-second-affected-cloudflare-dns/.
39. *Holmes D.* Inside the Hotspot VM: Clocks, Timers and Scheduling Events. — Part I — Windows. October 2, 2006 [Электронный ресурс]. — Режим доступа: https://blogs.oracle.com/dholmes/entry/inside_the_hotspot_vm_clocks.
40. *Loughran S.* Time on Multi-Core, Multi-Socket Servers. September 17, 2015 [Электронный ресурс]. — Режим доступа: http://steveloughran.blogspot.com.by/2015/09/time-on-multi-core-multi-socket-servers.html.
41. *Corbett J. C., Dean J., Epstein M., et al.* Spanner: Google's Globally-Distributed Database // at 10th USENIX Symposium on Operating System Design and Implementation (OSDI), October 2012 [Электронный ресурс]. — Режим доступа: https://research.google.com/archive/spanner.html.
42. *Caporaloni M., Ambrosini R.* How Closely Can a Personal Computer Clock Track the UTC Timescale Via the Internet? // European Journal of Physics, volume 23, number 4, pages L17–L21, June 2012 [Электронный ресурс]. — Режим доступа: http://iopscience.iop.org/article/10.1088/0143-0807/23/4/103/meta;jsessionid=9600D7C1936C7DAC92ED28B32FED6D80.c2.iopscience.cld.iop.org.
43. *Minar N.* A Survey of the NTP Network. December 1999 [Электронный ресурс]. — Режим доступа: http://alumni.media.mit.edu/~nelson/research/ntp-survey99/.
44. *Holub V.* Synchronizing Clocks in a Cassandra Cluster Pt. 1 — The Problem. March 14, 2014 [Электронный ресурс]. — Режим доступа: https://blog.logentries.com/2014/03/synchronizing-clocks-in-a-cassandra-cluster-pt-1-the-problem/.
45. *Kamp P.–H.* The One-Second War (What Time Will You Die?) // ACM Queue, volume 9, number 4, pages 44–48, April 2011 [Электронный ресурс]. — Режим доступа: http://queue.acm.org/detail.cfm?id=1967009.
46. *Minar N.* Leap Second Crashes Half the Internet. July 3, 2012 [Электронный ресурс]. — Режим доступа: http://www.somebits.com/weblog/tech/bad/leap-second-2012.html.
47. *Pascoe C.* Time, Technology and Leaping Seconds. September 15, 2011 [Электронный ресурс]. — Режим доступа: https://googleblog.blogspot.com.by/2011/09/time-technology-and-leaping-seconds.html.
48. *Zhao M., Barr J.* Look Before You Leap — The Coming Leap Second and AWS. May 18, 2015 [Электронный ресурс]. — Режим доступа: https://aws.amazon.com/ru/blogs/aws/look-before-you-leap-the-coming-leap-second-and-aws/.
49. *Veitch D., Vijayalayan K.* Network Timing and the 2015 Leap Second // 17th International Conference on Passive and Active Measurement (PAM), April 2016 [Электронный ресурс]. — Режим доступа: http://crin.eng.uts.edu.au/~darryl/Publications/LeapSecond_camera.pdf.
50. Timekeeping in VMware Virtual Machines // Information Guide, VMware, Inc., December 2011 [Электронный ресурс]. — Режим доступа: https://www.vmware.com/techpapers/2006/timekeeping-in-vmware-virtual-machines-238.html.

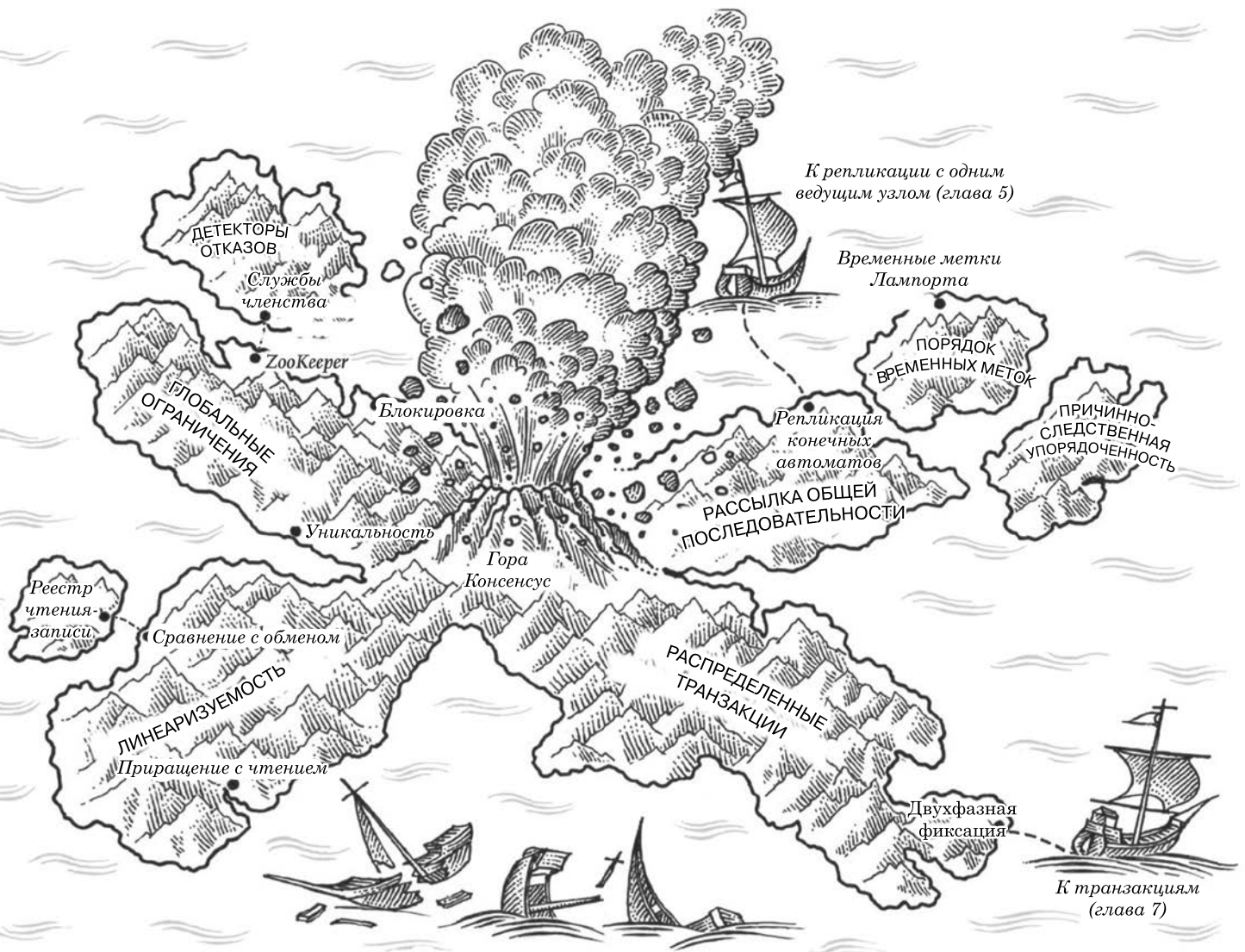
51. MiFID II / MiFIR: Regulatory Technical and Implementing Standards — Annex I (Draft) // European Securities and Markets Authority, Report ESMA/2015/1464, September 2015 [Электронный ресурс]. — Режим доступа: https://www.esma.europa.eu/sites/default/files/library/2015/11/2015-esma-1464_annex_i_-_draft_rts_and_its_on_mifid_ii_and_mifir.pdf.
52. *Bigum L.* Solving MiFID II Clock Synchronisation With Minimum Spend (Part 1). November 27, 2015 [Электронный ресурс]. — Режим доступа: https://www.lmax.com/blog/staff-blogs/2015/11/27/solving-mifid-ii-clock-synchronisation-minimum-spend-part-1/.
53. *Kingsbury K.* Call Me Maybe: Cassandra. September 24, 2013 [Электронный ресурс]. — Режим доступа: https://aphyr.com/posts/294-call-me-maybe-cassandra/.
54. *Daily J.* Clocks Are Bad, or, Welcome to the Wonderful World of Distributed Systems. November 12, 2013 [Электронный ресурс]. — Режим доступа: http://basho.com/posts/technical/clocks-are-bad-or-welcome-to-distributed-systems/.
55. *Kingsbury K.* The Trouble with Timestamps. October 12, 2013 [Электронный ресурс]. — Режим доступа: https://aphyr.com/posts/299-the-trouble-with-timestamps.
56. *Lamport L.* Time, Clocks, and the Ordering of Events in a Distributed System // Communications of the ACM, volume 21, number 7, pages 558–565, July 1978 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Flamport%2Fpubs%2Ftime-clocks.pdf.
57. *Kulkarni S., Demirbas M., Madeppa D., et al.* Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases // State University of New York at Buffalo, Computer Science and Engineering Technical Report 2014-04, May 2014 [Электронный ресурс]. — Режим доступа: https://www.cse.buffalo.edu//tech-reports/2014-04.pdf.
58. *Sheehy J.* There Is No Now: Problems With Simultaneity in Distributed Systems // ACM Queue, volume 13, number 3, pages 36–41, March 2015 [Электронный ресурс]. — Режим доступа: https://queue.acm.org/detail.cfm?id=2745385.
59. *Demirbas M.* Spanner: Google's Globally-Distributed Database. July 4, 2013 [Электронный ресурс]. — Режим доступа: http://muratbuffalo.blogspot.com.by/2013/07/spanner-googles-globally-distributed_4.html.
60. *Malkhi D., Martin J.-P.* Spanner's Concurrency Control // ACM SIGACT News, volume 44, number 3, pages 73–77, September 2013 [Электронный ресурс]. — Режим доступа: http://www.cs.cornell.edu/~ie53/publications/DC-col51-Sep13.pdf.
61. *Bravo M., Diegues N., Zeng J., et al.* On the Use of Clocks to Enforce Consistency in the Cloud // IEEE Data Engineering Bulletin, volume 38, number 1, pages 18–31, March 2015 [Электронный ресурс]. — Режим доступа: http://sites.computer.org/debull/A15mar/p18.pdf.

62. *Kimball S.* Living Without Atomic Clocks. February 17, 2016 [Электронный ресурс]. — Режим доступа: https://www.cockroachlabs.com/blog/living-without-atomic-clocks/.
63. *Gray C. G., Cheriton D. R.* Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency // at 12th ACM Symposium on Operating Systems Principles (SOSP), December 1989 [Электронный ресурс]. — Режим доступа: http://web.stanford.edu/class/cs240/readings/89-leases.pdf.
64. *Lipcon T.* Avoiding Full GCs in Apache HBase with MemStore-Local Allocation Buffers: Part 1. February 24, 2011 [Электронный ресурс]. — Режим доступа: http://blog.cloudera.com/blog/2011/02/avoiding-full-gcs-in-hbase-with-memstore-local-allocation-buffers-part-1/.
65. *Thompson M.* Java Garbage Collection Distilled. July 16, 2013 [Электронный ресурс]. — Режим доступа: https://mechanical-sympathy.blogspot.com.by/2013/07/java-garbage-collection-distilled.html.
66. *Ragozin A.* How to Tame Java GC Pauses? Surviving 16GiB Heap and Greater. June 28, 2011 [Электронный ресурс]. — Режим доступа: https://dzone.com/articles/how-tame-java-gc-pauses.
67. *Clark C., Fraser K., Hand S., et al.* Live Migration of Virtual Machines // 2nd USENIX Symposium on Symposium on Networked Systems Design & Implementation (NSDI), May 2005 [Электронный ресурс]. — Режим доступа: http://www.cl.cam.ac.uk/research/srg/netos/papers/2005-nsdi-migration.pdf.
68. *Shaver M.* fsyncers and Curveballs. May 25, 2008 [Электронный ресурс]. — Режим доступа: http://shaver.off.net/diary/2008/05/25/fsyncers-and-curveballs/.
69. *Zhuang Z., Tran C.* Eliminating Large JVM GC Pauses Caused by Background IO Traffic. February 10, 2016 [Электронный ресурс]. — Режим доступа: https://engineering.linkedin.com/blog/2016/02/eliminating-large-jvm-gc-pauses-caused-by-background-io-traffic.
70. *Terei D., Levy A.* Blade: A Data Center Garbage Collector. April 13, 2015 [Электронный ресурс]. — Режим доступа: https://arxiv.org/pdf/1504.02578.pdf.
71. *Maas M., Harris T., Asanović K., Kubiatowicz J.* Trash Day: Coordinating Garbage Collection in Distributed Systems // 15th USENIX Workshop on Hot Topics in Operating Systems (HotOS), May 2015 [Электронный ресурс]. — Режим доступа: https://timharris.uk/papers/2015-hotos.pdf.
72. Predictable Low Latency // Cinnober Financial Technology AB, November 24, 2013 [Электронный ресурс]. — Режим доступа: https://cdn2.hubspot.net/hubfs/1624455/Website_2016/Content/Whitepapers/Cinnober%20on%20GC%20pause%20free%20Java%20applications.pdf.
73. *Fowler M.* The LMAX Architecture. July 12, 2011 [Электронный ресурс]. — Режим доступа: https://martinfowler.com/articles/lmax.html.
74. *Junqueira F. P., Reed B.* ZooKeeper: Distributed Process Coordination. — O'Reilly Media, 2013.

75. *Söztutar E.* HBase and HDFS: Understanding Filesystem Usage in HBase // HBaseCon, June 2013 [Электронный ресурс]. — Режим доступа: https://www.slideshare.net/enissoz/hbase-and-hdfs-understanding-filesystem-usage.
76. *McCaffrey C.* Clients Are Jerks: AKA How Halo 4 DoSed the Services at Launch & How We Survived. June 23, 2015 [Электронный ресурс]. — Режим доступа: https://caitiem.com/2015/06/23/clients-are-jerks-aka-how-halo-4-dosed-the-services-at-launch-how-we-survived/.
77. *Lamport L., Shostak R., Pease M.* The Byzantine Generals Problem // ACM Transactions on Programming Languages and Systems (TOPLAS), volume 4, number 3, pages 382–401, July 1982 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Flamport%2Fpubs%2Fbyz.pdf.
78. *Gray J. N.* Notes on Data Base Operating Systems // Operating Systems: An Advanced Course, Lecture Notes in Computer Science, volume 60, edited by R. Bayer, R. M. Graham, and G. Seegmüller, pages 393–481, Springer-Verlag, 1978. http://jimgray.azurewebsites.net/papers/dbos.pdf.
79. *Palmer B.* How Complicated Was the Byzantine Empire? October 20, 2011 [Электронный ресурс]. — Режим доступа: http://www.slate.com/articles/news_and_politics/explainer/2011/10/the_byzantine_tax_code_how_complicated_was_byzantium_anyway_.html.
80. *Lamport L.* My Writings. December 16, 2014 [Электронный ресурс]. — Режим доступа: http://lamport.azurewebsites.net/pubs/pubs.html.
81. *Rushby J.* Bus Architectures for Safety-Critical Embedded Systems // 1st International Workshop on Embedded Software (EMSOFT), October 2001 [Электронный ресурс]. — Режим доступа: http://www.csl.sri.com/papers/emsoft01/emsoft01.pdf.
82. *Edge J.* ELC: SpaceX Lessons Learned. March 6, 2013 [Электронный ресурс]. — Режим доступа: https://lwn.net/Articles/540368/.
83. *Miller A., LaViola J. Jr.* Anonymous Byzantine Consensus from Moderately-Hard Puzzles: A Model for Bitcoin // University of Central Florida, Technical Report CS-TR-14-01, April 2014 [Электронный ресурс]. — Режим доступа: http://nakamotoinstitute.org/static/docs/anonymous-byzantine-consensus.pdf.
84. *Mickens J.* The Saddest Moment // USENIX ;login: logout, May 2013 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/system/files/login-logout_1305_mickens.pdf.
85. *Gilman E.* The Discovery of Apache ZooKeeper's Poison Packet. May 7, 2015 [Электронный ресурс]. — Режим доступа: https://www.pagerduty.com/blog/the-discovery-of-apache-zookeepers-poison-packet/.
86. *Stone J., Partridge C.* When the CRC and TCP Checksum Disagree // ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), August 2000 [Электронный ресурс]. — Режим

доступа: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.27.7611&rep=rep1&type=pdf.

87. *Jones E.* How Both TCP and Ethernet Checksums Fail. October 5, 2015 [Электронный ресурс]. — Режим доступа: http://www.evanjones.ca/tcp-and-ethernet-checksums-fail.html.
88. *Dwork C., Lynch N., Stockmeyer L.* Consensus in the Presence of Partial Synchrony // Journal of the ACM, volume 35, number 2, pages 288–323, April 1988 [Электронный ресурс]. — Режим доступа: https://www.net.t-labs.tu-berlin.de/~petr/ADC-07/papers/DLS88.pdf.
89. *Bailis P., Ghodsi A.* Eventual Consistency Today: Limitations, Extensions, and Beyond // ACM Queue, volume 11, number 3, pages 55-63, March 2013 [Электронный ресурс]. — Режим доступа: http://queue.acm.org/detail.cfm?id=2462076.
90. *Alpern B., Schneider F. B.* Defining Liveness // Information Processing Letters, volume 21, number 4, pages 181–185, October 1985 [Электронный ресурс]. — Режим доступа: https://www.cs.cornell.edu/fbs/publications/DefLiveness.pdf.
91. *Junqueira F. P.* Dude, Where's My Metadata? May 28, 2015 [Электронный ресурс]. — Режим доступа: https://fpj.me/2015/05/28/dude-wheres-my-metadata/.
92. *Sanders S.* January 28th Incident Report. February 3, 2016 [Электронный ресурс]. — Режим доступа: https://github.com/blog/2106-january-28th-incident-report.
93. *Kreps J.* A Few Notes on Kafka and Jepsen. September 25, 2013 [Электронный ресурс]. — Режим доступа: http://blog.empathybox.com/post/62279088548/a-few-notes-on-kafka-and-jepsen.
94. *Do T., Hao M., Leesatapornwongsa T., et al.* Limplock: Understanding the Impact of Limpware on Scale-out Cloud Systems // 4th ACM Symposium on Cloud Computing (SoCC), October 2013 [Электронный ресурс]. — Режим доступа: http://ucare.cs.uchicago.edu/pdf/socc13-limplock.pdf.
95. *McSherry F., Isard M., Murray D. G.* Scalability! But at What COST? // 15th USENIX Workshop on Hot Topics in Operating Systems (HotOS), May 2015 [Электронный ресурс]. — Режим доступа: http://www.frankmcsherry.org/assets/COST.pdf.
96. *Падуа С.* Невероятные приключения Лавлейс и Бэббиджа. (Почти) правдивая история первого компьютера. — М.: Манн, Иванов и Фербер, 2017. — 320 с.


К репликации с одним ведущим узлом (глава 5)
ДЕТЕКТОРЫ ОТКАЗОВ
Службы членства
ZooKeeper
Временные метки Лампорта
ПОРЯДОК ВРЕМЕННЫХ МЕТОК
ГЛОБАЛЬНЫЕ ОГРАНИЧЕНИЯ
Блокировка
Репликация конечных автоматов
ПРИЧИННО-СЛЕДСТВЕННАЯ УПОРЯДОЧЕННОСТЬ
РАССЫЛКА ОБЩЕЙ ПОСЛЕДОВАТЕЛЬНОСТИ
Уникальность
Гора Консенсус
Реестр чтения-записи
Сравнение с обменом
ЛИНЕАРИЗУЕМОСТЬ
Приращение с чтением
РАСПРЕДЕЛЕННЫЕ ТРАНЗАКЦИИ
Двухфазная фиксация
К транзакциям (глава 7)
ОБЛОМКИ ДОМОРОЩЕННЫХ АЛГОРИТМОВ КОНСЕНСУСА

9 Согласованность и консенсус

> Что лучше: быть неправым и живым или правым, но мертвым?
>
> *Джей Крепс. Несколько заметок о Кафке и Джепсене (2013)*

Как было показано в главе 8, есть множество причин, способных нарушить работу распределенных систем. Самый простой способ обработки таких сбоев — отключить весь сервис и показать пользователю сообщение об ошибке. При невозможности воплотить это решение следует найти способы *справляться с ошибками*, то есть обеспечивать правильную работу сервиса, даже если один из его внутренних компонентов неисправен.

В этой главе мы рассмотрим несколько примеров алгоритмов и протоколов для построения отказоустойчивых распределенных систем. Мы будем исходить из предположения, что в системе способны возникнуть все проблемы, представленные в главе 8: пакеты могут потеряться, поступать в неверной последовательности, дублироваться или задерживаться в сети на произвольный срок; часы работают в лучшем случае приблизительно, а работа узлов может в любой момент приостановиться (например, для сбора мусора) или вообще прекратиться из-за сбоя.

Лучший способ построить отказоустойчивую систему — создать некоторые общие абстракции с полезными гарантиями, реализовать их один раз, а затем позволить приложениям полагаться на эти гарантии. Данный подход аналогичен случаю с транзакциями (см. главу 7): при использовании транзакции приложение может симулировать отсутствие сбоев (атомарность), отсутствие конкурентного обращения к базе данных (изоляция) и абсолютную надежность устройств хранения (долговечность). В действительности сбои, условия гонки и отказы дисков время

от времени возникают, но абстракция транзакций скрывает эти проблемы, так что приложение может о них не беспокоиться.

Мы продолжим двигаться в том же направлении — будем искать абстракции, которые позволят приложению игнорировать отдельные проблемы, возникающие в распределенных системах. Например, одной из самых важных абстракций для распределенных систем является *консенсус* — согласованность между всеми узлами по какому-то вопросу. Как мы скоро увидим, надежное обеспечение консенсуса, несмотря на сетевые сбои и отказы процессов, — удивительно сложная задача.

При достижении консенсуса приложения могут использовать его для самых разных целей. Например, предположим, что имеется база данных с репликацией с одним ведущим узлом. Если он отключился и нужно перейти в другой узел, то остальные узлы базы могут применять консенсус для выбора нового ведущего узла. Как описано в подразделе «Перебои в обслуживании узлов» раздела 5.1, важно, чтобы ведущий узел был только один и все узлы были единого мнения о том, кто является таковым. Ситуация, в которой два узла считают себя ведущими, называется *разделением интеллекта* и часто приводит к потере данных. Правильные реализации консенсуса помогают избежать подобных проблем.

Далее в этой главе, в разделе 9.4, мы рассмотрим алгоритмы достижения консенсуса и решения связанных с ним проблем. Но сначала узнаем, какие гарантии и абстракции возможно предоставить в распределенной системе.

Нужно понять масштаб того, что можно и чего нельзя сделать: в некоторых ситуациях система способна допускать ошибки и продолжать работать; в других ситуациях такое невозможно. Эти границы осуществимого были подробно исследованы как в виде теоретических доказательств, так и в практических реализациях. В данной главе мы рассмотрим основные ограничения.

Исследователи распределенных систем изучали эти темы десятки лет. Материала накопилось много, так что мы лишь пройдемся по его поверхности. В нашей книге не хватит места для подробного описания формальных моделей и доказательств, поэтому мы лишь кратко рассмотрим их. Если вы заинтересуетесь подробностями, то воспользуйтесь библиографическим списком.

9.1. Гарантии согласованности

В разделе 5.2 были рассмотрены некоторые проблемы хронометража, возникающие в реплицированной базе данных. Если посмотреть на два узла БД в один и тот же момент времени, то, скорее всего, можно увидеть в них различные данные, потому что запросы на запись поступают на разные узлы в разное время. Эти несоответствия возникают независимо от используемого в базе данных метода репликации (с одним ведущим узлом, с несколькими ведущими узлами или без ведущего).

Большинство реплицированных БД обеспечивают по меньшей мере *конечную согласованность*. Это означает, что если прекратить записывать данные в базу и подождать некоторое неопределенное время, то в конце концов все запросы на чтение вернут одно и то же значение [1]. Другими словами, несогласованность является временной и в конечном итоге устраняется сама собой (при условии устранения в итоге и всех неисправностей в сети). Лучшим названием для конечной согласованности могла бы быть *конвергенция*, поскольку мы ожидаем, что все реплики со временем сойдутся к одному и тому же значению [2].

Однако это очень слабая гарантия. Она ничего не говорит о том, *когда именно* реплики сойдутся. До момента конвергенции операции чтения могут возвращать что угодно или вообще ничего [1]. Например, если вы запишете значение и сразу же прочтете его снова, то нет никакой гарантии, что увидите то же значение, которое написали, поскольку чтение может быть перенаправлено на другую реплику (см. подраздел «Читаем свои же записи» раздела 5.2).

Конечная согласованность неудобна для разработчиков приложений: она слишком сильно отличается от поведения переменных в обычной однопоточной программе. Если вы присвоите значение переменной и сразу ее прочитаете, то вряд ли будете ожидать, что прочитаете старое значение или что операция чтения вообще не удастся. Внешне база данных выглядит как переменная, которую можно читать и записывать, но на самом деле ее семантика гораздо сложнее [3].

При работе с базой данных, обеспечивающей только слабые гарантии, приходится постоянно помнить о ее ограничениях и не рассчитывать на слишком многое. Ошибки в этой области часто трудно найти путем тестирования, потому что большую часть времени приложение может работать хорошо. Пограничные случаи конечной согласованности становятся очевидными только при сбоях в системе (например, при прерывании соединения с сетью) или большом количестве конкурентных процессов.

В этой главе мы рассмотрим более сильные модели согласованности, пригодные для реализации в информационных системах. У каждой из них есть своя цена: системы с более надежными гарантиями могут иметь худшую производительность или быть менее отказоустойчивыми, чем системы с более слабыми гарантиями. Тем не менее системы с более сильными гарантиями могут быть привлекательными, поскольку их легче правильно использовать. Рассмотрев несколько моделей согласованности, вы сможете принимать обоснованные решения о том, какое из них лучше всего соответствует вашим потребностям.

Существует некоторое сходство между распределенными моделями согласованности и обсуждавшейся ранее иерархией уровней изоляции в транзакциях [4, 5] (см. раздел 7.2). Однако, несмотря на ряд совпадений, в основном это независимые задачи: изоляция транзакций заключается главным образом в том, чтобы избежать условий гонки из-за конкурентного выполнения транзакций, тогда как основная цель распределенной согласованности — координировать состояние реплик в условиях задержек и сбоев.

Эта глава охватывает широкий круг тем, но, как будет видно позже, на практике они глубоко связаны.

- Мы начнем с рассмотрения одной из самых сильных моделей согласованности общего назначения — *линеаризуемости,* исследуем ее достоинства и недостатки.
- Затем изучим проблему упорядочения событий в распределенной системе (см. раздел 9.3), в частности причинно-следственные связи и единую последовательность.
- В разделе 9.4 рассмотрим, как обеспечить атомарную фиксацию распределенных транзакций, что в итоге приведет нас к решениям проблемы консенсуса.

9.2. Линеаризуемость

В базе данных с конечной согласованностью при направлении одинакового запроса одновременно в две разные реплики можно получить два разных ответа. Это сбивает с толку. Разве не было бы намного проще, если бы база могла создать у пользователя иллюзию, что существует только одна реплика (то есть только одна копия данных)? Тогда у каждого клиента было бы одинаковое представление о данных, и не пришлось бы беспокоиться о задержке репликации.

Эта идея лежит в основе *линеаризуемости* [6] (также известной как *атомарная согласованность* [7], *сильная согласованность*, *непосредственная согласованность* и *внешняя согласованность* [8]). Точное определение линеаризуемости довольно тонкое, и мы рассмотрим его далее в разделе. Основная идея состоит в следующем: система внешне выглядела так, как если бы в ней была только одна копия данных и все операции с ними были атомарными. С такой гарантией, даже при наличии на самом деле множества реплик, приложению не приходится о них беспокоиться.

В линеаризуемой системе сразу же после удачного завершения клиентом записи все клиенты, читающие БД, должны иметь возможность видеть только что написанное значение. Поддержание иллюзии единственной копии данных означает гарантию того, что прочитанное значение является самым последним обновленным значением и не связано с устаревшим кэшем или репликой. Другими словами, линеаризуемость есть *гарантия актуальности*. Чтобы пояснить эту идею, рассмотрим пример нелинеаризуемой системы.

На рис. 9.1 показан пример нелинеазируемого спортивного сайта [9]. Алиса и Боб сидят в одной комнате и смотрят на телефоны, чтобы узнать результат финала Кубка мира по футболу 2014 года. Как только объявлен окончательный результат, Алиса обновляет страницу, видит победителя и взволнованно рассказывает об этом Бобу. Боб недоверчиво *перезагружает* страницу на своем телефоне, но его запрос попадает на реплику базы данных, на которой случилась задержка, и поэтому его телефон показывает, что игра все еще продолжается.

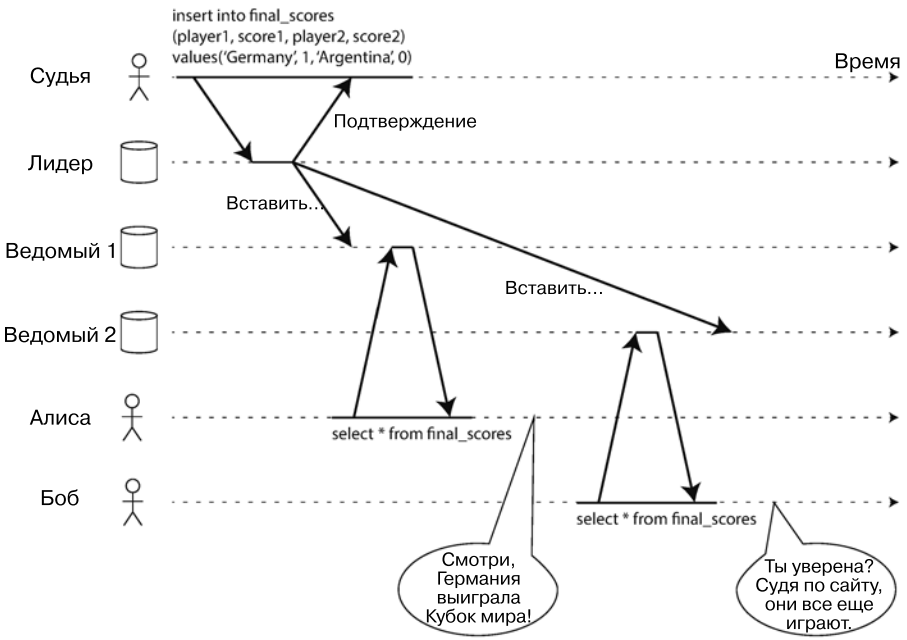


Рис. 9.1. Эта система нелинеаризуема, в результате чего поклонники футбола путаются

Если бы Алиса и Боб одновременно обновили свои страницы, то меньше бы удивились, получив два разных результата запроса, поскольку точно не знали, в какое время их запросы обрабатывались сервером. Но Боб знает, что нажал кнопку перезагрузки (инициировал свой запрос) *после* того, как услышал от Алисы окончательный счет, и поэтому ожидает, что результат его запроса будет по крайней мере столь же свежим, как и у Алисы. Факт возврата старого результата на запрос Боба является нарушением линеаризуемости.

Что делает систему линеаризуемой

В основе линеаризуемости лежит простая идея: снаружи система должна вести себя так, как если бы в ней существовала только одна копия данных. Однако для понимания того, что именно это означает в действительности, нужно немного поразмыслить. Для большего погружения в смысл линеаризуемости рассмотрим еще несколько примеров.

На рис. 9.2 показаны три клиента, конкурентно просматривающие и записывающие один и тот же ключ x в линеаризуемой базе данных. В литературе о распределенных системах x называется реестром — на практике это может быть ключ в хранилище типа «ключ — значение», строка в реляционной БД или документ в базе данных документов.

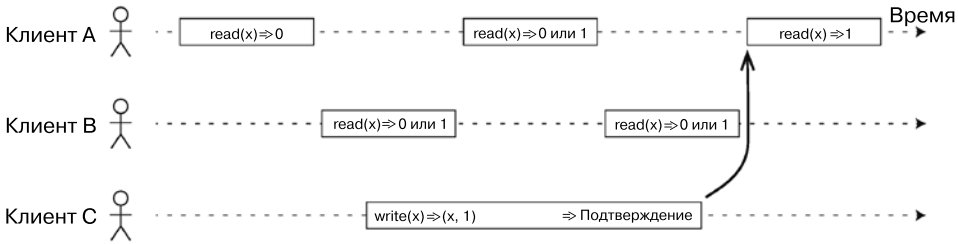


Рис. 9.2. Если запрос чтения осуществляется конкурентно запросу на запись, то он может вернуть либо старое, либо новое значение

Для простоты на рис. 9.2 показаны только запросы с точки зрения клиентов, а не их внутреннее представление в базе данных. Каждая шкала — сделанный клиентом запрос; начало ее соответствует времени, когда клиент отправил запрос, а конец — когда получил ответ. Из-за различных задержек в сети клиент не знает точно, когда база обрабатывала его запрос. Он лишь знает, что это произошло где-то между отправкой запроса и получением ответа[1].

В этом примере реестр выполняет следующие два типа операций:

- *read (x) ⇒ v* означает: клиент попросил прочитать значение реестра *x*, а база данных вернула значение *v*;
- *write (x, v) ⇒ r* означает: клиент запросил присвоить реестру *x* значение *v*, а база данных вернула ответ *r* (что может означать *ok* или *error*).

На рис. 9.2 значение регистра *x* первоначально равно 0. Клиент C выполняет запрос на запись с целью присвоить ему значение 1. Пока это происходит, клиенты A и B несколько раз опросили базу данных, чтобы узнать последнее значение. Какие ответы могут получить A и B для своих запросов на чтение?

- Первая операция чтения клиентом A завершается до начала записи, поэтому она обязательно вернет старое значение 0.
- Последняя операция чтения клиента A начинается после завершения записи, поэтому в случае линеаризуемости база данных обязательно вернет новое значение 1: мы знаем, что фактическая запись в базу должна произойти где-то между началом и концом операции записи, а чтение — где-то между началом и концом

[1] Тонкость этой схемы заключается в том, что она предполагает существование глобальных часов, время которых представлено на горизонтальной оси. В реальных системах обычно нет точных часов (см. раздел 8.3), однако данное предположение нам подходит: для анализа распределенного алгоритма можно сделать допущение, что точные глобальные часы существуют, поскольку алгоритм не имеет к ним доступа [47]. Вместо этого он может видеть только искаженное приближение реального времени, создаваемое кварцевым генератором и NTP.

операции чтения. Если чтение началось после того, как закончилась запись, то оно должно быть обработано после записи, вследствие чего его результатом станет новое только что записанное значение.

- Все операции чтения, которые пересекаются во времени с операцией записи, могут возвращать либо 0, либо 1: мы не можем знать, была ли совершена запись в момент обработки операции чтения. Эти операции происходят *конкурентно* с записью.

Однако для полного описания линеаризуемости этого недостаточно: если операции чтения, конкурентные операции записи, могут возвращать либо старое, либо новое значение, то клиенты будут видеть значение, которое несколько раз меняется со старого на новое и обратно, пока длится запись. Система, эмулирующая «единственную копию данных», так работать не должна[1].

Чтобы сделать систему линеаризуемой, нужно добавить еще одно ограничение, показанное на рис. 9.3.

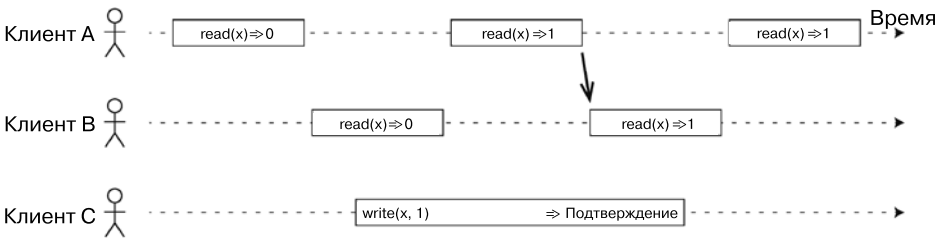


Рис. 9.3. После того как операция чтения вернет новое значение, все последующие операции чтения (для этого и других клиентов) также должны возвращать новое значение

В линеаризуемой системе предполагается, что существует некий момент времени (между началом и концом операции записи), при котором значение *x* атомарно меняется с 0 на 1. Таким образом, если для одного клиента операция чтения вернула новое значение 1, то все последующие операции чтения тоже должны возвращать новое значение, даже если операция записи еще не завершена.

На рис 9.3 эта зависимость от хронометража проиллюстрирована стрелкой. Клиент A первым прочитал новое значение: 1. Сразу после него клиент B начинает новое чтение. Поскольку чтение B происходит сразу после чтения A, оно также должно возвращать 1, даже если запись C продолжается. (Именно в такой ситуации оказались Алиса и Боб на рис 9.1: после того как Алиса прочитала новое значение, Боб тоже ожидает, что прочитает новое значение.)

[1] Реестр, в котором чтение может возвращать либо старое, либо новое значение, если в это время производится запись, называют обычным реестром [7, 25].

Мы можем внести в эту временную схему дополнительные уточнения, чтобы визуализировать каждую операцию, выполняемую атомарно в определенный момент времени. На рис. 9.4 показан более сложный пример [10].

На рис. 9.4, кроме чтения и записи, добавлен третий тип операций: *cas (x, v old, v new)* $\Rightarrow r$ означает, что клиент запросил операцию атомарного *сравнения с изменением значения* (см. пункт «Сравнение с обменом» подраздела «Предотвращение потери обновлений» раздела 7.2). Если текущее значение регистра x равно *v old*, то оно должно быть атомарно изменено на *v new*. Если же $x \neq v\ old$, то значение реестра не изменится, а операция вернет ошибку. r — ответ базы данных (*ok* или *error*).

Каждая операция на рис. 9.4 отмечена вертикальной линией (на шкале каждой операции) в тот момент, когда предполагается, что операция выполнена. Эти маркеры объединены в последовательность, которая должна быть допустимой последовательностью чтения и записи реестра (каждое чтение должно возвращать значение, установленное при последней записи).

Требование линеаризуемости состоит в том, что линии, соединяющие маркеры операций, всегда движутся вперед во времени (слева направо), а не назад. Это требование предоставляет гарантию возврата, упоминавшуюся ранее: после того как новое значение записано или прочитано, все последующие операции чтения выдают то значение, которое было записано, до тех пор пока оно не будет перезаписано снова.

На рис. 9.4 показано несколько интересных моментов.

- Первый клиент, B, отправил запрос на чтение x, затем клиент D отправил запрос на присвоение x значения 0, а позже клиент A отправил запрос на присвоение x значения 1. Тем не менее при чтении клиент B получил значение 1 (то, что было записано A). Это нормально: база данных сначала обработала запись D, затем запись A и, наконец, чтение B. Описанная последовательность не является той, в которой были отправлены запросы, но она приемлема, поскольку три запроса конкурентны. Возможно, запрос чтения B немного задержался в сети, поэтому он добрался до базы только после двух операций записи.
- Операция чтения клиента B вернула 1 до того, как клиент A получил ответ от базы данных, в котором сообщалось об успешной записи значения 1. Здесь тоже все в порядке: это не говорит о том, что значение было прочитано до того, как было записано; а лишь означает незначительную задержку в сети ответа *ok* от базы данных для клиента A.
- Эта модель не предполагает изоляции транзакций: другой клиент способен изменить значение в любое время. Например, клиент C сначала читает 1, а затем читает 2, так как между двумя операциями чтения это значение было изменено клиентом B. Операция атомарного сравнения с присвоением *(cas)* может быть использована для проверки того, что значение не было конкурентно изменено

другим клиентом: запросы *cas клиентов* B и C были успешны, но запрос *cas* клиента D вернул ошибку (в тот момент, когда база данных его обрабатывала, значение x было больше 0).

- Последняя операция чтения клиента B (в затененной полосе) нелинеаризуема. Эта операция конкурентна записи *cas* клиента C, которая обновляет x с 2 до 4. В отсутствие других запросов было бы хорошо, если бы чтение B вернуло 2. Однако до начала чтения B клиент A уже прочитал новое значение, равное 4, вследствие чего B не может получить более старое значение, чем A. Описанная ситуация аналогична той, что у Алисы и Боба на рис. 9.1.

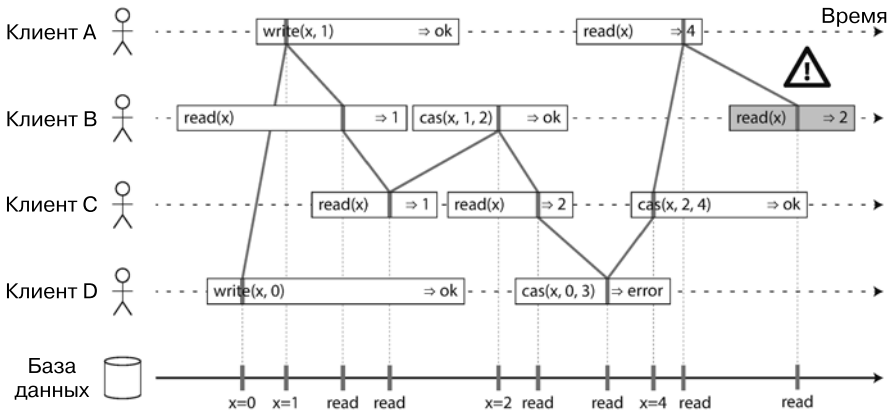


Рис. 9.4. Визуализация временных точек, в которых должны вступить в силу операции чтения и записи. Последняя операция чтения B нелинеаризуема

Это наглядное представление линеаризуемости; ее формальное определение [6] дает более точное описание. Можно проверить (хотя и понадобятся большие вычислительные затраты), является ли поведение системы линеаризуемым, записывая время всех запросов и ответов и проверяя, можно ли расположить их в правильной логической последовательности [11].

Линеаризуемость и последовательность

Линеаризуемость часто путают с сериализуемостью (см. раздел 7.3), поскольку кажется, что оба слова означают нечто вроде «можно упорядочить в виде последовательности». Однако они представляют собой две совершенно разные гарантии, и их важно различать.

- *Сериализуемость* — свойство изоляции *транзакций*, при котором каждая транзакция может читать и записывать несколько объектов (строки, документы, записи); см. подраздел «Однообъектные и многообъектные операции» раздела 7.1. Это гарантирует, что транзакции ведут себя так же, как если бы выполнялись

в некой последовательности (каждая из них завершается до начала следующей). Отличие такого порядка запросов от порядка, в котором транзакции фактически выполнялись, считается нормальным [12].

- *Линеаризуемость* — гарантия возврата при чтении и записи реестра *(отдельного объекта)*. Она не объединяет операции в транзакции, поэтому не предотвращает таких проблем, как асимметрия записи (см. подраздел «Асимметрия записи и фантомы» раздела 7.2), если не предпринять дополнительные меры, такие как материализация конфликтов (см. одноименный пункт подраздела «Асимметрия записи и фантомы» раздела 7.2).

База данных может обеспечивать как сериализуемость, так и линеаризуемость. Это сочетание известно как *строгая сериализуемость* или *сильная сериализуемость* с *одной копией* (1SR-сильная) [4, 13]. Реализации сериализуемости на основе двухфазной блокировки (см. подраздел «Двухфазная блокировка (2PL)» раздела 7.3) или фактическое последовательное выполнение (см. подраздел «По-настоящему последовательное выполнение» раздела 7.3), как правило, линеаризуемы.

Однако сериализованная изоляция снимков состояния (см. подраздел «Сериализуемая изоляция снимков состояния (SSI)» раздела 7.3) нелинеаризуема: по своей архитектуре она выполняет чтение из последовательного снимка состояния, чтобы избежать блокировки между запросами чтения и записи. Весь смысл согласованного снимка состояния заключается в следующем: в него не включены записи, которые являются более свежими, чем сам снимок, и, таким образом, чтение снимка не является линеаризуемым.

Опора на линеаризуемость

В каких случаях полезна линеаризуемость? Просмотр окончательного результата спортивного матча, пожалуй, чересчур легкомысленный пример: результат, устаревший на несколько секунд, вряд ли причинит какой-либо реальный вред в этой ситуации. Однако есть несколько областей, в которых линеаризуемость является важным требованием для правильной работы системы.

Блокировка и выбор ведущего узла

Система, использующая репликацию с одним ведущим узлом, должна гарантировать, что тот действительно только один, а не несколько (разделение интеллекта). Один из способов выбрать ведущий узел — применить блокировку: каждый запускаемый узел пытается получить блокировку, и тот, которому это удастся, становится ведущим [14]. Независимо от того, как реализована блокировка, она должна быть линеаризуемой: необходимо соглашение между узлами о том, какому узлу принадлежит блокировка, иначе это бесполезно.

Для реализации распределенной блокировки и выбора ведущего узла часто используются такие координационные сервисы, как Apache ZooKeeper [15] и etcd [16].

В них применяются консенсусные алгоритмы для отказоустойчивой реализации линеаризуемых операций (мы обсудим такие алгоритмы далее, в подразделе «Отказоустойчивый консенсус» раздела 9.4)[1]. Существует еще много тонких нюансов правильной реализации блокировок и выбора ведущего узла (например, проблема ограждения, описанная в пункте «Ведущий узел и блокировки» подраздела «Истина определяется большинством» раздела 8.4). Библиотеки, такие как Apache Curator [17], помогают в этом, предоставляя более высокоуровневые решения поверх ZooKeeper. Однако сервис линеаризуемого хранения является главной основой для этих задач координации.

Распределенные блокировки также применяются на гораздо более низком уровне в некоторых распределенных базах данных, таких как Oracle Real Application Clusters (RAC) [18]. В RAC используются блокировки страниц диска, притом несколько узлов совместно обращаются к одной дисковой системе хранения. Поскольку эти линеаризуемые блокировки находятся на критически важном пути выполнения транзакций, в системе RAC для коммуникации между узлами БД обычно задействовано кластерное соединение.

Ограничения и гарантии уникальности

Ограничения уникальности широко распространены в базах данных: например, имя пользователя или адрес электронной почты должны однозначно его идентифицировать, а в сервисе хранения файлов не может быть двух файлов с одинаковыми путем и именем. Чтобы принудительно применить данное ограничение при записи данных, нужна линеаризуемость (например, когда два человека пытаются конкурентно создать пользователя или файл с одинаковыми именами, одна из этих операций завершится с ошибкой).

В действительности эта ситуация похожа на блокировку: когда пользователь регистрируется в сервисе, можно представить, что он получает «блокировку» на свое имя. Такая операция очень похожа на атомарное сравнение с присвоением: пользователь делает имя своим идентификатором при условии, что оно еще не занято.

Подобные проблемы возникают, если нужно гарантировать, что баланс банковского счета никогда не станет отрицательным, или что не будет продано больше товаров, чем есть в наличии на складе, или что два человека не забронируют одно и то же место в самолете или в театре. Все эти ограничения требуют одного обновленного

[1] Строго говоря, ZooKeeper и etcd обеспечивают линеаризуемую запись, но результаты операций чтения могут быть устаревшими, поскольку по умолчанию их способна обслуживать любая из реплик. Можно дополнительно запросить линеаризуемое чтение: в etcd оно называется чтением кворума [16], а в ZooKeeper перед операцией чтения нужно вызвать sync() [15]; см. также пункт «Реализация линеаризуемого хранилища с помощью рассылки общей последовательности» подраздела «Рассылка общей последовательности» раздела 9.3.

значения (баланс на счету, количество товаров, занятые места), согласованного для всех узлов.

В реальных приложениях иногда бывает приемлемым ослабить требования к таким ограничениям (например, если на рейс забронировано больше мест, чем есть в самолете, то можно перевести таких клиентов на другой рейс, предложив им компенсацию за неудобства). В таких случаях линеаризуемость может не понадобиться. Мы обсудим такие слабые ограничения в подразделе «Своевременность и целостность» раздела 12.3.

Однако жесткое ограничение уникальности, какое обычно используется в реляционных базах данных, требует линеаризуемости. Другие типы ограничений, такие как внешние ключи или ограничения атрибутов, могут быть реализованы без необходимости линеаризуемости [19].

Межканальные синхронизационные зависимости

Обратите внимание на одну деталь на рис. 9.1: если бы Алиса не сказала о счете, то Боб не узнал бы, что результат его запроса был устаревшим. Он бы просто обновил страницу через несколько секунд и увидел бы окончательный результат. Нарушение линеаризуемости было замечено только благодаря появлению в системе дополнительного канала связи (голос Алисы достиг ушей Боба).

Подобные ситуации возникают и в компьютерных системах. Предположим, есть сайт, на который пользователи могут загружать фотографии, а фоновый процесс уменьшает размер изображения, чтобы снизить разрешение и ускорить загрузку миниатюры. Архитектура и поток данных в этой системе показаны на рис. 9.5.

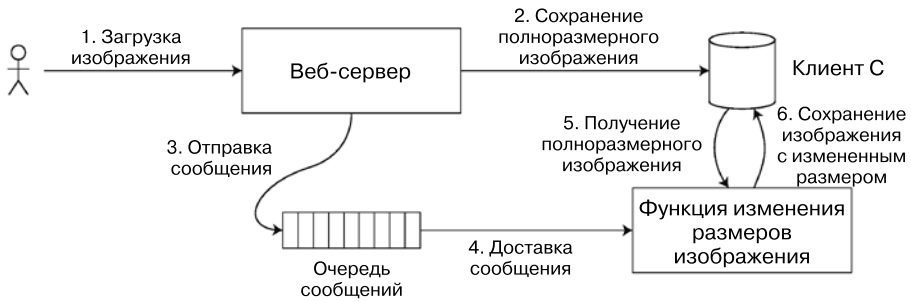


Рис. 9.5. Веб-сервер и функция изменения размера изображений взаимодействуют и через хранилище файлов, и через очередь сообщений, создавая возможности для гонки

Чтобы изменить размер фото, необходимо явно выполнить соответствующую задачу. Данная инструкция будет отправлена с веб-сервера в функцию изменения размера через очередь сообщений (см. главу 11). Веб-сервер размещает в очереди не саму фотографию, так как большинство брокеров сообщений предназначены

для небольших сообщений, а размер изображения может достигать нескольких мегабайт. Вместо этого оно сначала записывается в сервис хранения файлов, и после завершения записи инструкция для функции изменения размера помещается в очередь.

Если сервис хранения файлов линеаризуем, то данная система должна работать правильно. Но в противном случае существует риск гонки: очередь сообщений (прохождение этапов 3 и 4 на рис. 9.5) может оказаться быстрее, чем внутренняя репликация в сервисе хранения. В этом случае, когда функция изменения размера выберет изображение (этап 5), она увидит его старую версию или вообще ничего. При обработке старого изображения полноразмерная и уменьшенная версии в хранилище файлов больше не будут соответствовать друг другу.

Данная проблема возникает из-за того, что между веб-сервером и функцией изменения размера есть два канала связи: хранилище файлов и очередь сообщений. Без гарантии получения самых последних версий изображений, предоставляемой линеаризуемостью, возможны условия гонки между этими двумя каналами. Такая ситуация аналогична показанной на рис. 9.1, где также было состояние гонки между двумя каналами связи: репликацией БД и реальным звуковым каналом между ртом Алисы и ушами Боба.

Линеаризуемость — не единственный способ избежать состояния гонки, но его проще всего понять. Если дополнительный канал связи находится под вашим контролем (например, что касается очереди сообщений, но не в случае с Алисой и Бобом), то можно использовать альтернативные подходы, аналогичные тому, что обсуждался в подразделе «Читаем свои же записи» раздела 5.2, однако это потребует дополнительного усложнения системы.

Реализация линеаризуемых систем

Теперь, рассмотрев несколько примеров полезного применения линеаризуемости, подумаем о том, как мы можно реализовать систему с линеаризуемой семантикой.

Поскольку линеаризуемость, в сущности, означает «вести себя так, как будто существует только одна копия данных и все операции над ней являются атомарными», самым простым ответом было бы действительно использовать только одну такую копию. Однако этот вариант не позволяет справляться с отказами: если узел, содержащий единственную копию данных, претерпел сбой, то они будут потеряны или по крайней мере недоступны до тех пор, пока узел не заработает снова.

Наиболее распространенным методом обеспечения отказоустойчивости системы является использование репликации. Вспомним методы репликации, описанные в главе 5, и сравним их линеаризуемость.

- *Репликация с одним ведущим узлом (потенциально линеаризуемая).* В системе с репликацией с одним ведущим узлом (см. раздел 5.1) этот узел хранит основную

копию данных, которые используются для записи, а ведомые — резервные копии в других узлах. Если операции чтения обслуживаются ведущим узлом или синхронно обновляемыми ведомыми, то они могут быть линеаризуемыми[1]. Однако на практике не каждая база данных с одним ведущим узлом является линеаризуемой — либо по архитектуре (например, потому, что в ней применяется изоляция снимков состояния), либо из-за ошибок, возникающих вследствие протекания конкурентных процессов [10].

Использование ведущего узла для чтения основывается на предположении, что точно известно, какой именно узел является таковым. Как обсуждалось в подразделе «Истина определяется большинством» раздела 8.4, вполне вероятно, что узел может считать себя ведущим, в то время как на самом деле это не так — и если ложный ведущий продолжит обслуживать запросы, то, скорее всего, линеаризуемость нарушится [20]. При асинхронной репликации ошибка может распространиться на другие ресурсы и даже привести к потере зафиксированной записи (см. подраздел «Перебои в обслуживании узлов» раздела 5.1). Это нарушает как долговечность, так и линеаризуемость.

- *Консенсусные алгоритмы (линеаризуемые).* Отдельные консенсусные алгоритмы, которые будут рассмотрены далее в этой главе, имеют сходство с репликацией с одним ведущим узлом. Однако консенсусные протоколы содержат меры, позволяющие предотвратить разделение интеллекта и использование устаревших реплик. Благодаря этим нюансам консенсусные алгоритмы дают возможность безопасно реализовать линеаризуемое хранилище. Так работают ZooKeeper [21] и etcd [22].
- *Репликация с несколькими ведущими узлами (нелинеаризуемая).* Системы с репликацией с несколькими ведущими узлами обычно не являются линеаризуемыми, поскольку конкурентно обрабатывают записи на нескольких узлах и асинхронно реплицируют их на другие узлы. По этой причине в них могут появляться конфликты записи, требующие разрешения (см. подраздел «Обработка конфликтов записи» раздела 5.3). Такие конфликты являются следствием отсутствия единой копии данных.
- *Репликация без ведущего узла (не всегда линеаризуемая).* О системах с репликацией без ведущего узла (стиль Dynamo, см. раздел 5.4) иногда говорят, что в них можно получить «сильную согласованность», если потребовать чтение и запись кворума ($w + r > n$). Но это не всегда так — все зависит от точной конфигурации кворумов и от того, как определяется сильная согласованность.
- Методы разрешения конфликтов по типу «выигрывает последний», основанные на часах времени суток (например, в Cassandra, см. подраздел «Ненадежность синхронизированных часов» раздела 8.3), почти наверняка являются нелинеаризуемыми, поскольку метки времени таких часов не могут быть гарантирован-

[1] Секционирование (шардинг данных) в базе данных с одним ведущим узлом, так что в каждом разделе есть свой ведущий узел, не влияет на линеаризуемость, речь идет только об одном объекте. Другое дело — межсетевые транзакции (см. раздел 9.4).

но согласованными с фактической последовательностью событий из-за сдвига часов. Нестрогие кворумы (см. подраздел «Нестрогие кворумы и направленная передача» раздела 5.4) также сводят вероятность линеаризуемости к нулю. Даже при строгих кворумах возможно нелинеаризуемое поведение, как будет показано ниже.

Линеаризуемость и кворумы. Интуитивно кажется, что операции чтения и записи при строгом кворуме в модели типа Dynamo должны быть линеаризуемыми. Однако с учетом разновременных задержек в сети возможны условия гонки, как показано на рис. 9.6.

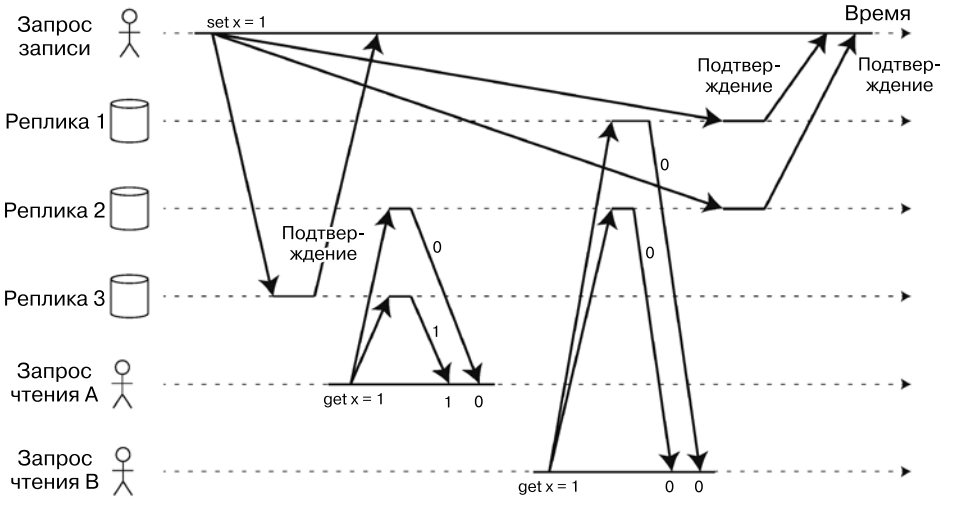


Рис. 9.6. Несмотря на строгий кворум, операции выполняются нелинеаризуемо

На рис. 9.6 начальное значение x равно 0 и клиент, инициирующий запись, обновляет x до 1, отправив запрос записи на все три реплики ($n = 3, w = 3$). В это же время клиент A считывает данные из кворума, составляющего два узла ($r = 2$), и видит новое значение 1 на одном из них. Кроме того, одновременно с записью клиент B считывает данные из другого кворума, также состоящего из двух узлов, и получает из обоих старое значение 0.

Условие кворума выполнено ($w + r > n$), но эта операция тем не менее нелинеаризуема: запрос клиента B начинается после запроса клиента A, но клиент B получает старое значение, а клиент A — новое. (Все та же ситуация с Алисой и Бобом из примера на рис. 9.1.)

Интересно отметить, что кворумы в стиле Dynamo *можно* линеаризовать за счет снижения производительности: операция чтения должна выполнить синхронное разрешение конфликта чтения (см. пункт «Разрешение конфликтов при чтении

и антиэнтропия» подраздела «Запись в базу данных при отказе одного из узлов» раздела 5.4), прежде чем возвращать результаты приложению [23], а операция записи — прочитать последнее состояние кворума узлов перед записью данных [24, 25]. Однако в Riak синхронное разрешение конфликта чтения не выполняется, чтобы не снижать производительность [26]. В Cassandra *действительно* выполняется разрешение конфликта чтения перед завершением чтения кворума [27], но в случае нескольких конкурентных операций записи с одним ключом линеаризуемость теряется из-за разрешения конфликтов методом «выигрывает последний».

Более того, таким способом могут быть реализованы только линеаризуемые операции чтения и записи, но не линеаризуемая операция сравнения с изменением, поскольку для этого требуется консенсусный алгоритм [28].

Таким образом, наиболее безопасно будет предположить, что Dynamo-система без ведущего узла не обеспечивает линеаризуемость.

Цена линеаризуемости

Поскольку одни методы репликации обеспечивают линеаризуемость, а другие — нет, интересно исследовать более подробно преимущества и недостатки линеаризуемости.

В главе 5 мы уже обсуждали некоторые варианты использования различных методов репликации; например, увидели, что репликация с несколькими ведущими узлами — часто хороший выбор для случая с несколькими ЦОДами (см. пункт «Эксплуатация с несколькими ЦОДами» подраздела «Сценарии использования репликации с несколькими ведущими узлами» раздела 5.3). Пример такой системы показан на рис. 9.7.

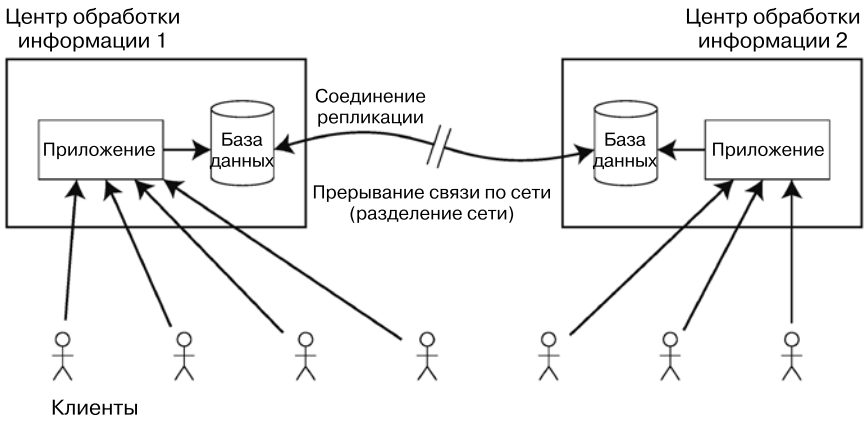


Рис. 9.7. При нарушении связи в сети приходится делать выбор между линеаризуемостью и доступностью

Что произойдет, если прервется связь по сети между двумя ЦОДами? Предположим, что сеть внутри каждого центра работает и связь с клиентами тоже есть, но центры не могут соединяться друг с другом.

Благодаря базе данных с несколькими ведущими узлами каждый ЦОД может продолжать нормальную работу: поскольку записи из одного центра асинхронно реплицируются на другой, они просто будут поставлены в очередь и переданы, как только сетевое подключение восстановится.

Однако если используется репликация с одним ведущим узлом, то такой узел должен находиться в одном из ЦОДов. Все записи и все линеаризуемые операции чтения должны отправляться ведущему узлу, поэтому запросы чтения и записи, поступающие от клиентов, которые подключены к ведомому ЦОДу, должны синхронно по сети направляться в ведущий центр.

При нарушении сетевого соединения между центрами в случае репликации с одним ведущим узлом клиенты, подключенные к ведомым центрам обработки информации, не могут связаться с ведущим узлом. Они не смогут делать записи в базу данных или линеаризуемые операции чтения. Они все еще смогут читать сообщения от ведомого центра, но эти сообщения могут быть устаревшими (нелинеаризуемыми). Если приложение требует линеаризуемого чтения и записи, то разрыв сетевого соединения приводит к тому, что приложение станет недоступным в тех ЦОДах, которые не могут связаться с ведущим узлом.

Если клиенты могут напрямую подключаться к тому ЦОДу, который является ведущим, то это не проблема, поскольку там приложение продолжит нормально функционировать. Но клиенты, которым доступен только ведомый, будут испытывать перебои в обслуживании до тех пор, пока не будет восстановлена связь по сети.

Теорема CAP

Эта проблема касается не только репликации с одним или несколькими ведущими узлами, но и любой линеаризуемой базы данных, независимо от того, как именно она реализована. Кроме того, проблема имеет отношение не только к системе с несколькими ЦОДами — она может возникать в любой ненадежной сети даже внутри одного такого центра. Компромисс выглядит следующим образом[1].

- Если приложение *требует* линеаризуемости, а некоторые реплики не имеют доступа к другим репликам из-за проблем в сети, то эти реплики не могут обрабатывать запросы, пока они отключены: они должны либо ждать, пока связь будет восстановлена, либо возвращать ошибку (в любом случае они становятся *недоступными*).

[1] Эти два варианта иногда называют CP (согласованными, но недоступными в сетевых разделах) и AP (доступными, но несогласованными в сетевых разделах) соответственно. Однако у данной классификации есть ряд недостатков [9], так что ее лучше избегать.

- Если приложение *не требует* линеаризуемости, то оно может быть записано таким образом, чтобы каждая реплика могла обрабатывать запросы независимо, даже будучи отключенной от других реплик (например, в конфигурации с несколькими ведущими узлами). Тогда приложение может оставаться доступным даже в случае проблем с сетью, но его поведение нелинеаризуемо.

Таким образом, приложения, не требующие линеаризуемости, более устойчивы к неполадкам в сети. Это представление широко известно как *теорема CAP* [29–32], названная так Эриком Брюером (Eric Brewer) в 2000 году, хотя данный компромисс был известен разработчикам распределенных БД еще с 1970-х годов [33–36].

Первоначально принцип CAP предлагался как эмпирическое правило, без точных определений, с целью начать обсуждение компромиссов в базах данных. В то время многие распределенные БД стремились к предоставлению линеаризуемой семантики на кластере машин с общим хранилищем [18]. Принцип CAP побуждал инженеров БД искать другие варианты архитектуры распределенных систем с общим доступом, которые бы лучше подходили для реализации широкомасштабных веб-сервисов [37]. CAP заслуживает признания за этот шаг вперед в культуре разработки — начиная с середины 2000-х годов мы становимся свидетелями взрыва новых технологий в области баз данных (известных как NoSQL).

Бесполезная теорема CAP

Аббревиатура CAP иногда расшифровывается как *Consistency, Availability, Partition tolerance: pick 2 out of 3* — «Согласованность, доступность, устойчивость к нарушению связности сети: выберите 2 из 3». К сожалению, в действительности все не так [32]. Поскольку нарушение связности сети является своего рода ошибкой, у нас, в сущности, нет выбора: нарушение связности будет, нравится это или нет [38].

В тех случаях, когда сеть работает правильно, система может обеспечивать и согласованность (линеаризуемость), и доступность из любой точки. При возникновении сбоя в сети приходится выбирать либо линеаризуемость, либо доступность. Таким образом, более удачным вариантом расшифровки CAP будет *either Consistent or Available when Partitioned* — «либо последовательность, либо доступность при нарушении связности сети» [39]. В более надежных сетях этот выбор приходится делать реже, но в какой-то момент он все равно неизбежен.

В обсуждениях, касающихся CAP, есть несколько противоречивых определений термина *«доступность»*, а формализация как теорема [30] не соответствует ее обычному значению [40]. Многие так называемые высокодоступные (отказоустойчивые) системы фактически не соответствуют специфическому определению доступности CAP. В целом вокруг нее много недоразумений и путаницы, и это не упрощает понимание систем, так что лучше ее избегать.

Теорема CAP как формальное определение [30] имеет очень узкую область применения: она рассматривает только одну модель согласованности (линеаризуемость)

и один вид сбоя (*нарушение связности сети*[1] или узлы, которые работоспособны, но отключены друг от друга). В ней ничего не говорится о задержках в сети, мертвых узлах и других компромиссах. Таким образом, хотя теорема CAP имеет историческое значение, она малоприменима на практике при проектировании систем [9, 40].

В распределенных системах есть еще много интересных доказательств нереализуемости [41]. Сегодня на смену CAP пришли более точные доказательства [2, 42], вследствие чего в настоящее время эта теорема представляет главным образом исторический интерес.

Линеаризуемость и сетевые задержки

Хотя линеаризуемость и является полезной гарантией, на практике удивительно мало систем являются линеаризуемыми. Например, даже оперативная память современного многоядерного процессора нелинеаризуема [43]: если поток, работающий на одном ядре CPU, записывает значение в ячейку памяти, а поток на другом ядре сразу же считывает это значение, то нет гарантии, что он прочитает значение, записанное первым потоком (если только не используется *барьер* или *защита памяти* [44]).

Причиной такого поведения является то, что у каждого ядра CPU есть собственные кэш памяти и буфер хранения. Данные, направляемые в память, сначала по умолчанию попадают в кэш, и все изменения записываются в основную память асинхронно. Поскольку доступ к данным в кэше происходит намного быстрее, чем доступ к основной памяти [45], эта функция обеспечивает высокую производительность современных процессоров. Однако если имеется несколько копий данных (одна в основной памяти и, возможно, еще несколько в различных кэшах) и эти копии асинхронно обновляются, то линеаризуемость теряется.

Зачем нужен этот компромисс? Нет смысла использовать теорему CAP для обоснования многоядерной модели согласованности памяти: обычно предполагается, что внутри одного компьютера связь надежна и едва ли можно ожидать продолжения нормальной работы одного ядра процессора, если оно отключено от остальной части компьютера. Причиной снижения линеаризуемости в данном случае является не отказоустойчивость, а *производительность*.

То же самое относится ко многим распределенным базам данных, которые предпочитают не предоставлять гарантии линеаризуемости: они делают это прежде всего

[1] Как показано в подразделе «Сетевые сбои на практике» раздела 8.2, в этой книге секционированием (partitioning) называется намеренное разделение большого набора данных на более мелкие (сегментирование, см. главу 6). Сетевое разделение, или нарушение связности сети (network partition), напротив, представляет собой тип сетевого сбоя, который обычно не рассматривается отдельно от других видов сбоев. Однако, поскольку он касается буквы P в слове CAP, мы будем стараться избежать путаницы.

для повышения производительности, а не для отказоустойчивости [46]. Линеаризуемость медленная, и это верно всегда, а не только во время сетевых сбоев.

Разве нельзя найти более эффективную реализацию линеаризуемого хранилища? Похоже, нет: Х. Аттия (H. Attiya) и Дж. Уэлч (J. Welch) [47] доказали, что если нужно реализовать линеаризуемость, то время отклика запросов на чтение и запись будет по крайней мере пропорционально времени неопределенных сетевых задержек. В сети с очень разновременными задержками, как и в большинстве компьютерных сетей (см. подраздел «Время ожидания и неограниченные задержки» раздела 8.2), время отклика для линеаризуемых операций чтения и записи неизбежно будет высоким. Более быстрый алгоритм для линеаризуемости не существует, но более слабые модели согласованности могут оказаться гораздо быстрее, и этот компромисс важен для систем, чувствительных к задержкам. В главе 12 мы обсудим некоторые методы, позволяющие избежать линеаризуемости, не жертвуя корректностью.

9.3. Гарантии упорядоченности

Как уже отмечалось, линеаризуемый реестр ведет себя так, как если бы существовала только одна копия данных, а каждая операция выглядела бы атомарной в любой момент времени. Это определение подразумевает, что операции выполняются в определенной последовательности. Она была показана на рис. 9.4, где операции были представлены в том порядке, в котором должны были выполняться.

В нашей книге постоянно обсуждается сохранение последовательности; таким образом подразумевается, что это важная фундаментальная идея. Перечислю вкратце ряд других областей, в которых обсуждалась последовательность.

- Из главы 5 мы узнали, что главная цель ведущего узла в репликации с одним таким узлом — определить *последовательность операций записи* в журнале репликации, то есть порядок, в котором ведомые узлы будут выполнять эти операции. При отсутствии единого ведущего узла возможны конфликты из-за конкурентных операций (см. подраздел «Обработка конфликтов записи» раздела 5.3).
- Сериализуемость, описанная в главе 7, заключается в обеспечении того, чтобы транзакции вели себя так, как если бы они выполнялись в *некой последовательности*. Этого можно достичь, действительно выполняя транзакции в таком порядке либо допуская конкурентное выполнение, но предотвращая конфликты сериализации (путем блокировки или прерывания).
- Использование временных меток и часов в распределенных системах, о которых говорилось в главе 8 (см. подраздел «Ненадежность синхронизированных часов» раздела 8.3), — еще одна попытка внести порядок в этот беспорядочный мир, например, чтобы определить, какая из двух записей произошла позже.

Оказывается, существуют глубокие взаимосвязи между упорядочением, линеаризуемостью и консенсусом. Хотя эта идея немного более теоретическая и абстрактная,

чем материал, изложенный в остальной части данной книги, она очень полезна для лучшего понимания того, что системы могут и чего не могут сделать. Мы рассмотрим указанную тему ниже.

Порядок и причинность

Есть несколько причин, почему значение сохранения правильной последовательности продолжает расти, и одна из них такова: это помогает сохранить *причинность*. В книге нам уже встретилось несколько примеров, где важна причинность.

- В подразделе «Согласованное префиксное чтение» раздела 5.2 на рис. 5.5 был представлен пример, когда участник чата сначала увидел ответ на вопрос, а затем сам вопрос, на который и был дан ответ. Это сбивает с толку, поскольку нарушает наши представления о причинах и следствиях: если на вопрос уже ответили, то ясно, что вопрос должен был следовать первым, так как человек, дающий ответ, должен был видеть вопрос (если только он не является экстрасенсом и не способен предвидеть будущее). Между вопросом и ответом существует *причинно-следственная зависимость*.
- Аналогичная ситуация была представлена на рис. 5.9, где мы рассмотрели репликацию между тремя ведущими узлами и заметили, что некоторые операции записи могут «обгонять» другие из-за сетевых задержек. С точки зрения одной из реплик это выглядело как обновление несуществующей строки. Причинность здесь означает, что вначале необходимо создать строку и только потом ее можно будет обновить.
- В подразделе «Обнаружение конкурентных операций записи» раздела 5.4 мы заметили, что для двух операций A и B существует три возможности: либо A произошла раньше, чем B, либо B произошла раньше, чем A, либо A и B являются конкурентными. Эта связь типа *«происходит до»* является еще одним выражением причинности: если операция A произошла раньше, чем B, то это значит, что операция B могла знать об A, или быть основанной на ней, или зависеть от нее. Если же A и B конкурентны, то между ними нет причинно-следственной связи; другими словами, мы уверены: ни одна из этих операций не знает о существовании другой.
- В контексте изоляции снимков состояний для транзакций (см. подраздел «Изоляция снимков состояний и воспроизводимое чтение» раздела 7.2) говорилось: транзакция считывается из согласованного снимка состояния. Но что есть «согласованный» в данном контексте? Это означает *«согласованный с причинностью»*: если в снимке содержится ответ, то там должен содержаться и вопрос, на который и был дан ответ [48]. Снимок всей базы данных в конкретный момент времени делает ее согласованной с причинностью: результаты всех операций, произошедших до этого момента, видны, но действия, совершенные после создания снимка, отсутствуют. Асимметрия чтения (неповторяемые чтения, см. рис. 7.6) означает чтение данных в состоянии, нарушающем причинность.

- В примерах асимметрии записи между транзакциями (см. подраздел «Асимметрия записи и фантомы» раздела 7.2) также продемонстрированы причинно-следственные зависимости: на рис. 7.8 Алиса могла отказаться от вызова, поскольку транзакция посчитала, что Боб все еще звонит, и наоборот. В этом случае действие исходящего вызова зависит от наблюдения за тем, кто в данный момент звонит. Сериализуемая изоляция снимков состояния (см. одноименный подраздел раздела 7.3) позволяет обнаружить искажение, отслеживая причинно-следственные зависимости между транзакциями.
- В примере, где Алиса и Боб смотрят футбол (см. рис. 9.1), тот факт, что Боб получил с сервера старые данные о том, что матч все еще идет, после того как услышал от Алисы результат завершившегося матча, является нарушением причинности: восклицание Алисы причинно зависит от объявления счета, поэтому Боб должен был бы также увидеть счет после того, как услышал Алису. Такая же картина проявилась в пункте «Межканальные синхронизационные зависимости» подраздела «Опора на линеаризуемость» раздела 9.2 в примере с сервисом изменения размера изображения.

Причинность требует расположения событий в определенном порядке: причина должна идти раньше результата; сообщение отправляется до того, как было получено; вопрос приходит перед ответом. Как и в реальной жизни, одно вытекает из другого: один узел читает некие данные, а затем записывает результат, другой узел читает записанный результат и, в свою очередь, что-то пишет и т. д. Эти цепочки причинно-зависимых операций определяют причинно-следственный порядок в системе — что перед чем произошло.

В случае подчинения системы последовательности, соответствующей причинно-следственным связям, говорят, что она *причинно-согласованна*. Например, изоляция снимков состояния обеспечивает причинную согласованность: если при чтении БД вы видите какую-то часть данных, то должны иметь возможность видеть и любые данные, которые причинно предшествуют этим (при условии, что они не были удалены ранее).

Причинно-следственное упорядочение — это не полное упорядочение

Полное упорядочение позволяет сравнить два произвольных элемента и в любой момент сказать, какой из них больше, а какой — меньше. Например, натуральные числа полностью упорядочены: если я задам вам два числа, скажем 5 и 13, то вы можете ответить, что 13 больше 5.

Однако математические множества не полностью упорядочены: что больше — $\{a, b\}$ или $\{b, c\}$? Их нельзя сравнивать, поскольку ни одно из них не является подмножеством другого. Мы говорим, что эти множества *несравнимы* и, следовательно, математические множества *частично упорядочены*: в некоторых случаях одно множество больше другого (если одно из них содержит все элементы другого), в других случаях они несравнимы.

Разница между полным и частичным упорядочением отражается в разных моделях согласованности баз данных.

- *Линеаризуемость*. В линеаризуемой системе имеет место *полное упорядочение* операций: если система ведет себя так, как будто в ней существует только одна копия данных и каждая операция является атомарной, то это означает, что для любых двух операций всегда можно сказать, какая из них произошла первой. Такое полное упорядочение показано в виде временной шкалы на рис. 9.4.
- *Причинность*. Как уже говорилось, две операции являются конкурентными, если ни одна из них не произошла раньше другой (см. пункт «Связь типа "происходит до" и конкурентный доступ» подраздела «Обнаружение конкурентных операций записи» раздела 5.4). Другими словами, два события упорядочены при условии наличия между ними причинно-следственной связи (одно произошло раньше другого), но они несравнимы, если конкурентны. Это значит, что причинность определяет *частичную*, а не полную упорядоченность: одни операции упорядочены одна относительно другой, а другие — несравнимы.

Согласно этому определению в линеаризуемом хранилище данных нет конкурентных операций: должна быть единая временная шкала, относительно которой все операции полностью упорядочены. Может существовать несколько запросов, ожидающих обработки, но хранилище гарантирует, что каждый из них будет обработан атомарно в один момент времени, действуя на одну копию данных, по одной временной шкале, без каких-либо конкурентных процессов.

Конкурентные процессы означали бы, что временная шкала разветвляется и потом сливается снова, — и в этом случае операции в разных ветвях шкалы несравнимы (конкурентны). Данное явление было рассмотрено в главе 5: например, на рис. 5.14 показана не полная упорядоченность, а скорее беспорядочный набор различных операций, происходящих одновременно. Стрелки на схеме показывают причинно-следственные зависимости, определяющие частичную упорядоченность операций.

Если вы знакомы с распределенными системами контроля версий, такими как Git, то их истории версий очень похожи на график причинно-следственных зависимостей. Часто версии разрабатываются одна вслед за другой, по прямой, но иногда создаются ветви (когда над проектом одновременно работают несколько человек), а когда объединяются одновременно созданные версии, происходят слияния.

Линеаризуемость — более сильная зависимость, чем причинная согласованность

Итак, какова связь между причинно-следственной упорядоченностью и линеаризуемостью? Ответ заключается в том, что линеаризуемость *подразумевает* причинность: любая линеаризуемая система сохранит причинно-следственные зависимости [7]. В частности, если в системе имеется несколько каналов связи (например, очередь сообщений и сервис хранения файлов на рис. 9.5), то линеаризуемость гарантирует автоматическое сохранение причинности, для этого не требуются какие-либо

дополнительные меры (например, передача временных меток между компонентами системы).

Именно благодаря тому, что линеаризуемость гарантирует сохранение причинности, линеаризуемые системы столь понятны и привлекательны. Однако, как обсуждалось в подразделе «Цена линеаризуемости» раздела 9.2, сделать систему линеаризуемой обычно означает снизить ее производительность и доступность, особенно если ей свойственны значительные сетевые задержки (например, при условии, что она географически распределена). Как следствие, некоторые распределенные информационные системы отказались от линеаризуемости — это позволяет им обеспечивать более высокую производительность, хотя и может затруднять работу с ними.

Хорошей новостью является то, что здесь можно найти золотую середину. Линеаризуемость не единственный способ сохранения причинности, есть и другие. Система может быть причинно-последовательной без падения производительности, возможном при линеаризации (в частности, таких систем не касается теорема CAP). В сущности, причинно-следственная согласованность — это наиболее сильная модель согласованности, которая не замедляет работу системы из-за сетевых задержек и остается доступной при сбоях сети [2, 42].

Во многих случаях системы, на первый взгляд требующие линеаризуемости, в действительности требуют только причинно-следственную согласованность, которая может быть реализована более эффективно. Основываясь на этом наблюдении, исследователи изучают новые типы баз данных, в которых сохраняется причинность и чьи характеристики производительности и доступности аналогичны характеристикам конечно-согласованных систем [49–51].

Поскольку данное исследование довольно свежее, его результаты пока мало используются в разрабатываемых системах, и все еще остаются проблемы, которые необходимо преодолеть [52, 53]. Однако это перспективное направление для будущих систем.

Выявление причинно-следственных связей

Мы не будем здесь вдаваться в подробности того, каким способом нелинейные системы поддерживают причинно-следственную согласованность, а лишь кратко рассмотрим некоторые ключевые идеи.

Для сохранения причинности нужно знать, *какая операция произошла раньше, а какая — позже*. Это частичная упорядоченность: конкурентные операции могут обрабатываться в любом порядке, но если одна из них произошла раньше другой, то они должны обрабатываться в данной последовательности на всех репликах. Таким образом, когда реплика обрабатывает операцию, она должна гарантировать, что все причинно-предшествующие ей операции (произошедшие раньше нее) уже обработаны; если какая-либо из предыдущих операций отсутствует, то более поздняя операция должна дождаться обработки этой предыдущей операции.

Чтобы определить причинные зависимости, необходимо описать то, откуда узел системы знает о причинных зависимостях. Если узел уже видел значение X, когда записывал значение Y, то X и Y могут быть причинно связаны. При этом анализе используются вопросы, подобные тем, которые обычно задают при уголовном расследовании мошенничества: знал ли генеральный директор об X, когда принял решение Y?

Методы определения того, какая операция произошла раньше, а какая — позже, аналогичны тем, что были рассмотрены в подразделе «Обнаружение конкурентных операций записи» раздела 5.4. Там обсуждалась причинность в хранилище информации без ведущего узла, где нужно обнаруживать конкурентные записи по одному и тому же ключу, чтобы предотвратить потерю обновлений. Причинная зависимость идет дальше: она должна отслеживать такие зависимости не для одного ключа, а для всей базы данных. В этом случае используют обобщенные векторы версий [54].

Чтобы построить причинную последовательность, БД должна знать, какая версия данных была прочитана приложением. Именно поэтому на рис. 5.13 номер версии из предыдущей операции при записи передается обратно в базу. Аналогичная идея применяется при обнаружении конфликтов SSI, как показано в подразделе «Сериализуемая изоляция снимков состояния (SSI)» раздела 7.3: при завершении транзакции база проверяет, является ли актуальной та версия данных, которую она читает. С этой целью БД отслеживает, какой транзакцией были прочитаны те или иные данные.

Упорядоченность по порядковым номерам

Хотя причинность является важной теоретической концепцией, в реальности отслеживание всех причинно-следственных зависимостей может оказаться непрактичным. Во многих приложениях клиенты читают множество данных перед тем, как что-то написать, и потому неясно, зависит ли запись от всех сообщений или только от неких, прочитанных перед этим. Явное отслеживание всех данных, которые были прочитаны, будет означать большие вычислительные затраты.

Однако есть способ получше: для упорядочения событий можно использовать *порядковые номера* или *временные метки*. Такая метка не должна ставиться по часам времени суток (или физическим часам, с которыми связано много проблем, как показано в разделе 8.3). Вместо этого можно опираться на *логические часы* — алгоритм генерации последовательности чисел для идентификации операций, обычно с применением счетчиков, которые увеличиваются для каждой операции.

Такие порядковые номера или временные метки являются компактными (размером всего несколько байтов) и обеспечивают *полную упорядоченность*: каждая операция имеет уникальный порядковый номер; всегда можно сравнить два таких номера, чтобы определить, какой из них больше (и следовательно, какая операция произошла позже).

В частности, благодаря этому можно строить полную упорядоченность по порядковым номерам, *согласующуюся с причинностью*[1]: таким образом гарантируется, что если операция A была причиной операции B, то в общей последовательности A идет перед B (порядковый номер A меньше, чем у B). Конкурентные операции могут быть упорядочены произвольно. Такая полная упорядоченность не только хранит всю информацию о причинно-следственных зависимостях, но вносит даже больше порядка, чем необходимо для сохранения причинности.

В базе данных с репликацией с одним ведущим узлом (см. раздел 5.1) журнал репликации определяет общую последовательность операций записи, которая согласуется с причинностью. Ведущий узел может просто увеличивать счетчик для каждой операции, тем самым присваивая монотонно возрастающие порядковые номера всем операциям в журнале репликации. Если ведомый узел применяет записи в том порядке, в котором они появляются в журнале репликации, то его состояние всегда является причинно-упорядоченным (даже в случае его отставания от ведущего узла).

Генераторы неполных порядковых номеров

Если единого ведущего узла нет (возможно, потому, что применяется база данных с несколькими ведущими, или без него, или же разделенная БД), менее понятно, как генерировать порядковые номера для операций. На практике используются следующие методы.

- Каждый узел может генерировать собственный независимый набор порядковых номеров. Например, если есть два узла, то один из них может выдавать только нечетные числа, а другой — только четные. Обобщая этот принцип, можно зарезервировать несколько битов в двоичном представлении порядкового номера, в котором будет храниться уникальный идентификатор узла. Тогда два разных узла не смогут сгенерировать одинаковые порядковые номера.
- Можно прикреплять к каждой операции временную метку от часов времени суток (физических часов) [55]. Такие метки не являются последовательными, но если у них достаточно высокое разрешение, то этого может быть достаточно для полной упорядоченности операций. Данное обстоятельство используется при разрешении конфликтов методом «выигрывает последний» (см. пункт «Метки даты/времени и упорядочение событий» подраздела «Ненадежность синхронизированных часов» раздела 8.3).

---

[1] Полную упорядоченность, несовместимую с причинностью, легко создать, но она не очень полезна. Например, можно генерировать случайные UUID для каждой операции и затем сравнивать их лексикографически для определения общей последовательности операций. Это в самом деле полная упорядоченность, но случайные UUID не сообщают нам ничего о том, какая операция в действительности произошла раньше, а какая — позже и были ли операции конкурентными.

- Можно предварительно распределить между узлами диапазоны порядковых номеров. Например, узел A получит диапазон от 1 до 1000, а узел B — от 1001 до 2000. Затем каждый из них может независимо назначать порядковые номера из своего диапазона, а когда они закончатся, получать новый.

Эти три метода имеют более высокую производительность и лучше масштабируются, чем если бы все операции проходили через один ведущий узел, который увеличивает счетчик. Они позволяют генерировать уникальные, приблизительно возрастающие порядковые номера для каждой операции. Однако все методы имеют проблему: создаваемые ими порядковые номера *не согласуются с причинностью*.

Проблемы причинности возникают из-за того, что генераторы порядковых номеров неправильно фиксируют порядок операций, исходящих от разных узлов.

- Каждый узел может обрабатывать разное количество операций в секунду. Таким образом, если один узел генерирует четные числа, а другой — нечетные, то счетчик четных чисел может отставать от счетчика нечетных или наоборот. При наличии операции с нечетным номером и операции с четным нельзя точно определить, какая из них произошла раньше другой.
- Временные метки физических часов подвержены сдвигу часов, из-за чего могут оказаться несовместимыми с причинностью. Например, на рис. 8.3 показан сценарий, в котором операция, произошедшая позже, получила более раннюю метку времени[1].
- При назначении диапазонов порядковых номеров одной операции может быть присвоен порядковый номер в диапазоне от 1001 до 2000, в то время как причинно более поздняя операция может получить номер в диапазоне от 1 до 1000. Такие порядковые номера опять же несовместимы с причинностью.

Временные метки Лампорта

Несмотря на то что три описанных генератора порядковых номеров несовместимы с причинностью, на практике существует простой метод генерации порядковых номеров, который согласуется с нею. Он называется *временной меткой Лампорта* и был предложен Лесли Лампортом (Leslie Lamport) в 1978 году [56] — сейчас это одна из самых цитируемых работ в области распределенных информационных систем.

---

[1] Временные метки физических часов можно сделать согласованными с причинностью: в пункте «Синхронизация часов для глобальных снимков состояния» подраздела «Ненадежность синхронизированных часов» раздела 8.3 рассматривался продукт Google Spanner, который оценивает ожидаемый сдвиг часов и ожидает в течение неопределенного времени, прежде чем совершить запись. Этот метод гарантирует, что причинно более поздняя транзакция получит большую временную метку. Однако львиная доля часов не способны обеспечить требуемую метрику неопределенности.

Использование временных меток Лампорта показано на рис. 9.8. Каждый узел имеет уникальный идентификатор и хранит счетчик количества обработанных им операций. Временная метка Лампорта — это просто пара типа *«счетчик, ID узла»*. Два узла могут иногда иметь одно и то же значение счетчика, но благодаря ID узла каждая метка времени становится уникальной.

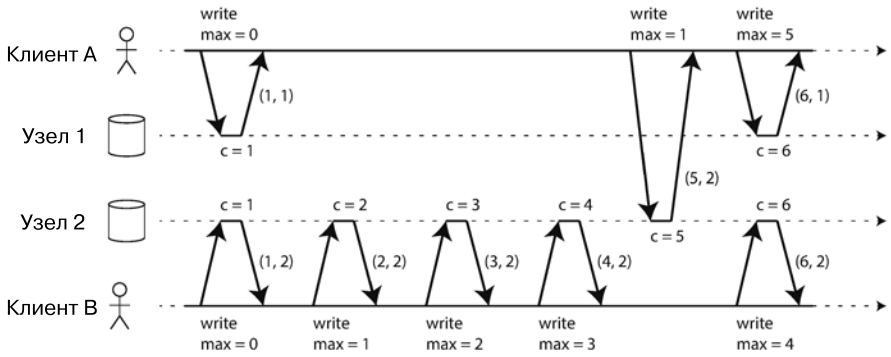


Рис. 9.8. Временные метки Лампорта обеспечивают полное упорядочение, соответствующее причинности

Временная метка Лампорта не имеет отношения к физическим часам времени суток, но обеспечивает полную упорядоченность: из двух временных меток большей будет та, у которой больше значение счетчика; если значения счетчиков одинаковы, то больше будет метка с бóльшим ID узла.

Пока что приведенное описание в целом совпадает с четными и нечетными счетчиками, описанными выше. Основная идея временных меток Лампорта, которая делает их совместимыми с причинностью, заключается в следующем: каждый узел и каждый клиент отслеживают *максимальное* значение счетчика, встречавшееся им до сих пор, и включают данное значение в каждый запрос. Когда узел получает запрос или ответ со значением счетчика, бóльшим, чем его собственное, он немедленно увеличивает свой счетчик до этого максимума.

Такое положение дел показано на рис. 9.8, где клиент A получает от узла 2 значение счетчика 5 и затем передает это максимальное значение 5 узлу 1. В это время счетчик узла 1 был равен всего 1, но он сразу же увеличивается до 5, так что при следующей операции счетчик увеличится на единицу и будет равен 6.

Поскольку максимальное значение счетчика передается с каждой операцией, эта схема гарантирует, что упорядочение с временными метками Лампорта будет причинно-согласованным: каждая причинная зависимость приводит к увеличению временной метки.

Временные метки Лампорта иногда путают с векторами версий, описанными в подразделе «Обнаружение конкурентных операций записи» раздела 5.4. Несмотря на

то что между ними действительно есть некоторое сходство, у этих методов разные цели: векторы версий позволяют определить, являются ли две операции конкурентными, или же одна из них — следствие другой, тогда как временные метки Лампорта всегда обеспечивают полное упорядочение. По общей последовательности временных меток Лампорта невозможно определить, являются ли две операции конкурентными или же находятся в причинно-следственной зависимости. Преимущество временных меток Лампорта над векторами версий состоит в том, что они более компактны.

Временных меток недостаточно

Несмотря на то что временные метки Лампорта определяют полную последовательность операций, которая согласуется с причинностью, их недостаточно для решения многих распространенных проблем, возникающих в распределенных информационных системах.

Например, рассмотрим систему, которая должна гарантировать, что имя пользователя однозначно идентифицирует его учетную запись. Если два пользователя одновременно попытаются создать учетную запись с одним и тем же именем пользователя, это должно получиться только у одного из них, а второй должен получить сообщение об ошибке. (Данная проблема упоминалась ранее в пункте «Ведущий узел и блокировки» подраздела «Истина определяется большинством» раздела 8.4.)

На первый взгляд кажется, что для решения этой задачи достаточно полного упорядочения операций (например, с помощью временных меток Лампорта): если создаются две учетные записи с одинаковым именем пользователя, то завершается успешно та операция, которая имеет меньшую временную метку (пользователь, первым занявший данное имя), и пускай тот, у кого временная метка больше, потерпит неудачу. Поскольку временные метки полностью упорядочены, это сравнение всегда будет справедливым.

Такой метод работает для определения победителя постфактум: сначала надо собрать все операции создания имени пользователя в системе, а потом можно сравнить их временные метки. Однако его недостаточно, если узел только что получил запрос от пользователя для создания нового имени пользователя и должен решить *прямо сейчас*, будет ли данный запрос успешным. В текущий момент узел не знает, будет ли другой узел в это же время создавать учетную запись с тем же именем пользователя и какую временную метку он назначит данной операции.

Для обретения уверенности в том, что ни один узел не создает в это самое время учетную запись с таким же именем пользователя и более низкой меткой времени, нужно проверить все остальные узлы и узнать, чем они заняты [56]. Если один из узлов терпит сбой или недоступен из-за проблем со связью, то вся система остановится. Такая отказоустойчивость нам не подходит.

Проблема заключается в следующем: полное упорядочение операций возникает только после того, как были собраны все операции. Если другой узел сгенерировал некоторые операции, но вы еще не знаете, что они собой представляют, то не можете построить полную последовательность операций: может появиться необходимость вставить неизвестные операции от другого узла в произвольные места общей последовательности.

Итак, для реализации чего-то вроде ограничения по уникальности для имен пользователей недостаточно полного упорядочения операций — также необходимо знать, когда будет построена окончательная последовательность. Если при выполнении операции по созданию имени пользователя ни один другой узел не может заявить права на то же имя, но с более ранней временной меткой в общей последовательности, то можно уверенно объявить операцию успешной.

Эта идея — знать, когда общая последовательность построена, — реализована в методе *рассылки общей последовательности*.

Рассылка общей последовательности

Если программа работает только на одном ядре процессора, то определить общую последовательность операций легко: это тот порядок, в котором они выполнялись процессором. Однако в распределенной системе построить согласованную общую последовательность операций для всех узлов гораздо сложнее. В последнем пункте была рассмотрена последовательность, построенная по временным меткам или порядковым номерам. Но мы обнаружили, что этот метод не такой эффективный, как репликация с одним ведущим узлом (если использовать упорядочение по временным меткам для реализации ограничения уникальности, то сбои в сети недопустимы).

Как уже обсуждалось, репликация с одним ведущим узлом определяет общую последовательность операций, выбирая один узел в качестве ведущего и упорядочивая все операции относительно его процессора. Однако в этом случае возникает проблема масштабирования системы для случая, когда пропускная способность выше, чем может обрабатывать один ведущий узел. Кроме того, неясно, как обеспечить отказоустойчивость в случае, если ведущий даст сбой (см. подраздел «Перебои в обслуживании узлов» раздела 5.1). В литературе о распределенных системах эта проблема известна как *рассылка общей последовательности* или *атомарная рассылка* [25, 57, 58][1].

[1] Термин «атомарная рассылка» является традиционным, но он очень неточен, поскольку несовместим с другими случаями использования слова «атомарный»: у него нет ничего общего с атомарностью в транзакциях ACID и он лишь косвенно связан с атомарными операциями (при многопоточном программировании) и атомарными реестрами (линеаризуемое хранилище). Еще один его синоним — многоадресная рассылка полной последовательности.



Гарантия объема упорядоченности

Разделенные базы данных с одним ведущим узлом в каждом разделе часто поддерживают последовательность операций только в границах раздела. Это значит, что они не предоставляют гарантии согласованности (например, согласованные снимки состояний, ссылки на внешние ключи) между разделами. Общее упорядочение по всем разделам возможно, но требует дополнительной координации [59].

Рассылка общей последовательности обычно описывается как протокол обмена сообщениями между узлами. Неформально это требует, чтобы всегда выполнялись два следующих требования безопасности.

- *Надежная доставка*. Ни одно сообщение не должно быть потеряно. Если сообщение доставляется одному узлу, то оно доставляется всем узлам.
- *Полностью упорядоченная доставка*. Сообщения доставляются во все узлы в одном и том же порядке.

Корректный алгоритм рассылки общей последовательности должен гарантировать, что функции надежности и упорядоченности выполняются всегда, даже если узел или сеть неисправны. Конечно, сообщения не станут доставляться при перебоях в сети, но алгоритм может повторять попытки до тех пор, пока сеть не восстановится и сообщения не будут переданы (и они все равно должны быть доставлены в правильном порядке).

Использование рассылки общей последовательности

Сервисы консенсуса, такие как ZooKeeper и т. п., фактически реализуют рассылку общей последовательности. Данный факт намекает на существование тесной связи между рассылкой полной последовательности и консенсусом, который будет рассмотрен далее в этой главе.

Рассылка общей последовательности — именно то, что нужно для репликации базы данных: если каждое сообщение представляет собой запись в базу и каждая реплика обрабатывает одни и те же записи в той же последовательности, то реплики будут оставаться согласованными между собой (кроме временных задержек репликации). Этот принцип известен как *репликация конечных автоматов* [60], и мы вернемся к нему в главе 11.

Аналогично рассылка общей последовательности может использоваться для реализации сериализуемых транзакций: как описано в подразделе «По-настоящему последовательное выполнение» раздела 7.3, если каждое сообщение представляет собой детерминированную транзакцию, которая должна выполняться как хранимая процедура, и если каждый узел обрабатывает эти сообщения в одном и том же порядке, то разделы и реплики базы данных будут согласованы между собой [61].

Важным аспектом рассылки общей последовательности является то, что последовательность фиксируется в момент доставки сообщений: узел не имеет права вставлять сообщения задним числом в более раннюю позицию последовательности, если последующие сообщения уже доставлены. Благодаря этому рассылка общей последовательности более строгая, чем временное упорядочение.

Рассылка общей последовательности также подобна созданию *журнала* (репликации, транзакций или записи): доставка сообщения напоминает запись в журнал. Поскольку все узлы должны доставлять одни и те же сообщения в одинаковом порядке, то могут читать журнал и видеть одну и ту же последовательность сообщений.

Кроме того, рассылка общей последовательности полезна для реализации сервиса блокировки, которая предоставляет ограждающие маркеры (см. одноименный пункт подраздела «Истина определяется большинством» раздела 8.4). Каждый запрос на получение блокировки заносится в журнал как сообщение. Все сообщения последовательно нумеруются в порядке их появления в журнале. Затем эти порядковые номера могут служить маркерами-ограничителями, поскольку монотонно возрастают. В ZooKeeper такие порядковые номера называются `zxid` [15].

Реализация линеаризуемого хранилища с помощью рассылки общей последовательности

Как показано на рис. 9.4, если система линеаризуема, то в ней существует рассылка общей последовательности операций. Означает ли это, что линеаризуемость — то же самое, что и рассылка общей последовательности? Не совсем. Но между ними существует тесная взаимосвязь[1].

Рассылка общей последовательности асинхронна: она гарантирует надежную доставку сообщений в фиксированном порядке, но не гарантирует *время* доставки сообщений (так что один получатель может отставать от остальных). Напротив, линеаризуемость гарантирует своевременность: операция чтения заведомо выдаст последнее записанное значение.

Однако при наличии рассылки общей последовательности можно создать на ее основе линеаризуемое хранилище, например гарантировать, что имена пользова-

[1] С формальной точки зрения линеаризуемый реестр чтения-записи является «более простой» задачей. Рассылка общей последовательности эквивалентна консенсусу [67], у которого нет детерминированного решения в асинхронной модели аварийной остановки [68], тогда как линеаризуемый реестр чтения-записи может быть реализован в этой системной модели [23–25]. Однако поддержка атомарных операций, таких как сравнение с присвоением или приращение с чтением в реестре, делает его эквивалентным консенсусу [28]. Таким образом, проблемы консенсуса и линеаризуемого реестра тесно взаимосвязаны.

телей будут уникальны и станут однозначно идентифицировать учетные записи пользователей.

Представьте, что для каждого возможного имени пользователя имеется линеаризуемый реестр с атомарной операцией сравнения с присвоением. Первоначально значение каждого реестра равно `null` (указывающее на незанятость данного имени). Когда пользователь хочет создать имя, в реестре для этого имени выполняется операция сравнения с присвоением и его значением становится ID учетной записи пользователя при условии, что предыдущее значение реестра было равно `null`. Если несколько пользователей попытаются одновременно получить одно и то же имя, то только одна из операций сравнения с присвоением будет успешной, так как остальные увидят значение, отличное от `null` (вследствие линеаризуемости).

Такую линеаризуемую операцию сравнения с присвоением можно реализовать следующим образом, задействуя рассылку общей последовательности как журнал с разрешением только на дополнение [62, 63].

1. Записать сообщение в журнал, предварительно указав желаемое имя пользователя.
2. Прочитать журнал и дождаться, пока добавленное в п. 1 сообщение не будет возвращено[1].
3. Проверить наличие сообщений с требованием такого же имени пользователя. Допустим, первое сообщение для желаемого имени является вашим сообщением, тогда операция считается успешной: можно зафиксировать заявку на имя (например, добавив в журнал новое сообщение) и подтвердить его клиенту. Если же первое сообщение для желаемого имени принадлежит другому пользователю, то операцию следует прервать.

Поскольку записи журнала рассылаются всем узлам в одной и той же последовательности, при наличии нескольких конкурентных записей все узлы будут единого мнения о том, какая из них была первой. Выбор первой из конфликтующих записей в качестве успешной и прерывание последующих гарантирует, что все узлы будут согласованы в отношении того, была ли запись принята или прервана. Аналогичный метод можно использовать для реализации на базе журнала сериализуемых транзакций с несколькими объектами [62].

Несмотря на то что эта процедура обеспечивает линеаризуемую запись, она не гарантирует линеаризуемое чтение: если читать из хранилища, которое асинхронно обновляется из журнала, то данные могут быть устаревшими. (Точнее, описанная здесь процедура обеспечивает *последовательную согласованность* [47, 64], иногда также называемую *временной согласованностью* [65, 66], — несколько менее сильная

[1] Если не ждать, а сразу подтвердить запись после ее постановки в очередь, то получим нечто похожее на модель согласованной памяти многоядерных процессоров x86 [43]. Эта модель нелинеаризуема и непоследовательна.

гарантия, чем линеаризуемость.) Перечислим несколько вариантов, позволяющих сделать чтение линеаризуемым.

- Можно последовательно читать через журнал, добавляя туда сообщение, считывая журнал и затем выполняя фактическое чтение после возврата сообщения. Таким образом, позиция сообщения в журнале определяет момент времени, в который происходит чтение. (Примерно так реализованы операции чтения в Quorum [16].)
- Если журнал позволяет получить позицию последнего сообщения линеаризуемым способом, то можно запросить ее, дождаться, пока вам будут доставлены все записи до этой позиции, а затем выполнить чтение. (Данная идея лежит в основе операции `sync()` в ZooKeeper [15].)
- Можно выполнить чтение реплики, которая синхронно обновляется при записи и, следовательно, наверняка будет актуальной. (Этот метод используется при цепной репликации [63]; см. также врезку «Исследования вопроса репликации» в подразделе «Синхронная и асинхронная репликация» раздела 5.1.)

Реализация рассылки общей последовательности с помощью линеаризуемого хранилища

В предыдущем пункте было показано, как построить линеаризуемую операцию сравнения с присвоением на основе рассылки общей последовательности. Но можно посмотреть на этот метод с другой стороны: предположить, что у нас имеется линеаризуемое хранилище, и показать, как построить на его основе рассылку общей последовательности.

Самый простой способ — предположить, что у нас есть линеаризуемый реестр, который хранит целое число и позволяет выполнять атомарную операцию приращения с чтением [28] или же, как вариант, атомарную операцию сравнения с присвоением.

Алгоритм прост: для каждого сообщения, которое мы хотим отправить через рассылку общей последовательности, выполняем операцию приращения с чтением и получаем линеаризуемое целое число, а затем прикрепляем полученное значение реестра к сообщению в виде порядкового номера. Далее можно отправить сообщение всем узлам (повторно пересылая все потерянные сообщения), а получатели будут доставлять сообщения последовательно, по порядковым номерам.

Обратите внимание: в отличие от временных меток Лампорта, числа, которые мы получаем в результате приращения линеаризуемого реестра, образуют последовательность без пропусков. Таким образом, если узел доставил сообщение 4 и получает входящее сообщение с порядковым номером 6, то знает, что сначала должен дождаться сообщения 5 и только потом сможет доставить сообщение 6. В случае временных меток Лампорта все не так, и это ключевое различие между рассылкой общей последовательности и упорядочением по временным меткам.

Насколько сложно было бы генерировать линеаризуемое целое число с помощью операции атомарного приращения с чтением? Как обычно, в отсутствие сбоев это было бы легко: можно просто хранить его в переменной на одном узле. Проблема заключается в обработке ситуации, когда сетевое соединение с данным узлом прерывается, и в восстановлении значения, когда в узле произошел сбой [59]. В общем, если достаточно хорошо продумать генерацию линеаризуемых порядковых номеров, то неизбежно получим консенсусный алгоритм.

Это не совпадение: можно доказать, что линеаризуемый реестр сравнения с присвоением (или приращения с чтением) и рассылка общей последовательности *эквивалентны консенсусу* [28, 67]. Другими словами, если одна из указанных проблем решена, то она превращается в решение для других. Удивительно, не правда ли?

Наконец, пора решить проблему консенсуса, и этому мы посвятим оставшуюся часть данной главы.

9.4. Распределенные транзакции и консенсус

Консенсус является одной из важнейших и фундаментальных проблем распределенных вычислений. На первый взгляд все кажется легким: говоря простыми словами, цель состоит в том, чтобы *заставить несколько узлов согласовать некие объекты*. Вы могли подумать: это не должно быть слишком сложно. К сожалению, многие системы были разрушены, поскольку их создатели исходили из ошибочного убеждения, что данную проблему легко решить.

Несмотря на всю важность консенсуса, посвященный ему раздел появляется лишь в конце этой книги, так как сама тема очень тонкая и оценка всех нюансов требует некоторых предварительных знаний. Даже в научно-исследовательском сообществе понимание консенсуса складывалось постепенно, в течение десятилетий, и сопровождалось многими недоразумениями. Теперь, после изучения репликации (глава 5), транзакций (глава 7), системных моделей (глава 8), линеаризуемости и рассылки общей последовательности (настоящая глава), мы наконец готовы взяться за проблему консенсуса.

Существует ряд ситуаций, в которых важно, чтобы узлы были согласованными, в том числе следующие.

- *Выбор ведущего узла*. В базе данных с репликацией с одним ведущим узлом все узлы должны прийти к единому мнению о том, какой из них является ведущим. Его позиция может быть оспорена, если некоторые узлы не смогут общаться с другими из-за сбоя в сети. В этом случае важно достичь консенсуса, чтобы избежать неудачного восстановления после отказа, что приводит к ситуации с разделенным интеллектом, когда два узла считают себя ведущими (см. подраздел «Перебои в обслуживании узлов» раздела 5.1). Если бы ведущих узлов

оказалось два, то они оба подтверждали бы операции записи и их данные расходились бы; это приводило бы к несогласованности и потере данных.

- *Атомарная фиксация*. В базе данных, которая поддерживает транзакции, охватывающие несколько узлов или разделов, иногда возникает проблема: транзакция может не сработать на одних узлах, но успешно закончиться на других. Для поддержания атомарности транзакций (в смысле ACID, см. пункт «Атомарность» подраздела «Смысл аббревиатуры ACID» раздела 7.1) нужно заставить все узлы согласовать исход транзакции: либо во всех узлах она будет прервана/произойдет откат (если что-то пойдет не так), либо же во всех узлах будет подтверждена (если ничего не случится). Этот пример консенсуса известен как проблема атомарной фиксации[1].

Невозможность консенсуса

Вероятно, вы слышали о результатах FLP [68] — исследования, названного в честь авторов, Фишера (Fischer), Линча (Lynch) и Патерсона (Paterson). В нем доказывается невозможность построить алгоритм, который бы всегда позволял достичь консенсуса, если в узле возможен сбой. В распределенной системе приходится предположить, что узлы могут давать сбои, поэтому надежный консенсус невозможен. Чем же мы тут занимаемся, обсуждая алгоритмы достижения консенсуса?

Ответ заключается в том, что результат FLP подтвержден в модели асинхронной системы (см. подраздел «Модели системы на практике» раздела 8.4). Это очень ограниченная модель, предполагающая детерминированный алгоритм, который не может использовать часы или задержки. Когда в алгоритме разрешено применять задержки или какой-либо другой способ идентификации узлов, в которых, вероятно, произошел сбой (даже если иногда такое подозрение ошибочно), консенсус становится достижимым [67]. Даже простого позволения алгоритму задействовать случайные числа уже достаточно, чтобы обойти выводы о его невозможности [69].

Таким образом, несмотря на то что результат FLP о невозможности консенсуса имеет большое теоретическое значение, на практике в распределенных системах консенсус обычно достижим.

В этом разделе мы более подробно рассмотрим проблему атомарной фиксации. В частности, обсудим алгоритм *двухфазной фиксации* (two-phase commit, 2PC), который является наиболее распространенным способом атомарной фиксации

[1] Определение атомарной фиксации несколько отличается от определения консенсуса: атомарная транзакция может быть завершена в том и только в том случае, если все ее участники проголосовали за завершение, и должна быть прервана, если один из участников должен ее прервать. Консенсус может быть достигнут в отношении любого значения, предложенного одним из участников. Однако атомарное подтверждение и консенсус могут быть сведены одно к другому [70, 71]. Неблокирующая атомарная фиксация сложнее консенсуса (см. пункт «Трехфазная фиксация» подраздела «Атомарная и двухфазная фиксация (2PC)» раздела 9.4).

и реализован в различных базах данных, системах обмена сообщениями и серверах приложений. Оказывается, 2PC является своего рода консенсусным алгоритмом, хотя и не очень хорошим [70, 71].

Изучив 2PC, мы перейдем к лучшим консенсусным алгоритмам, таким как те, что используются в ZooKeeper (Zab) и etcd (Raft).

Атомарная и двухфазная фиксация (2PC)

Из главы 7 мы узнали цель атомарных транзакций: обеспечить простую семантику в том случае, если в процессе серии операций записи что-то пойдет не так. Результатом транзакции является либо успешная *фиксация*, и в этом случае все операции записи, входящие в состав транзакции, делаются долговременными, либо *прерывание*, и тогда все операции записи транзакции откатываются (отменяются или отбрасывается).

Атомарность предотвращает засорение базы данных неполными результатами и недообновленными состояниями неудачных транзакций. Это особенно важно для транзакций с участием нескольких объектов (см. подраздел «Однообъектные и многообъектные операции» раздела 7.1) и баз с поддержкой вторичных индексов. Каждый такой индекс представляет собой отдельную информационную структуру, построенную на основе первичных данных. Поэтому при изменении некоторых данных соответствующее изменение должно произойти и во вторичном индексе. Атомарность гарантирует, что этот индекс останется согласованным с первичными данными (если бы он стал несовместимым с ними, то пользы от него было бы немного).

От одноузлового до распределенной атомарной фиксации

Для транзакций, выполняемых в одном узле базы данных, атомарность обычно достигается за счет механизма хранения. Когда клиент запрашивает у узла БД фиксацию транзакции, база делает записи транзакций долговечными (обычно за счет журнала записи с упреждением, см. пункт «Обеспечение надежности B-деревьев» подраздела «B-деревья» раздела 3.1), а затем вносит в журнал на диске запись о фиксации. Если база данных выходит из строя в середине этого процесса, то при перезапуске узла транзакция восстанавливается из журнала. При успешной записи сообщения о фиксации на диск до сбоя транзакция считается зафиксированной; в противном случае все записи этой транзакции отменяются.

Таким образом, в одном узле фиксация транзакций полностью зависит от той *последовательности*, в которой данные записываются на диск для долговременного хранения: сначала данные, затем запись о фиксации [72]. Определяющим моментом того, будет ли транзакция зафиксирована или отменена, является окончание записи

о завершении фиксации на диск: до этого времени все еще можно отменить (из-за сбоя), но после транзакция завершена (даже если произошел сбой БД). Таким образом, за атомарную фиксацию отвечает всего одно устройство (контроллер данного дискового накопителя, прикрепленный к данному узлу).

Но как быть, если в транзакции участвует несколько узлов? Например, когда выполняется транзакция для нескольких объектов в распределенной базе данных или имеется вторичный индекс с разделением по терминам (первичные данные находятся в одном узле, а элементы индекса — в другом, см. раздел 6.3). Большинство распределенных информационных хранилищ типа NoSQL не поддерживают такие распределенные транзакции, в отличие от многих кластерных реляционных систем (см. подраздел «Распределенные транзакции на практике» текущего раздела).

В этих случаях недостаточно только отправить всем узлам запрос о фиксации и затем совершить транзакцию на каждом из них в отдельности: вполне может случиться так, что фиксация будет успешно выполнена в одних узлах и не сработает в других и, как следствие, нарушится гарантия атомарности.

- В одних узлах могут быть обнаружены нарушения ограничений или конфликты и возникнет необходимость прерывания, в то время как в других узлах транзакция будет успешно завершена.
- Некоторые из запросов на фиксацию могут быть потеряны в сети или отменены из-за сетевых задержек, в то время как остальные пройдут.
- Одни узлы могут претерпеть сбой прежде, чем будет завершена запись о фиксации, и выполнить откат для восстановления, в то время как другие успешно завершают транзакцию.

Если одни узлы совершают транзакцию, а другие прерывают ее, то узлы становятся несовместимыми между собой (как на рис. 7.3). После того как транзакция зафиксирована в одном узле, ее нельзя отменить в случае выяснения позже, что в другом узле она была прервана. По этой причине узел должен зафиксировать транзакцию только после того, как будет точно известно, что на остальных узлах данная транзакция также успешно завершена.

Фиксирование транзакции должно быть неотменяемым — не должно быть возможности «передумать» и задним числом прервать транзакцию после ее фиксации. Причина этого правила такова: после того как данные были зафиксированы, они становятся видимыми для других транзакций и, следовательно, другие клиенты могут начать опираться на эти данные. Описанный принцип лежит в основе *чтения зафиксированных данных*, представленного в одноименном подразделе раздела 7.2. Если разрешить прерывание транзакции после ее фиксации, то любые транзакции, считывающие данные, были бы основаны на данных, которые могли бы быть задним числом объявлены как несуществующие, и тогда их тоже пришлось бы отменить.

(Последствия завершенной транзакции в дальнейшем могут быть отменены другой, *компенсирующей транзакцией* [73, 74]. Однако с точки зрения базы данных это отдельная транзакция, и, следовательно, все требования к корректности между транзакциями являются проблемой приложения.)

Введение в двухфазную фиксацию

Двухфазная фиксация является алгоритмом для атомарной фиксации транзакций в случае нескольких узлов. Другими словами, она гарантирует, что все узлы либо зафиксировали транзакцию, либо прервали ее. Это классический алгоритм для распределенных баз данных [13, 35, 75]. Алгоритм 2PC применяется внутри отдельных БД, а также доступен для приложений в виде *XA-транзакций* [76, 77] (которые, например, поддерживаются Java Transaction API) или через WS-AtomicTransaction для веб-сервисов SOAP [78, 79].

Основная схема 2PC показана на рис. 9.9. Вместо одного запроса на фиксацию, как в случае транзакции в одном узле, процесс фиксации/прерывания в 2PC разбит на две фазы (отсюда и название).

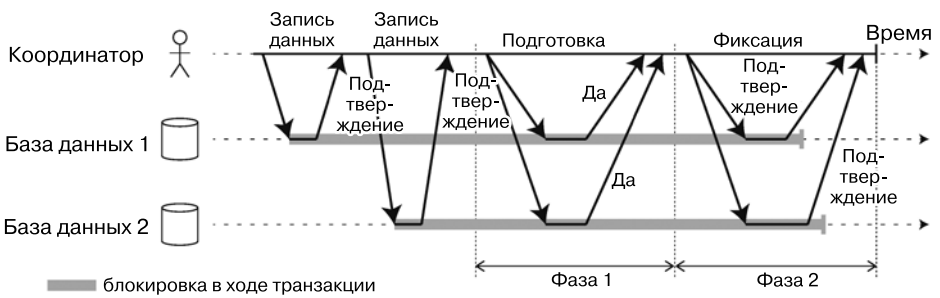


Рис. 9.9. Успешное выполнение двухфазной фиксации (2PC)



Не путайте 2PC и 2PL

Двухфазная фиксация (2PC) и двухфазная блокировка (см. подраздел «Двухфазная блокировка (2PL)» раздела 7.3) — совершенно разные вещи. 2PC обеспечивает атомарную фиксацию в распределенной базе данных, тогда как 2PL — сериализуемую изоляцию. Чтобы избежать путаницы, лучше всего думать о них как о совершенно разных понятиях и игнорировать неудачное сходство названий.

В 2PC используется новый компонент, который обычно не появляется в транзакциях в одном узле: *координатор* (также известный как *диспетчер транзакций*).

Он часто реализуется в виде библиотеки в том же процессе приложения, который запрашивает транзакцию (например, встроенной в контейнер Java EE), но может быть и отдельным процессом или сервисом. Примерами таких координаторов являются Narayana, JOTM, BTM и MSDTC.

Транзакция 2PC начинается с того, что приложение, как обычно, выполняет чтение и запись данных в нескольких узлах БД. Эти узлы называют *участниками* транзакции. Когда приложение готово к фиксации, координатор начинает этап 1: он отправляет запрос на *подготовку* каждому из узлов, спрашивая их, могут ли они выполнить фиксацию. Затем координатор отслеживает ответы участников.

- Если все участники ответили «да», показав, что они готовы к фиксации, то на этапе 2 координатор отправляет запрос *фиксации* и выполняется фиксация.
- Если один из участников ответил «нет», то на этапе 2 координатор отправляет всем узлам запрос *прерывания*.

Описанный процесс в чем-то похож на традиционную церемонию бракосочетания в западных культурах: священник сначала спрашивает отдельно невесту и жениха, хочет ли каждый из них вступить в брак, и обычно получает от обоих ответ «да». Получив подтверждения от обоих, он объявляет их мужем и женой: транзакция зафиксирована, и этот счастливый факт сообщается всем присутствующим. Если же невеста или жених не скажет «да», то церемония прерывается [73].

Система обещаний

Из этого краткого описания может быть непонятно, почему двухфазная фиксация обеспечивает атомарность, а однофазная фиксация по нескольким узлам — нет. Разумеется, при двухфазной фиксации запросы на подготовку и фиксацию также можно легко потерять. В чем же отличие 2PC?

Чтобы понять, почему это работает, разобъем данный процесс на более мелкие детали.

1. Когда приложение начинает распределенную транзакцию, оно запрашивает у координатора ID транзакции. Этот идентификатор глобально уникален.
2. Приложение начинает одноузловую транзакцию для каждого из участников и прикрепляет этот глобально уникальный ID к каждой одноузловой транзакции. Все операции чтения и записи выполняются в рамках одной из этих транзакций. Если на данном этапе что-то пойдет не так (например, случится сбой узла или будет превышено время запроса), то координатор или любой из участников может отменить транзакцию.
3. Когда приложение готово к фиксации, координатор отправляет всем участникам запрос на ее подготовку, помеченный глобальным ID транзакции. Если какой-либо из этих запросов окажется неудачным или превысит свое

время, то координатор отправит всем участникам запрос прерывания транзакции с этим ID.

4. Когда участник получает запрос на подготовку, он убеждается, что точно может завершить транзакцию при любых обстоятельствах, в том числе записать все данные транзакции на диск (если в момент записи диск сломается, случится сбой питания или на диске не хватит места, то это не будет уважительной причиной, чтобы впоследствии отказаться от фиксации) и убедиться в отсутствии каких-либо конфликтов или нарушения ограничений. Отвечая координатору «да», узел обещает завершить транзакцию без ошибок в случае надобности. Другими словами, участник отказывается от права отменить транзакцию, но фактически не совершает ее.
5. Когда координатор получил ответы на все запросы подготовки, он принимает окончательное решение о том, следует ли зафиксировать или прервать транзакцию (фиксация выполняется только в том случае, если все участники ответили «да»). Координатор записывает это решение на диск в свой журнал транзакций, чтобы впоследствии, в случае сбоя, знать, какое решение было принято. Это называется *точкой фиксации*.
6. После того как решение координатора записано на диск, запрос на фиксацию или прерывание отправляется всем участникам. Если данный запрос окажется неудачным или его время истечет, то координатор повторяет его до тех пор, пока запрос не будет успешно отправлен. Это точка невозврата: в случае принятия решения оно должно быть исполнено, независимо от количества попыток. Если в это время участник претерпел сбой, то транзакция будет завершена после того, как он восстановится. Поскольку участник ответил «да», он не может отказаться от завершения транзакции после восстановления.

Таким образом, в данном протоколе есть две критические точки невозврата: когда участник отвечает «да», он обещает, что гарантированно сможет позднее завершить транзакцию (хотя координатор все еще может отказаться от нее); как только координатор принимает решение, оно является бесповоротным. Эти гарантии обеспечивают атомарность 2PC. (При атомарной фиксации в одном узле оба указанных события объединяются в одно: внесение записи о фиксации в журнал транзакций.)

Возвращаясь к аналогии с браком, пока они не сказали «да», у жениха и невесты есть возможность прервать транзакцию, ответив «ни за что» (или нечто подобное). Однако, ответив «да», они не могут отменить данное решение. Если, ответив «да», молодожен потеряет сознание и не услышит, как священник скажет: «Объявляю вас мужем и женой», то это не изменит факта завершения транзакции. Когда позже он придет в чувство, он узнает, женат ли теперь, обратившись к священнику и запросив у него состояние транзакции по ее глобальному ID, или может дождаться очередного повторного запроса священника о завершении транзакции (поскольку он будет их повторять все время, пока молодожен находится без сознания).

Отказ координатора

Мы обсудили, что произойдет, когда во время 2PC с одним из участников или с сетью случится сбой: если какой-либо из запросов на подготовку не сработает или же его время истечет, то координатор прервет транзакцию; если же не сработает один из запросов фиксации или отмены, то координатор будет повторять их в течение неопределенного времени. Но каковы варианты событий в случае выхода координатора из строя?

Если с координатором случится сбой перед отправкой запросов на подготовку, то участник может безопасно отменить транзакцию. Но, как только участник получил запрос на подготовку и ответил «да», он больше не может отменить транзакцию в одностороннем порядке — ему следует дождаться ответа от координатора о том, была ли транзакция подтверждена или прервана. Если в этот момент координатор выйдет из строя или случится сбой в сети, то участнику остается только ждать. Транзакция участника в таком состоянии называется *сомнительной* или *неопределенной*.

Такая ситуация проиллюстрирована на рис. 9.10. В этом примере координатор принял решение совершить транзакцию и база данных 2 получила запрос на фиксацию. Однако координатор вышел из строя, прежде чем смог отправить запрос фиксации в базу данных 1, и поэтому она не знает, следует завершить или прервать транзакцию. Даже временная задержка здесь не помогает: если после задержки база 1 в одностороннем порядке отменит транзакцию, то окажется несовместимой с базой 2, которая ее зафиксировала. Фиксировать транзакцию в одностороннем порядке также небезопасно, поскольку другой участник мог ее прервать.

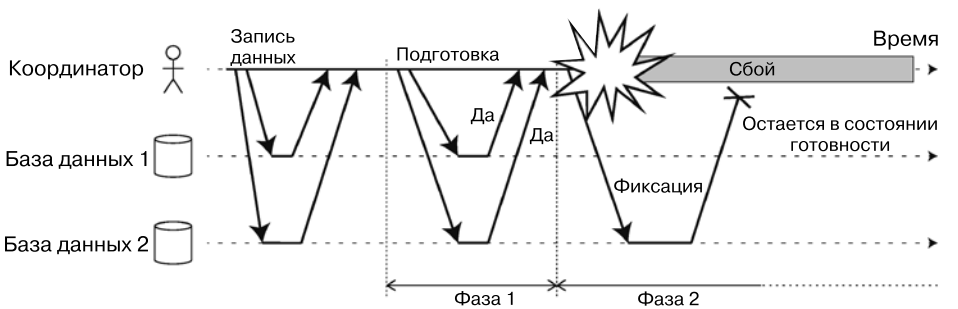


Рис. 9.10. Участники ответили «да», и в этот момент координатор вышел из строя. База данных 1 не знает, следует завершить или прервать транзакцию

Не получая ответа от координатора, участник никак не может узнать, следует завершить или прервать транзакцию. В принципе, участники могли бы общаться между собой с целью узнать, что ответил каждый из них на запрос координатора, и прийти к какому-то соглашению, но это не является частью протокола 2PC.

Единственный способ, которым может завершиться процедура 2PC, — это ожидание восстановления координатора. Вот почему он должен записать свое решение о фиксации или отмене транзакции в журнал транзакций на диске, прежде чем отправлять участникам запросы о фиксации или отмене транзакции: когда координатор восстановится, он определит статус всех сомнительных транзакций, прочитав свой журнал. Любые транзакции, не имеющие записи о фиксации в этом журнале, прерываются. Таким образом, точка фиксации 2PC сводится к регулярной одноузловой атомарной фиксации у координатора.

Трехфазная фиксация

Двухфазную фиксацию называют *блокирующим* протоколом атомарной фиксации из-за того, что 2PC способна зависнуть в ожидании восстановления координатора. Теоретически можно построить *неблокирующий* протокол атомарной фиксации, который не будет зависать, если узел выйдет из строя. Однако реализовать это на практике непросто.

В качестве альтернативы 2PC был предложен алгоритм, называемый *трехфазной фиксацией* (three-phase commit, 3PC) [13, 80]. Однако он предполагает наличие сети с ограниченной задержкой и узлами с ограниченным временем отклика; в большинстве реальных систем с неограниченными сетевыми задержками и паузами процессов (см. главу 8) он не гарантирует атомарность.

В общем случае для неблокирующей атомарной фиксации необходим *идеальный детектор отказов* [67, 71] — надежный механизм для определения того, вышел ли узел из строя. В сети с неограниченными задержками время ожидания не является надежным детектором отказа, поскольку время запроса может истечь из-за сетевой проблемы, даже если ни один узел не вышел из строя. По этой причине 2PC продолжает использоваться, несмотря на известную проблему со сбоем координатора.

Распределенные транзакции на практике

Распределенные транзакции, особенно реализованные с двухфазной фиксацией, имеют смешанную репутацию. С одной стороны, считается, что они предоставляют гарантию безопасности, которой было бы трудно добиться иными средствами. С другой — их критикуют за то, что они вызывают операционные проблемы, практически убивают производительность и обещают больше, чем могут выполнить [81–84]. Многие облачные сервисы предпочитают обойтись без распределенных транзакций из-за связанных с ними проблем [85, 86].

В некоторых реализациях распределенных транзакций за надежность приходится расплачиваться производительностью. Например, утверждают следующее: распределенные транзакции в MySQL более чем в десять раз медленнее, чем одноузловые транзакции [87], — неудивительно, что их не советуют использовать. Большая

часть затрат на производительность при двухфазной фиксации связана с дополнительным форсированием диска (`fsync`), необходимым для восстановления после сбоя [88], и дополнительными передачами данных по сети.

Однако вместо того, чтобы вообще отказаться от распределенных транзакций, мы рассмотрим их более подробно, поскольку из этого следуют важные выводы. Для начала уточню, что именно подразумеваю под «распределенными транзакциями». Существуют два довольно сильно различающихся типа распределенных транзакций, которые часто объединяют.

- *Внутренние распределенные транзакции базы данных*. Некоторые распределенные БД (использующие репликацию и секционирование в стандартной конфигурации) поддерживают внутренние транзакции между узлами самой базы. Например, такая внутренняя поддержка транзакций реализована в системе хранения информации VoltDB и в MySQL Cluster NDB. В этом случае все узлы, участвующие в транзакции, задействуют одно и то же программное обеспечение БД.
- *Гетерогенные распределенные транзакции*. Участниками *гетерогенной* транзакции являются разные технологии: например, две БД разных производителей или даже системы без БД, такие как брокеры сообщений. Распределенная транзакция в этих системах должна обеспечивать атомарную фиксацию, даже если их природа совершенно различна.

Внутренние транзакции базы данных не должны быть совместимы с какой-либо другой системой, так что в них может использоваться любой протокол и применяться оптимизация, специфичная для этой конкретной технологии. Как следствие, внутренние распределенные транзакции базы часто могут работать вполне хорошо. Однако транзакции, распространяющиеся на гетерогенные технологии, гораздо сложнее.

Однократная обработка сообщений

Гетерогенные распределенные транзакции позволяют интегрировать различные системы в эффективные комплексы. Например, сообщение из очереди сообщений может быть помечено как обработанное тогда и только тогда, когда транзакция базы данных для его обработки была успешно завершена. Такая возможность реализуется с помощью атомарной фиксации подтверждения сообщения, а база выполняет запись за одну транзакцию. Благодаря поддержке распределенных транзакций это осуществимо, несмотря на то что брокер сообщений и БД — несвязанные технологии, работающие на разных машинах.

Если либо доставка сообщения, либо транзакция базы данных завершаются неудачно, то обе операции отменяются, и поэтому брокер сообщений способен впоследствии безопасно повторить отправку сообщения. Таким образом, благодаря атомарной фиксации сообщения и побочных эффектов его обработки можно

гарантировать, что сообщение будет *полностью* обработано ровно один раз, даже если для успешного завершения операции потребуется несколько попыток. Прерывание транзакции отменяет все побочные эффекты частично завершенной транзакции.

Такая распределенная транзакция возможна, только когда все участвующие в ней системы используют один и тот же протокол атомарной фиксации. Например, предположим, что побочным эффектом обработки сообщения является отправка электронной почты, а ее сервер не поддерживает двухфазную фиксацию. Тогда, если обработка сообщения закончится неудачно и будет повторена, электронное письмо может быть отправлено несколько раз. Но если в случае прерывания транзакции все побочные эффекты обработки сообщения отменяются, то обработку можно безопасно повторить, как будто ничего не произошло.

Мы вернемся к теме однократной обработки сообщений в главе 11, но прежде рассмотрим протокол атомарной фиксации, благодаря которому такие гетерогенные распределенные транзакции становятся возможными.

XA-транзакции

X/Open XA (сокращенное eXtended Architecture) — это стандарт для реализации двухфазной фиксации в гетерогенных технологиях [76, 77]. Он был внедрен в 1991 году и с тех пор широко применяется: XA поддерживается во многих традиционных реляционных базах данных (включая PostgreSQL, MySQL, DB2, SQL Server и Oracle) и брокерах сообщений (таких как ActiveMQ, HornetQ, MSMQ и IBM MQ).

XA не является сетевым протоколом — это просто C API для взаимодействия с координатором транзакций. Привязки к этому API существуют и в других языках; например, в приложениях Java EE XA-транзакции реализованы с использованием Java Transaction API (JTA), который, в свою очередь, поддерживается многими драйверами баз данных с помощью Java Database Connectivity (JDBC) и драйверами брокеров сообщений благодаря применению API Java Message Service (JMS).

XA предполагает, что для связи с базами данных или сервисами обмена сообщениями участников приложение использует сетевой драйвер или клиентскую библиотеку. Если драйвер поддерживает XA, то это значит, что он вызывает XA API для определения, является ли операция частью распределенной транзакции, и если да, то отправляет необходимую информацию на сервер базы. Драйвер также предоставляет функции обратного вызова, с помощью которых координатор будет отправлять участнику запросы на подготовку, завершение или прерывание транзакции.

Координатор транзакций реализует XA API. В стандарте не указывается, как именно это должно быть сделано, но на практике координатор часто представляет собой обычную библиотеку. Она загружается в тот же процесс, что и приложение, инициирующее транзакцию (а не в виде отдельного сервиса). Координатор отслеживает

участников транзакции, собирает их ответы на запросы подготовки (через функцию обратного вызова в драйвере) и использует журнал на локальном диске для отслеживания решений о фиксации или прерывании каждой транзакции.

Если процесс приложения даст сбой или машина, на которой работает приложение, выйдет из строя, то координатор тоже отключается. Все участники с подготовленными, но незавершенными транзакциями остаются в состоянии неопределенности. Поскольку журнал координатора находится на локальном диске сервера приложений, этот сервер необходимо перезапустить, а библиотека координатора должна будет прочитать журнал, чтобы восстановить результат фиксации или прерывания для каждой транзакции. Только тогда координатор сможет использовать обратные вызовы XA для драйверов базы данных с целью отправить участникам запросы на завершение или прерывание транзакции, в зависимости от обстоятельств. Сервер БД не может напрямую связаться с координатором, поскольку все сообщения проходят через клиентскую библиотеку.

Блокировка во время неопределенности

Почему мы так заботимся о том, чтобы транзакция не застряла в состоянии неопределенности? Не может ли остальная часть системы тем временем работать, игнорируя сомнительную транзакцию, пока ее состояние рано или поздно не определится?

Проблема заключается в *блокировке*. Как было показано в подразделе «Чтение зафиксированных данных» раздела 7.2, транзакции базы данных обычно используют эксклюзивную блокировку на уровне строк для любых строк, которые они изменяют, во избежание «грязной» записи. Кроме того, если мы хотим получить сериализуемую изоляцию, то БД с двухфазной блокировкой тоже должна будет применить общую блокировку для любых строк, которые будут *прочитаны* в ходе транзакции (см. подраздел «Двухфазная блокировка (2PL)» раздела 7.3).

База данных не может снять эти блокировки до тех пор, пока транзакция не будет завершена или прервана (заштрихованная область на рис. 9.9). Так что при использовании двухфазной фиксации транзакция должна сохранять блокировки все время, пока находится в состоянии неопределенности. При выходе координатора из строя и повторном включении через 20 минут эти блокировки будут удерживаться в течение 20 минут. Если по какой-то причине журнал координатора будет полностью потерян, то блокировки будут сохраняться вечно или по крайней мере до тех пор, пока администратор не решит проблему вручную.

Пока сохраняются блокировки, никакая другая транзакция не может изменять эти строки. В зависимости от базы данных они могут быть заблокированы для чтения и другими транзакциями. Таким образом, последние не могут просто продолжать свою работу — если им нужен доступ к тем же данным, то они будут заблокированы. Это может привести к тому, что большая часть приложения станет недоступной до тех пор, пока не будет устранена неопределенность.

Восстановление после сбоя координатора

Теоретически если координатор выходит из строя и перезапускается, то он должен полностью восстановить свое состояние по журналу и по нему же определить состояние всех неопределенных транзакций. Однако на практике случаются транзакции, *зависшие* в состоянии неопределенности [89, 90], то есть такие, для которых координатор по какой-то причине не может принять окончательного решения (например, потому, что вследствие ошибки программного обеспечения журнал транзакций был утерян или поврежден). Состояние таких транзакций не может быть определено автоматически, поэтому они отправляются в базу данных на постоянное хранение, удерживая блокировки и блокируя другие транзакции.

Перезагрузка серверов баз данных не устранит указанную проблему, поскольку при правильной реализации 2PC блокировки сомнительных транзакций должны сохраняться даже после перезагрузки (иначе может последовать нарушение гарантии атомарности). Это скользкая ситуация.

Единственный выход из нее — вмешательство администратора, который решит, следует ли завершить или прервать ту или иную транзакцию, и вручную введет соответствующие команды. Он должен изучить участников каждой сомнительной транзакции, определить, успел ли какой-либо из них завершить или прервать транзакцию, и применить этот результат к остальным участникам. Решение проблемы потенциально требует выполнения вручную большого объема работы — скорее всего, в состоянии сильного стресса и нехватки времени во время серьезного сбоя производства (а иначе почему координатор оказался в таком плохом состоянии?).

Во многих реализациях XA есть аварийный выход, называемый *эвристическими решениями*. Он позволяет участнику в одностороннем порядке принять решение о прерывании или завершении сомнительной транзакции без окончательного решения от координатора [76, 77, 91]. Проясню ситуацию: слово «эвристический» здесь эвфемизм, замена для *вероятного нарушения атомарности*, поскольку это действие нарушает систему гарантий при двухфазной фиксации. Таким образом, эвристические решения предназначены только для выхода из катастрофических ситуаций, а вовсе не для регулярного использования.

Ограничения для распределенных транзакций

XA-транзакции решают насущную и важную проблему, связанную с поддержанием нескольких информационных систем-участников. Но, как мы увидели, они также создают серьезные операционные проблемы. В частности, их главная особенность заключается в том, что координатор транзакций сам по себе своего рода база данных (в которой сохраняются результаты транзакций), и поэтому к ней необходимо относиться с такой же тщательностью, как и к любой другой важной базе.

- Если координатор не реплицируется, а работает только на одной машине, то это является возможной точкой отказа всей системы (отказ координатора приведет

к тому, что остальные серверы приложений зафиксируют блокировки, связанные с сомнительными транзакциями). Удивительно, но многие реализации координаторов по умолчанию не являются высоконадежными или имеют лишь простейшую поддержку репликации.

- Многие серверные приложения разрабатываются по модели без учета состояния (в соответствии с HTTP), причем все постоянные состояния хранятся в базе данных. С одной стороны, здесь есть преимущество: серверы приложений можно добавлять и удалять по мере необходимости. Однако если координатор является частью сервера приложений, то меняет характер развертывания. Журналы координатора становятся важной частью долговременного состояния системы — столь же важными, как и сами БД, поскольку необходимы для восстановления сомнительных транзакций после сбоя. Такие серверы приложений больше не являются серверами без фиксации состояния.
- Поскольку XA-транзакции должны быть совместимы с широким спектром систем данных, они обязательно являются наименьшим общим знаменателем. Например, не могут обнаруживать взаимоблокировки в разных системах (так как для этого требуется стандартизованный протокол обмена информацией о блокировках, ожидаемых каждой транзакцией) и не работают с SSI (см. подраздел «Сериализуемая изоляция снимков состояния (SSI)» раздела 7.3), поскольку та требует протокол выявления конфликтов в разных системах.
- Для внутренних распределенных транзакций в базе данных (не XA) ограничения не столь велики, например, возможна распределенная версия SSI. Однако и здесь остается проблема: для успешного завершения транзакции 2PC *все* участники должны дать ответы. Следовательно, если *любая* часть системы нарушена, то транзакция тоже не состоится. Таким образом, распределенные транзакции имеют тенденцию *усугублять* сбои, что противоречит нашей цели построения отказоустойчивых систем.

Означает ли все это, что следует отказаться от надежды на совместимость нескольких систем? Не совсем: существуют альтернативные методы, позволяющие достичь того же результата без проблем, связанных с гетерогенными распределенными транзакциями. Мы вернемся к ним в главах 11 и 12, но прежде закончим тему консенсуса.

Отказоустойчивый консенсус

Если отойти от строгих определений, то консенсус означает достижение согласованности между несколькими узлами по какому-то вопросу. Допустим, несколько человек одновременно пытаются забронировать последнее место в самолете или одно и то же место в театре или зарегистрировать учетную запись с одинаковым именем пользователя. Консенсусный алгоритм позволяет определить, какой из этих несовместимых операций отдать предпочтение.

Задача консенсуса обычно формулируется так: один или несколько узлов *предлагают* значения, а консенсусный алгоритм *выбирает* одно из них. В примере

с бронированием, когда несколько клиентов одновременно пытаются купить последнее место, каждый узел, обрабатывающий запрос клиента, может предложить идентификатор клиента, которого он обслуживает, а в решении будет указано, какой из этих клиентов получил место.

В этой формулировке консенсусный алгоритм должен удовлетворять следующим требованиям [25][1].

- *Единое решение*. Никакие два узла не могут получить разные решения.
- *Целостность*. Ни один узел не получает два решения.
- *Действительность*. Если узел выбирает решение v, то v было предложено другим узлом.
- *Завершенность*. Каждый узел, который не вышел из строя, в конечном итоге выбирает то или иное значение.

Единое решение и целостность определяют основную идею консенсуса: каждый раз выбирается один результат и, как только это произошло, решение не может быть изменено. Свойство действительности существует главным образом для исключения тривиальных решений: например, можно построить алгоритм, который бы всегда выбирал нуль, независимо от того, что было предложено; этот алгоритм будет удовлетворять свойствам единого решения и целостности, но не свойству действительности.

Если не учитывать отказоустойчивость, то удовлетворить первые три требования легко: достаточно жестко назначить один узел «диктатором», и пускай он принимает все решения. Однако при выходе данного узла из строя система больше не сможет принимать какие-либо решения. Фактически именно это мы наблюдали в случае двухфазной фиксации: при неработающем координаторе участники сомнительной транзакции не могут решить, следует ли им ее завершать или прервать.

Свойство завершенности формализует идею отказоустойчивости. По сути, оно говорит, что консенсусный алгоритм не может просто остановиться и ничего не делать — он должен двигаться вперед. Даже если некоторые узлы вышли из строя, другие все равно должны принять решение. (Завершенность — свойство жизнеспособности, в то время как остальные три являются свойствами безопасности, см. пункт «Функциональная безопасность и живучесть» подраздела «Модели системы на практике» раздела 8.4.)

Системная модель консенсуса предполагает, что, когда узел выходит из строя, он внезапно исчезает из сети и больше не возвращается. (В отличие от сбоя программного обеспечения; представьте: произошло землетрясение и ЦОД, содержащий

[1] Данный конкретный вариант консенсуса называется единообразным консенсусом, что эквивалентно регулярному консенсусу в асинхронных системах с ненадежными детекторами отказа [71]. В академической литературе обычно идет речь о процессах, а не об узлах, но мы здесь будем использовать узлы для согласованности с остальной книгой.

ваш узел, разрушен оползнем. Вы должны предположить, что узел погребен под 30 футами грязи и никогда больше не выйдет на связь.) В этой системной модели любой алгоритм, который предлагает ожидать восстановления узла, не сможет удовлетворить требованию завершенности. В частности, 2PC не соответствует данному требованию.

Конечно, если *все* узлы выйдут из строя и ни один из них не сохранит работоспособность, то любой алгоритм бессилен. Существует ограничение по количеству отказов, которые способен выдержать алгоритм: в сущности, можно доказать, что любой консенсусный алгоритм требует правильного функционирования по крайней мере большинства узлов с целью гарантировать завершенность [67]. Это большинство может безопасно сформировать кворум (см. пункт «Операции записи и чтения по кворуму» подраздела «Запись в базу данных при отказе одного из узлов» раздела 5.4).

Таким образом, свойство завершенности основано на предположении, что менее половины узлов вышли из строя или недоступны. Однако большинство реализаций консенсуса гарантируют, что свойства безопасности — согласованность, целостность и достоверность — выполняются всегда, даже если большинство узлов вышли из строя или возникли серьезные неполадки в сети [92]. Таким образом, крупномасштабное отключение может приостановить обработку запросов в системе, но не способно нарушить систему консенсуса, заставив ее принимать недействительные решения.

Большинство консенсусных алгоритмов предполагают, что необъяснимых ошибок не существует (см. подраздел «Византийские сбои» раздела 8.4). Другими словами, если узел неправильно следует протоколу (например, отправляет противоречивые сообщения на разные узлы), то может нарушить защитные свойства последнего. Можно сделать консенсус устойчивым от ошибок такого типа, если им подвержены менее трети узлов [25, 93], но в данной книге не предусмотрено подробное обсуждение этих алгоритмов.

Консенсусные алгоритмы и рассылка общей последовательности

Наиболее известными отказоустойчивыми консенсусными алгоритмами являются Viewstamped Replication (VSR) [94, 95], Paxos [96–99], Raft [22, 100, 101] и Zab [15, 21, 102]. Между ними довольно много общего, но они не одинаковы [103]. В данной книге мы не будем подробно описывать различные алгоритмы: достаточно знать некоторые основополагающие общие идеи, если только вы не намерены самостоятельно построить консенсусную систему (что, вероятно, нецелесообразно — это слишком сложно [98, 104]).

Большинство упомянутых алгоритмов фактически не используют непосредственно описанную здесь формальную модель (предлагая и определяя единственное значение, удовлетворяющее условиям согласованности, целостности, действительности и завершенности). Вместо этого они принимают решение о *последовательности*

значений, что делает их алгоритмами *рассылки общей последовательности*, описанными ранее в данной главе (см. подраздел «Рассылка общей последовательности» раздела 9.3).

Напомню: для такой рассылки требуется, чтобы сообщения были доставлены всем узлам ровно один раз, в одном и том же порядке. Если вдуматься, то это эквивалентно выполнению нескольких циклов консенсуса: при каждом проходе узлы предлагают сообщение, которое хотят передать дальше, а затем принимают решение о следующем сообщении, которое должно быть доставлено в порядке общей последовательности [67].

Таким образом, рассылка общей последовательности эквивалентна нескольким циклам консенсуса (каждое принятое решение соответствует доставке одного сообщения).

- В соответствии с требованием согласованности все узлы решают доставлять одни и те же сообщения, в одном и том же порядке.
- В соответствии с требованием целостности сообщения не дублируются.
- В соответствии с требованием действительности сообщения не повреждаются и не появляются «из ниоткуда».
- В соответствии с требованием завершенности сообщения не теряются.

Viewstamped Replication, Raft и Zab реализуют рассылку общей последовательности напрямую, потому что это более эффективно, чем повторение циклов согласованности. В Paxos данная оптимизация известна как Multi-Paxos.

Репликация с одним ведущим узлом и консенсус

В главе 5 обсуждалась репликация с одним ведущим узлом (см. раздел 5.1), при которой все операции записи направляются ведущему узлу, а он передает их ведомым в том же порядке, тем самым сохраняя актуальность реплик. Разве это не то же самое, что рассылка общей последовательности? Почему в главе 5 нам не пришлось беспокоиться о консенсусе?

Ответ сводится к выбору ведущего узла. Если он выбирается вручную и настраивается людьми из операционного отдела, то, по существу, у вас уже есть «консенсусный алгоритм» диктаторского типа: только один узел может принимать операции записи (принимать решения о порядке внесения записей в журнал репликации). И при выходе этого узла из строя система станет недоступной для записи, пока операторы вручную не назначат ведущим другой узел. На практике такая система может хорошо работать, но она не удовлетворяет условию консенсуса, поскольку для ее работы требуется вмешательство человека.

В некоторых базах данных выполняется автоматический выбор ведущего узла и переход на другой ресурс: если старый ведущий выходит из строя, то новым становится один из ведомых узлов (см. подраздел «Перебои в обслуживании узлов»

раздела 5.1). Это приближает нас к отказоустойчивости рассылки общей последовательности и, соответственно, к достижению консенсуса.

Однако здесь есть некая трудность. Обсуждая ранее проблему разделения интеллекта, мы отмечали, что все узлы должны достичь согласованности в вопросе о том, кто является ведущим, иначе два узла могут одновременно считать себя таковыми и, следовательно, привести базу данных в состояние неопределенности. Исходя из вышесказанного для того чтобы выбрать ведущий узел, нужен консенсус. Но если описанные здесь консенсусные алгоритмы на самом деле являются алгоритмами рассылки общей последовательности, а такая рассылка подобна репликации с одним ведущим узлом, а для этой репликации требуется ведущий узел, то...

Похоже, для выбора ведущего узла нам сначала нужен такой узел. Чтобы достичь консенсуса, надо прежде достичь консенсуса. Как вырваться из этого заколдованного круга?

Нумерация периодов и кворумы

Все консенсусные протоколы, которые мы обсудили, основаны на использовании ведущего узла в той или иной форме, но не гарантируют, что он уникален. Вместо этого они предлагают более слабую гарантию: протоколы определяют *номер периода* (в Paxos называемый *номером бюллетеня*, в Viewstamped Replication — *номером просмотра,* в Raft — *номером термина*) и гарантируют, что в каждый период ведущий узел уникален.

Каждый раз, когда становится ясно, что действующий ведущий узел вышел из строя, между узлами начинается голосование с целью выбрать новый. Каждым таким выборам присваивается порядковый номер периода. Таким образом, все номера периодов полностью упорядочены и монотонно возрастают. В случае конфликта между двумя ведущими узлами, принадлежащими разным периодам (вероятно, потому, что впоследствии выяснится: в действительности предыдущий ведущий не выходил из строя), преимущество имеет ведущий узел с более высоким номером периода.

Прежде чем ведущий узел получает право принимать решения, он должен проверить, нет ли другого такого узла с более высоким номером периода, который мог бы принять другое решение. Откуда ведущий знает, что не был замещен другим узлом? В подразделе «Истина определяется большинством» раздела 8.4 было показано: узел не всегда может доверять собственному суждению. То обстоятельство, что он считает себя ведущим, еще не означает его принятия в таком качестве другими узлами.

Вместо этого он должен собирать голоса *кворума* узлов (см. пункт «Операции записи и чтения по кворуму» подраздела «Запись в базу данных при отказе одного из узлов» раздела 5.4). Для каждого решения, которое ведущий узел намерен принять, он должен отправить предлагаемое значение другим узлам и дождаться,

пока кворум узлов согласится с этим предложением. Обычно (но не всегда) кворум состоит из большинства узлов [105]. Узел голосует за предложение только в том случае, если ему неизвестен другой ведущий с более высоким номером периода.

Таким образом, есть два раунда голосования: один — чтобы выбрать ведущий узел, и второй — чтобы проголосовать за предложение его кандидатуры. Ключевая идея заключается в том, что кворумы для этих двух голосований должны дублироваться: при успешном голосовании за предложение ведущего по крайней мере один из узлов, голосовавших за него, должен был также принять участие в выборах действующего ведущего [105]. Таким образом, если в голосовании по предложению не был указан более высокий номер периода, то действующий ведущий узел может прийти к заключению: выборов ведущего с более высоким номером периода не было — и на данном основании сделать вывод, что он по-прежнему является ведущим и поэтому может спокойно принять предложенное значение.

На первый взгляд такой процесс голосования выглядит подобно двухфазной фиксации. Самые большие различия заключаются в том, что в 2PC координатор не избирается и отказоустойчивые консенсусные алгоритмы требуют голосов только от большинства узлов, тогда как в 2PC требуется ответ «да» от каждого участника. Более того, консенсусные алгоритмы предусматривают процесс восстановления, с помощью которого узлы могут перейти в согласованное состояние после выбора нового ведущего, с гарантией выполнения требований безопасности. Эти различия являются ключевой особенностью, обеспечивающей корректность и отказоустойчивость консенсусного алгоритма.

Ограничения консенсуса

Алгоритмы консенсуса являются огромным прорывом для распределенных систем: они привносят конкретные свойства безопасности (согласованность, целостность, достоверность) в системы, где все остальное неопределенно, и тем не менее они остаются отказоустойчивыми (способны продолжать работать, если большинство узлов работают и доступны по сети). Они обеспечивают рассылку общей последовательности и поэтому могут также выполнять линеаризуемые атомарные операции с сохранением отказоустойчивости (см. пункт «Реализация линеаризуемого хранилища с помощью рассылки общей последовательности» подраздела «Рассылка общей последовательности» раздела 9.3).

Тем не менее эти алгоритмы не применяются повсеместно, потому что за их преимущества приходится расплачиваться.

Процесс, с помощью которого узлы голосуют за предложение до его принятия, является своего рода синхронной репликацией. Как обсуждалось в подразделе «Синхронная и асинхронная репликация» раздела 5.1, базы данных часто настраиваются для использования асинхронной репликации. В такой конфигурации некоторые переданные данные могут быть потеряны при отказе, но часто с этим риском предпочитают мириться ради более высокой производительности.

Консенсусные системы всегда требуют строгого большинства. Это значит следующее: нужно минимум три узла, чтобы выдержать один сбой (оставшиеся два из трех составят большинство) или минимум пять узлов, чтобы выдержать два (оставшиеся три из пяти составят большинство). Если в результате сетевого сбоя некоторые узлы окажутся отключенными от остальных, то только основная часть сети сможет продолжать работу, а остальная будет заблокирована (см. также подраздел «Цена линеаризуемости» раздела 9.2).

Большинство консенсусных алгоритмов предполагает, что в голосовании участвует фиксированный набор узлов. То есть нельзя просто добавить или удалить узел в кластере. Расширения *динамического членства* в консенсусных алгоритмах позволяют изменять набор узлов в кластере, но они гораздо менее понятны, чем алгоритмы статического членства.

Для обнаружения узлов, вышедших из строя, консенсусные системы обычно полагаются на время ожидания. В системах с сильно различающимися сетевыми задержками, особенно в географически распределенных, часто бывает так, что узел ошибочно предполагает, будто ведущий вышел из строя, в то время как в действительности причина в задержках переходной сети. Данная ошибка не нарушает свойства безопасности, но частые выборы ведущего узла приводят к ужасающей производительности: система может в итоге тратить на это больше времени, чем на какую-либо полезную работу.

Иногда консенсусные алгоритмы особенно чувствительны к сетевым проблемам. Например, у Raft были неприятные пограничные случаи [106]: если вся сеть работает правильно, за исключением одной сетевой линии, которая постоянно ненадежна, то Raft может попасть в ситуацию, когда руководящая роль непрерывно кочует между двумя узлами или действующий ведущий постоянно вынужден уйти в отставку, поэтому система практически не выполняет полезной работы. Схожие проблемы наблюдаются и в других консенсусных алгоритмах. Построение более надежных алгоритмов для ненадежных сетей по-прежнему остается открытой исследовательской задачей.

Сервисы членства и координации

Проекты, подобные ZooKeeper или etcd, часто описываются как «распределенные хранилища типа “ключ — значение”» или «сервисы координации и настройки». API такого сервиса очень похож на базу данных: можно читать и записывать значения по заданному ключу и выполнять операции последовательно для набора ключей. Но если это, в сущности, БД, то зачем прилагать столь большие усилия для реализации консенсусных алгоритмов? В чем их отличия от любой другой базы?

Чтобы понять это, полезно вкратце рассмотреть, как используются сервисы, подобные ZooKeeper. Как разработчикам приложений, вам редко придется задействовать ZooKeeper напрямую, поскольку данная система, вообще-то, не очень

хорошо подходит для применения в качестве БД общего назначения. Скорее всего, вы будете опираться на нее косвенно, через какой-то другой проект. Например, HBase, Hadoop YARN, OpenStack Nova и Kafka полагаются на систему ZooKeeper, работающую в фоновом режиме. Какую выгоду они из нее извлекают?

ZooKeeper и etcd предназначены для хранения небольших объемов данных, которые полностью помещаются в памяти (хотя и записываются на диск для обеспечения долговечности), поэтому вы не захотите хранить в них все данные приложения. Этот небольшой объем реплицируется по всем узлам с помощью отказоустойчивого алгоритма рассылки общей последовательности. Как обсуждалось ранее, рассылка общей последовательности — именно то, что нужно для репликации базы данных: если каждое сообщение представляет собой запись в базу, то применение одинаковых записей в том же порядке сохраняет согласованность реплик.

Система ZooKeeper смоделирована после сервиса блокировки Google Chubby [14, 98]. В ней реализована не только рассылка общей последовательности (и, следовательно, консенсус), но и интересный набор других функций, которые оказываются особенно полезными при создании распределенных систем.

- *Линеаризуемые атомарные операции*. Используя атомарную операцию сравнения с присвоением, можно реализовать блокировку: если несколько узлов одновременно пытаются выполнить одну и ту же операцию, то лишь одна из них будет успешной. Консенсусный протокол гарантирует, что операция будет атомарной и линеаризуемой, даже если узел выйдет из строя или сетевое соединение прервется в любой точке. Распределенная блокировка обычно реализуется как *аренда*, у которой есть срок истечения, поэтому в случае отказа клиента она в итоге будет завершена (см. подраздел «Паузы при выполнении процессов» раздела 8.3).
- *Общая последовательность операций*. Как обсуждалось в пункте «Ведущий узел и блокировки» подраздела «Истина определяется большинством» раздела 8.4, когда какой-либо ресурс защищен с помощью блокировки или аренды, нужен маркер-ограничитель, чтобы в случае паузы процесса клиенты не конфликтовали друг с другом. Маркер-ограничитель — некое число, которое монотонно возрастает при каждой блокировке. В ZooKeeper это обеспечивается за счет полного упорядочения всех операций и предоставления каждой операции монотонно возрастающего идентификатора транзакции (`zxid`) и номера версии (`cversion`) [15].
- *Обнаружение сбоев*. Клиенты поддерживают длительные сеансы на серверах ZooKeeper, а клиент и сервер периодически обмениваются тактами, чтобы убедиться в текущей деятельности другого узла. Даже в случае временного прерывания соединения или выхода из строя узла ZooKeeper сеанс остается активным. Однако если передача тактов прекращается на время, превышающее время ожидания сеанса, ZooKeeper объявляет сеанс прерванным. Любые блокировки, удерживаемые сеансом, могут быть настроены так, чтобы автоматически освобождаться по окончании сеанса (ZooKeeper называет их *эфемерными узлами*).

- *Изменение уведомлений*. Клиент может не только считывать блокировки и значения, созданные другим клиентом, но и следить за изменениями в других узлах. Таким образом, клиент может узнать, что другой клиент присоединился к кластеру (на основе значения, которое он записывает в ZooKeeper) или вышел из строя (поскольку время его сеанса истекло и его эфемерные узлы исчезли). Благодаря подписке на уведомления клиенту не нужно совершать частые опросы, чтобы узнать об изменениях.

Из этих особенностей только линеаризуемые атомарные операции действительно требуют консенсуса. Однако сочетание описанных функций делает такие системы, как ZooKeeper, полезными для координации в распределенных средах.

Распределение нагрузки между узлами

Пример ситуации, где модель ZooKeeper/Chubby эффективна, — это наличие нескольких экземпляров процесса или сервиса, один из которых должен быть выбран в качестве ведущего или первичного. Если ведущий выходит из строя, то один из оставшихся узлов должен занять его место. Это полезно не только для баз данных с одним ведущим, но и для планировщиков задач и других подобных систем с сохранением состояния.

Другой пример — когда есть некий секционированный ресурс (база данных, потоки сообщений, хранилище файлов, распределенная система акторов и т. п.), и нужно решить, какому узлу будет назначен тот или иной раздел. По мере присоединения к кластеру новых узлов некоторые разделы потребуется перенести из существующих узлов в новые, чтобы равномерно распределить нагрузку (см. раздел 6.4). Когда узлы удаляются или выходят из строя, другим необходимо взять на себя их нагрузку.

Задачи такого типа могут быть решены с помощью разумного использования атомарных операций, эфемерных узлов и уведомлений в ZooKeeper. Если все сделано правильно, то данный подход позволяет приложению автоматически восстанавливаться после сбоев без вмешательства человека. Несмотря на появление таких библиотек, как Apache Curator [17], созданных для предоставления более высокоуровневых инструментов на основе клиентского API ZooKeeper, это все еще непросто. Но все же намного лучше, чем пытаться реализовать с нуля необходимые консенсусные алгоритмы, у которых, как правило, низкий процент успешности [107].

Приложение может вначале запускаться всего на одном узле, но впоследствии вырасти до нескольких тысяч. Попытка получить большинство голосов на таком количестве узлов была бы в высшей степени неэффективной. Вместо этого ZooKeeper работает на их фиксированном количестве (обычно трех или пяти) и получает львиную долю голосов для этих узлов, поддерживая потенциально большое количество клиентов. Таким образом, ZooKeeper предоставляет способ «аутсорсинга» некоторой части координационных узлов (консенсуса, последовательности операций и обнаружения сбоев) для внешнего сервиса.

Как правило, данные, управляемые ZooKeeper, меняются довольно медленно, эта система представляет информацию примерно в таком виде: «Узел, запущенный по адресу 10.1.1.23, является ведущим для раздела 7», и она может меняться течение нескольких минут или часов. Такая система не предназначена для хранения состояния приложения, которое способно меняться тысячи или даже миллионы раз в секунду. Если необходимо реплицировать состояние приложения из одного узла в другой, то можно воспользоваться другими инструментами (такими как Apache BookKeeper [108]).

Обнаружение сервиса

ZooKeeper, etcd и Consul также часто используются для *обнаружения сервисов*, то есть выяснения того, к какому IP-адресу необходимо подключиться, чтобы получить доступ к тому или иному сервису. В облачных ЦОДах, где виртуальные машины постоянно приходят и уходят, IP-адрес сервисов часто заранее неизвестен. Вместо этого можно настроить сервисы таким образом, чтобы при запуске они регистрировали свои конечные точки сети в реестре, где затем другие сервисы смогут их найти.

Тем не менее неясно, действительно ли обнаружение сервиса требует консенсуса. DNS, традиционный способ поиска IP-адреса по имени сервиса, для обеспечения хорошей производительности и доступности использует несколько уровней кэширования. Операции чтения из DNS абсолютно нелинеаризуемы, и обычно не считается проблемой, если результаты DNS-запроса немного устарели [109]. Важнее, чтобы DNS был надежно доступен, в том числе и при сетевых сбоях.

Для обнаружения сервиса не требуется консенсус, но нужны выборы ведущего узла. Таким образом, если в консенсусной системе ведущий уже известен, имеет смысл использовать данную информацию, чтобы помочь другим сервисам узнать об этом. С этой целью некоторые консенсусные системы поддерживают защищенные от записи реплики кэширования. Последние асинхронно получают журнал всех решений, принятых консенсусным алгоритмом, но не принимают активного участия в голосовании. Как следствие, они могут обслуживать запросы чтения, не требующие линеаризуемости.

Сервисы членства

ZooKeeper и подобные системы можно рассматривать как часть долгой истории исследований *сервисов членства*, восходящей к 1980-м. Эти исследования имели большое значение для создания высоконадежных систем, например, таких, которые используются для управления воздушным движением [110].

Сервисы членства определяет, какие узлы в настоящее время активны и являются действующими членами кластера. Как было показано в главе 8, из-за неограниченных сетевых задержек невозможно надежно определить, не вышел ли узел из строя.

Однако если соединить обнаружение сбоев с консенсусом, то узлы могут прийти к соглашению о том, какие из них следует считать действующими, а какие — вышедшими из строя.

По-прежнему может случиться так, что узел будет ошибочно объявлен вышедшим из строя на основе консенсуса, хотя в действительности он работоспособен. Но тем не менее очень полезно наличие в системе соглашения о том, какие узлы составляют действующее членство. Например, выбор ведущего может означать просто выбор наименьшего количества среди действующих членов, но этот метод не будет работать, если разные узлы имеют разные мнения о том, кто является таковым.

9.5. Резюме

В этой главе с разных точек зрения были рассмотрены темы согласованности и консенсуса. Мы подробно изучили линеаризуемость — популярную модель согласованности: ее цель состоит в том, чтобы реплицированные данные выглядели так, как будто существует только одна копия и все операции воздействуют на нее атомарно. Несмотря на привлекательность линеаризуемости, поскольку ее легко понять, она тем не менее приводит к тому, что база данных ведет себя как переменная в однопоточной программе — ее недостаток заключается в замедлении работы системы, особенно в средах с большими сетевыми задержками.

Мы также исследовали причинность, которая требует соблюдения последовательности событий в системе (что произошло раньше, а что позже, на основании причинно-следственных взаимосвязей). В отличие от линеаризуемости, выстраивающей все операции в единую, полностью упорядоченную временную последовательность, причинность позволяет построить более слабую модель согласованности: отдельные события могут быть конкурентными, как в истории версий с ее ветвлениями и слияниями. Причинная согласованность не несет накладных расходов на линеаризуемость и гораздо менее чувствительна к сетевым проблемам.

Однако даже если зафиксировать причинно-следственный порядок (например, с помощью временных меток Лампорта), то, как мы увидели, все равно остаются системы, которые не могут быть реализованы таким образом: в пункте «Временных меток недостаточно» подраздела «Упорядоченность по порядковым номерам» раздела 9.3 был представлен пример с обеспечением уникальности имени пользователя и отклонением конкурентных попыток регистрации под одним и тем же именем. Если один узел собирается принять регистрацию, то должен каким-то образом узнать, что другой не пытается в это же время зарегистрировать такое же имя. Данная проблема привела нас к идее *консенсуса*.

Мы увидели: достичь консенсуса означает принять решение о чем-то таким образом, чтобы с этим решением согласились все узлы и данное решение являлось неотменяемым. При ближайшем рассмотрении выяснилось: весь широкий спектр

проблем фактически сводится к консенсусу, они эквивалентны (в том смысле, что если есть решение для одной из них, то его можно легко превратить в таковое для остальных). К таким эквивалентным проблемам относятся следующие.

- *Линейные реестры сравнения с присвоением*. Реестр должен атомарно *принять решение*, присваивать ли значение в зависимости от того, соответствует ли его текущее значение параметру, указанному в операции.
- *Атомарная транзакция*. База данных должна *принять решение*, следует ли завершать или отменить распределенную транзакцию.
- *Рассылка общей последовательности*. Система обмена сообщениями должна *принять решение* о последовательности доставки сообщений.
- *Блокировки и аренда*. Когда несколько клиентов ратуют за блокировку или ее отмену, блокировка *принимает решение* о том, какой из них выбрать.
- *Сервис членства и координации*. Основываясь на детекторе отказа (например, времени задержки), система должна *принять решение*, какие узлы активны, а какие следует считать вышедшими из строя, потому что их сеансы были отключены.
- *Ограничение уникальности*. Когда несколько транзакций конкурентно пытаются создать конфликтующие записи с одним и тем же ключом, ограничение должно *принять решение*, какую из них разрешить, а какие — отменить по причине нарушения ограничения.

Все описанное очень просто, если узел только один или в случае, когда только один узел имеет возможность принимать решения. Именно это происходит в базе данных с одним ведущим узлом: на него возложены полномочия по принятию решений. Поэтому такие базы могут предоставлять линеаризуемые операции, ограничения уникальности, полностью упорядоченный журнал репликации и др.

Однако если этот единственный ведущий узел выходит из строя или становится недоступным вследствие разрыва сетевого соединения, то такая система становится неработоспособной. Существует три способа выхода из данной ситуации.

1. Дождаться, пока ведущий узел восстановится, и согласиться с тем, что все это время система будет заблокирована. Многие координаторы XA/JTA-транзакций выбирают именно такой вариант. Данный метод не полностью решает задачу консенсуса, поскольку не удовлетворяет требованию завершения: если ведущий узел не восстановится, то система может оказаться заблокированной навсегда.
2. Решить проблему вручную — пускай человек выберет новый ведущий узел и перенастроит систему для его использования. Такой вариант применяется во многих реляционных базах данных. Это своего рода консенсус методом «вмешательства высшей силы»: решение принимает человек-оператор, находящийся вне компьютерной системы. Скорость разрешения проблемы ограничена той скоростью, с которой могут действовать люди, — как правило, медленнее, чем компьютеры.

3. Использовать алгоритм автоматического выбора нового ведущего узла. Такой вариант требует консенсусного алгоритма, и рекомендуется задействовать проверенный алгоритм, который правильно обрабатывает неблагоприятные сетевые условия [107].

Несмотря на то что база данных с одним ведущим узлом способна обеспечить линеаризуемость без использования консенсусного алгоритма для каждой записи, она по-прежнему требует консенсуса для поддержания приоритета и выборов ведущего узла. Таким образом, в некотором смысле наличие этого узла — лишь полумера: консенсус по-прежнему необходим, только в другом месте и реже. Хорошая новость — существуют отказоустойчивые консенсусные алгоритмы и системы, и мы кратко обсудили их в данной главе.

Такие инструменты, как ZooKeeper, играют важную роль в обеспечении «аутсорсинга» консенсуса, обнаружении сбоев и обслуживании членства, которое могут задействовать приложения. Данная система непроста в применении, но это намного лучше, чем пытаться разработать собственные алгоритмы для решения всех проблем, описанных в главе 8. Если однажды вы захотите решить одну из задач, которые сводятся к консенсусу, и сделать решение отказоустойчивым, то советую использовать что-то наподобие ZooKeeper.

Однако не каждая система обязательно требует консенсуса: например, системы репликации без ведущего узла и системы с несколькими такими узлами обычно не используют глобальный консенсус. Конфликты, возникающие в подобных системах (см. «Обработка конфликтов записи» раздела 5.3), являются следствием отсутствия консенсуса между разными ведущими узлами. Но, может быть, это нормально: вероятно, нужно лишь научиться обходиться без линеаризуемости и лучше работать со структурами данных, история которых имеет ветвления с последующим слиянием версий.

В этой главе даны ссылки на обширные исследования по теории распределенных систем. Теоретические статьи и доказательства не всегда легко понять, а иногда в них делаются нереалистичные предположения. Однако они невероятно ценны, поскольку позволяют быть в курсе практических разработок в данной области: помогают рассуждать о том, что можно и чего нельзя сделать, и обнаружить неочевидные области, в которых применение распределенных систем часто бывает ошибочным. Если у вас есть время, то можете уделить его этим ссылкам — они заслуживают изучения.

Мы пришли к концу части II этой книги, в которой рассмотрели задачи репликации (глава 5), секционирование (глава 6), транзакции (глава 7), модели сбоев в распределенных системах (глава 8) и, наконец, согласованность и консенсус (глава 9). Теперь, заложив прочную теоретическую основу, в части III мы снова обратимся к более практическим системам и обсудим, как создавать эффективные приложения из гетерогенных блоков.

9.6. Библиография

1. *Bailis P., Ghodsi A.* Eventual Consistency Today: Limitations, Extensions, and Beyond // ACM Queue, volume 11, number 3, pages 55–63, March 2013 [Электронный ресурс]. — Режим доступа: http://queue.acm.org/detail.cfm?id=2462076.
2. *Mahajan P., Alvisi L., Dahlin M.* Consistency, Availability, and Convergence // University of Texas at Austin, Department of Computer Science, Tech Report UTCS TR-11-22, May 2011 [Электронный ресурс]. — Режим доступа: https://apps.cs.utexas.edu/tech_reports/reports/tr/TR-2036.pdf.
3. *Scotti A.* Adventures in Building Your Own Database // All Your Base, November 2015 [Электронный ресурс]. — Режим доступа: https://www.slideshare.net/AlexScotti1/allyourbase-55212398.
4. *Bailis P., Davidson A., Fekete A., et al.* Highly Available Transactions: Virtues and Limitations // 40th International Conference on Very Large Data Bases (VLDB), September 2014 [Электронный ресурс]. — Режим доступа: https://arxiv.org/pdf/1302.0309.pdf.
5. *Viotti P., Vukolić M.* Consistency in Non-Transactional Distributed Storage Systems. 12 April 2016 [Электронный ресурс]. — Режим доступа: https://arxiv.org/abs/1512.00168.
6. *Herlihy M. P., Wing J. M.* Linearizability: A Correctness Condition for Concurrent Objects // ACM Transactions on Programming Languages and Systems (TOPLAS), volume 12, number 3, pages 463–492, July 1990 [Электронный ресурс]. — Режим доступа: http://cs.brown.edu/~mph/HerlihyW90/p463-herlihy.pdf.
7. *Lamport L.* On interprocess communication // Distributed Computing, volume 1, number 2, pages 77–101, June 1986 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/interprocess-communication-part-basic-formalism-part-ii-algorithms/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Flamport%2Fpubs%2Finterprocess.pdf.
8. *Gifford D. K.* Information Storage in a Decentralized Computer System // Xerox Palo Alto Research Centers, CSL-81-8, June 1981 [Электронный ресурс]. — Режим доступа: http://www.mirrorservice.org/sites/www.bitsavers.org/pdf/xerox/parc/techReports/CSL-81-8_Information_Storage_in_a_Decentralized_Computer_System.pdf.
9. *Kleppmann M.* Please Stop Calling Databases CP or AP. May 11, 2015 [Электронный ресурс]. — Режим доступа: http://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html.
10. *Kingsbury K.* Call Me Maybe: MongoDB Stale Reads. April 20, 2015 [Электронный ресурс]. — Режим доступа: https://aphyr.com/posts/322-call-me-maybe-mongodb-stale-reads.

11. *Kingsbury K.* Computational Techniques in Knossos. May 17, 2014 [Электронный ресурс]. — Режим доступа: https://aphyr.com/posts/314-computational-techniques-in-knossos.
12. *Bailis P.* Linearizability Versus Serializability. September 24, 2014 [Электронный ресурс]. — Режим доступа: http://www.bailis.org/blog/linearizability-versus-serializability/.
13. *Bernstein P. A., Hadzilacos V., Goodman N.* Concurrency Control and Recovery in Database Systems. — Addison-Wesley, 1987.
14. *Burrows M.* The Chubby Lock Service for Loosely-Coupled Distributed Systems // 7th USENIX Symposium on Operating System Design and Implementation (OSDI), November 2006 [Электронный ресурс]. — Режим доступа: https://research.google.com/archive/chubby.html.
15. *Junqueira F. P., Reed B. (ZooKeeper)* Distributed Process Coordination. — O'Reilly Media, 2013.
16. etcd 2.0.12 Documentation // CoreOS, Inc., 2015 [Электронный ресурс]. — Режим доступа: https://coreos.com/etcd/docs/2.0.12/.
17. Apache Curator // Apache Software Foundation, 2015 [Электронный ресурс]. — Режим доступа: http://curator.apache.org/.
18. *Vallath M.* Oracle 10g RAC Grid, Services & Clustering. Elsevier Digital Press, 2006.
19. *Bailis P., Fekete A., Franklin M. J., et al.* Coordination-Avoiding Database Systems // Proceedings of the VLDB Endowment, volume 8, number 3, pages 185–196, November 2014 [Электронный ресурс]. — Режим доступа: https://arxiv.org/pdf/1402.2237.pdf.
20. *Kingsbury K.* Call Me Maybe: etcd and Consul. June 9, 2014 [Электронный ресурс]. — Режим доступа: https://aphyr.com/posts/316-call-me-maybe-etcd-and-consul.
21. *Junqueira F. P., Reed B. C., Serafini M.* Zab: HighPerformance Broadcast for Primary-Backup Systems // 41st IEEE International Conference on Dependable Systems and Networks (DSN), June 2011 [Электронный ресурс]. — Режим доступа: https://pdfs.semanticscholar.org/b02c/6b00bd5dbdbd951fddb00b906c82fa80f0b3.pdf.
22. *Ongaro D., Ousterhout J. K.* In Search of an Understandable Consensus Algorithm (Extended Version) // USENIX Annual Technical Conference (ATC), June 2014 [Электронный ресурс]. — Режим доступа: https://pdfs.semanticscholar.org/b02c/6b00bd5dbdbd951fddb00b906c82fa80f0b3.pdf.
23. *Attiya H., Bar-Noy A., Dolev D.* Sharing Memory Robustly in Message-Passing Systems // Journal of the ACM, volume 42, number 1, pages 124–142, January 1995 [Электронный ресурс]. — Режим доступа: http://www.cse.huji.ac.il/course/2004/dist/p124-attiya.pdf.
24. *Lynch N., Shvartsman A.* Robust Emulation of Shared Memory Using Dynamic Quorum-Acknowledged Broadcasts // 27th Annual International Symposium on Fault-Tolerant Computing (FTCS), June 1997 [Электронный ресурс]. — Режим доступа: http://groups.csail.mit.edu/tds/papers/Lynch/FTCS97.pdf.

25. *Cachin C., Guerraoui R., Rodrigues L.* Introduction to Reliable and Secure Distributed Programming, 2nd edition. — Springer, 2011 [Электронный ресурс]. — Режим доступа: http://www.distributedprogramming.net/.
26. *Elliott S., Allen M., Kleppmann M.* personal communication, thread on twitter.com, October 15, 2015 [Электронный ресурс]. — Режим доступа: https://twitter.com/lenary?protected_redirect=true.
27. *Ekström N., Panchenko M., Ellis J.* Possible Issue with Read Repair? // email thread on cassandra-dev mailing list, October 2012 [Электронный ресурс]. — Режим доступа: http://mail-archives.apache.org/mod_mbox/cassandra-dev/201210.mbox/%3CFA480D1DC3964E2C8B0A14E0880094C9%40Robotech%3E.
28. *Herlihy M. P.* Wait-Free Synchronization // ACM Transactions on Programming Languages and Systems (TOPLAS), volume 13, number 1, pages 124–149, Janury 1991 [Электронный ресурс]. — Режим доступа: https://cs.brown.edu/~mph/Herlihy91/p124-herlihy.pdf.
29. *Fox A., Brewer E. A.* Harvest, Yield, and Scalable Tolerant Systems // 7th Workshop on Hot Topics in Operating Systems (HotOS), March 1999 [Электронный ресурс]. — Режим доступа: https://radlab.cs.berkeley.edu/people/fox/static/pubs/pdf/c18.pdf.
30. *Gilbert S., Lynch N.* Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services // ACM SIGACT News, volume 33, number 2, pages 51–59, June 2002 [Электронный ресурс]. — Режим доступа: http://www.comp.nus.edu.sg/~gilbert/pubs/BrewersConjecture-SigAct.pdf.
31. *Gilbert S., Lynch N.* Perspectives on the CAP Theorem // IEEE Computer Magazine, volume 45, number 2, pages 30–36, February 2012 [Электронный ресурс]. — Режим доступа: http://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf.
32. *Brewer E. A.* CAP Twelve Years Later: How the 'Rules' Have Changed // IEEE Computer Magazine, volume 45, number 2, pages 23–29, February 2012 [Электронный ресурс]. — Режим доступа: http://cs609.cs.ua.edu/CAP12.pdf.
33. *Davidson S. B., Garcia-Molina H., Skeen D.* Consistency in Partitioned Networks // ACM Computing Surveys, volume 17, number 3, pages 341–370, September 1985 [Электронный ресурс]. — Режим доступа: http://delab.csd.auth.gr/~dimitris/courses/mpc_fall05/papers/invalidation/acm_csur85_partitioned_network_consistency.pdf.
34. *Johnson P. R., Thomas R. H.* RFC 677: The Maintenance of Duplicate Databases // Network Working Group, January 27, 1975 [Электронный ресурс]. — Режим доступа: https://tools.ietf.org/html/rfc677.
35. *Lindsay B. G., Selinger P. G., Galtieri C., et al.* Notes on Distributed Databases // IBM Research, Research Report RJ2571(33471), July 1979 [Электронный ресурс]. — Режим доступа: http://domino.research.ibm.com/library/cyberdig.nsf/papers/A776EC17FC2FCE73852579F100578964/$File/RJ2571.pdf.
36. *Fischer M. J., Michael A.* Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network // 1st ACM Symposium on Principles of Database Systems (PODS), March 1982 [Электронный ресурс]. — Режим доступа: http://www.cs.ucsb.edu/~agrawal/spring2011/ugrad/p70-fischer.pdf.

37. *Brewer E. A.* NoSQL: Past, Present, Future // QCon San Francisco, November 2012 [Электронный ресурс]. — Режим доступа: https://www.infoq.com/presentations/NoSQL-History.
38. *Robinson H.* CAP Confusion: Problems with 'Partition Tolerance // blog.cloudera.com, April 26, 2010 [Электронный ресурс]. — Режим доступа: http://blog.cloudera.com/blog/2010/04/cap-confusion-problems-with-partition-tolerance/.
39. *Cockcroft A.* Migrating to Microservices // QCon London, March 2014 [Электронный ресурс]. — Режим доступа: https://www.infoq.com/presentations/migration-cloud-native.
40. *Kleppmann M.* A Critique of the CAP Theorem. September 17, 2015 [Электронный ресурс]. — Режим доступа: https://arxiv.org/abs/1509.05393.
41. *Lynch N. A.* A Hundred Impossibility Proofs for Distributed Computing // 8th ACM Symposium on Principles of Distributed Computing (PODC), August 1989 [Электронный ресурс]. — Режим доступа: http://groups.csail.mit.edu/tds/papers/Lynch/podc89.pdf.
42. *Attiya H., Ellen F., Morrison A.* Limitations of Highly-Available Eventually-Consistent Data Stores // ACM Symposium on Principles of Distributed Computing (PODC), July 2015 [Электронный ресурс]. — Режим доступа: http://www.cs.technion.ac.il/people/mad/online-publications/podc2015-replds.pdf.
43. *Sewell P., Sarkar S., Owens S., et al.* x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors // Communications of the ACM, volume 53, number 7, pages 89–97, July 2010 [Электронный ресурс]. — Режим доступа: http://www.cl.cam.ac.uk/~pes20/weakmemory/cacm.pdf.
44. *Thompson M.* Memory sympathy.blogspot.co.uk, July 24, 2011. Barriers/Fences. July 24, 2011 [Электронный ресурс]. — Режим доступа: https://mechanical-sympathy.blogspot.com.by/2011/07/memory-barriersfences.html.
45. *Drepper U.* What Every Programmer Should Know About Memory // akkadia.org, November 21, 2007 [Электронный ресурс]. — Режим доступа: https://www.akkadia.org/drepper/cpumemory.pdf.
46. *Abadi D. J.* Consistency Tradeoffs in Modern Distributed Database System Design // IEEE Computer Magazine, volume 45, number 2, pages 37–42, February 2012 [Электронный ресурс]. — Режим доступа: http://cs-www.cs.yale.edu/homes/dna/papers/abadi-pacelc.pdf.
47. *Attiya H., Welch J. L.* Sequential Consistency Versus Linearizability // ACM Transactions on Computer Systems (TOCS), volume 12, number 2, pages 91–122, May 1994 [Электронный ресурс]. — Режим доступа: http://courses.csail.mit.edu/6.852/01/papers/p91-attiya.pdf.
48. *Ahamad M., Neiger G., Burns J. E., et al.* Causal Memory: Definitions, Implementation, and Programming // Distributed Computing, volume 9, number 1, pages 37–49, March 1995 [Электронный ресурс]. — Режим доступа: http://www-i2.informatik.rwth-aachen.de/i2/fileadmin/user_upload/documents/Seminar_MCMM11/Causal_memory_1996.pdf.

49. *Lloyd W., Freedman M. J., Kaminsky M., Andersen D. G.* Stronger Semantics for Low-Latency Geo-Replicated Storage // 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI), April 2013 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final149.pdf.

50. *Zawirski M., Bieniusa A., Balegas V., et al.* SwiftCloud: Fault-Tolerant Geo-Replication Integrated All the Way to the Client Machine // INRIA Research Report 8347, August 2013 [Электронный ресурс]. — Режим доступа: https://arxiv.org/abs/1310.3107.

51. *Bailis P., Ghodsi A., Hellerstein J. M., Stoica I.* Bolt-on Causal Consistency // ACM International Conference on Management of Data (SIGMOD), June 2013 [Электронный ресурс]. — Режим доступа: http://db.cs.berkeley.edu/papers/sigmod13-bolton.pdf.

52. *Ajoux P., Bronson N., Kumar S., et al.* Challenges to Adopting Stronger Consistency at Scale // 15th USENIX Workshop on Hot Topics in Operating Systems (HotOS), May 2015 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/system/files/conference/hotos15/hotos15-paper-ajoux.pdf.

53. *Bailis P.* Causality Is Expensive (and What to Do About It). February 5, 2014 [Электронный ресурс]. — Режим доступа: http://www.bailis.org/blog/causality-is-expensive-and-what-to-do-about-it/.

54. *Gonçalves R., Almeida P. S., Baquero C., Fonte V.* Concise Server-Wide Causality Management for Eventually Consistent Data Stores // 15th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), June 2015 [Электронный ресурс]. — Режим доступа: http://haslab.uminho.pt/tome/files/global_logical_clocks.pdf.

55. *Conery R.* A Better ID Generator for PostgreSQL. May 29, 2014 [Электронный ресурс]. — Режим доступа: http://rob.conery.io/2014/05/28/a-better-id-generator-for-postgresql/.

56. *Lamport L.* Time, Clocks, and the Ordering of Events in a Distributed System // Communications of the ACM, volume 21, number 7, pages 558–565, July 1978 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Flamport%2Fpubs%2Ftime-clocks.pdf.

57. *Défago X., Schiper A., Urbán P.* Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey // ACM Computing Surveys, volume 36, number 4, pages 372–421, December 2004 [Электронный ресурс]. — Режим доступа: https://dspace.jaist.ac.jp/dspace/bitstream/10119/4883/1/defago_et_al.pdf.

58. *Attiya H., Welch J.* Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2nd edition. — John Wiley & Sons, 2004.

59. *Balakrishnan M., Malkhi D., Prabhakaran V., et al.* CORFU: A Shared Log Design for Flash Clusters // 9th USENIX Symposium on Networked Systems Design and

Implementation (NSDI), April 2012 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final30.pdf.

60. *Schneider F. B.* Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial // ACM Computing Surveys, volume 22, number 4, pages 299–319, December 1990 [Электронный ресурс]. — Режим доступа: http://www.cs.cornell.edu/fbs/publications/smsurvey.pdf.
61. *Thomson A., Diamond T., Weng S.-C., et al.* Calvin: Fast Distributed Transactions for Partitioned Database Systems // ACM International Conference on Management of Data (SIGMOD), May 2012 [Электронный ресурс]. — Режим доступа: http://cs.yale.edu/homes/thomson/publications/calvin-sigmod12.pdf.
62. *Balakrishnan M., Malkhi D., Wobber T., et al.* Tango: Distributed Data Structures over a Shared Log // 24th ACM Symposium on Operating Systems Principles (SOSP), November 2013 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/tango-distributed-data-structures-over-a-shared-log/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F199947%2Ftango.pdf.
63. *Renesse van R., Schneider F. B.* Chain Replication for Supporting High Throughput and Availability // 6th USENIX Symposium on Operating System Design and Implementation (OSDI), December 2004 [Электронный ресурс]. — Режим доступа: http://static.usenix.org/legacy/events/osdi04/tech/full_papers/renesse/renesse.pdf.
64. *Lamport L.* How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs // IEEE Transactions on Computers, volume 28, number 9, pages 690–691, September 1979 [Электронный ресурс]. — Режим доступа: http://research-srv.microsoft.com/en-us/um/people/lamport/pubs/multi.pdf.
65. *Söztutar E., Das D., Shanklin C.* Apache HBase High Availability at the Next Level. January 22, 2015 [Электронный ресурс]. — Режим доступа: https://hortonworks.com/blog/apache-hbase-high-availability-next-level/.
66. *Cooper B. F., Ramakrishnan R., Srivastava U., et al.* PNUTS: Yahoo!'s Hosted Data Serving Platform // 34th International Conference on Very Large Data Bases (VLDB), August 2008 [Электронный ресурс]. — Режим доступа: https://people.mpi-sws.org/~druschel/courses/ds/papers/cooper-pnuts.pdf.
67. *Chandra T. D., Toueg S.* Unreliable Failure Detectors for Reliable Distributed Systems // Journal of the ACM, volume 43, number 2, pages 225–267, March 1996 [Электронный ресурс]. — Режим доступа: http://courses.csail.mit.edu/6.852/08/papers/CT96-JACM.pdf.
68. *Fischer M. J., Lynch N., Paterson M. S.* Impossibility of Distributed Consensus with One Faulty Process // Journal of the ACM, volume 32, number 2, pages 374–382, April 1985 [Электронный ресурс]. — Режим доступа: https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf.
69. *Ben-Or M.* Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols // 2nd ACM Symposium on Principles of Distributed Computing (PODC), August 1983.

70. *Gray J. N., Lamport L.* Consensus on Transaction Commit // ACM Transactions on Database Systems (TODS), volume 31, number 1, pages 133–160, March 2006 [Электронный ресурс]. — Режим доступа: http://db.cs.berkeley.edu/cs286/papers/paxoscommit-tods2006.pdf.

71. *Guerraoui R.* Revisiting the Relationship Between Non-Blocking Atomic Commitment and Consensus // 9th International Workshop on Distributed Algorithms (WDAG), September 1995 [Электронный ресурс]. — Режим доступа: https://pdfs.semanticscholar.org/5d06/489503b6f791aa56d2d7942359c2592e44b0.pdf.

72. *Pillai T. S., Chidambaram V., Alagappan R., et al.* All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications // 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), October 2014 [Электронный ресурс]. — Режим доступа: http://research.cs.wisc.edu/wind/Publications/alice-osdi14.pdf.

73. *Gray J.* The Transaction Concept: Virtues and Limitations // 7th International Conference on Very Large Data Bases (VLDB), September 1981 [Электронный ресурс]. — Режим доступа: http://jimgray.azurewebsites.net/papers/thetransactionconcept.pdf.

74. *Garcia-Molina H., Salem K.* Sagas // ACM International Conference on Management of Data (SIGMOD), May 1987 [Электронный ресурс]. — Режим доступа: http://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf.

75. *Mohan C., Lindsay B. G., Obermarck R.* Transaction Management in the R* Distributed Database Management System // ACM Transactions on Database Systems, volume 11, number 4, pages 378–396, December 1986 [Электронный ресурс]. — Режим доступа: https://cs.brown.edu/courses/csci2270/archives/2012/papers/dtxn/p378-mohan.pdf.

76. Distributed Transaction Processing: The XA Specification // X/Open Company Ltd., Technical Standard XO/CAE/91/300, December 1991 [Электронный ресурс]. — Режим доступа: http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf.

77. *Spille M.* XA Exposed, Part II. April 3, 2004 [Электронный ресурс]. — Режим доступа: www.jroller.com/pyrasun/entry/xa_exposed_part_ii_schwartz.

78. *Neto I. S., Reverbel F.* Lessons Learned from Implementing WS-Coordination and WS-AtomicTransaction // 7th IEEE/ACIS International Conference on Computer and Information Science (ICIS), May 2008 [Электронный ресурс]. — Режим доступа: https://www.ime.usp.br/~reverbel/papers/icis2008.pdf.

79. *Johnson J. E., Langworthy D. E., Lamport L., Vogt F. H.* Formal Specification of a Web Services Protocol // 1st International Workshop on Web Services and Formal Methods (WS-FM), February 2004 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/formal-specification-of-a-web-services-protocol/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Flamport%2Fpubs%2Fwsfm-web.pdf.

80. *Skeen D.* Nonblocking Commit Protocols // at ACM International Conference on Management of Data (SIGMOD), April 1981 [Электронный ресурс]. — Режим доступа: http://www.cs.utexas.edu/~lorenzo/corsi/cs380d/papers/Ske81.pdf.

81. *Hohpe G.* Your Coffee Shop Doesn't Use Two-Phase Commit // IEEE Software, volume 22, number 2, pages 64–66, March 2005 [Электронный ресурс]. — Режим доступа: https://www.martinfowler.com/ieeeSoftware/coffeeShop.pdf.
82. *Helland P.* Life Beyond Distributed Transactions: An Apostate's Opinion // at 3rd Biennial Conference on Innovative Data Systems Research (CIDR), January 2007 [Электронный ресурс]. — Режим доступа: http://www-db.cs.wisc.edu/cidr/cidr2007/papers/cidr07p15.pdf.
83. *Oliver J.* My Beef with MSDTC and Two-Phase Commits. April 4, 2011 [Электронный ресурс]. — Режим доступа: http://blog.jonathanoliver.com/my-beef-with-msdtc-and-two-phase-commits/.
84. *Eini O. (Ahende Rahien)* The Fallacy of Distributed Transactions. July 17, 2014 [Электронный ресурс]. — Режим доступа: https://ayende.com/blog/167362/the-fallacy-of-distributed-transactions.
85. *Vasters C.* Transactions in Windows Azure (with Service Bus) — An Email Discussion. July 30, 2012 [Электронный ресурс]. — Режим доступа: https://blogs.msdn.microsoft.com/clemensv/2012/07/30/transactions-in-windows-azure-with-service-bus-an-email-discussion/.
86. Understanding Transactionality in Azure // NServiceBus Documentation, Particular Software, 2015 [Электронный ресурс]. — Режим доступа: https://docs.particular.net/nservicebus/azure/understanding-transactionality-in-azure.
87. *Wigginton R., Lowe R., Albe M., Ipar F.* Distributed Transactions in MySQL // MySQL Conference and Expo, April 2013 [Электронный ресурс]. — Режим доступа: https://www.percona.com/live/mysql-conference-2013/sites/default/files/slides/XA_final.pdf.
88. *Spille M.* XA Exposed, Part I. April 3, 2004 [Электронный ресурс]. — Режим доступа: http://www.jroller.com/pyrasun/entry/xa_exposed.
89. *Dhariwal A.* Orphaned MSDTC Transactions (-2 spids). December 12, 2008 [Электронный ресурс]. — Режим доступа: http://www.eraofdata.com/sql-server/troubleshooting-sql-server/orphaned-msdtc-transactions-2-spids/.
90. *Randal P.* Real World Story of DBCC PAGE Saving the Day. June 19, 2013 [Электронный ресурс]. — Режим доступа: https://www.sqlskills.com/blogs/paul/real-world-story-of-dbcc-page-saving-the-day/.
91. in-doubt xact resolution Server Configuration Option // SQL Server 2016 documentation, Microsoft, Inc., 2016 [Электронный ресурс]. — Режим доступа: https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/in-doubt-xact-resolution-server-configuration-option.
92. *Dwork C., Lynch N., Stockmeyer L.* Consensus in the Presence of Partial Synchrony // Journal of the ACM, volume 35, number 2, pages 288–323, April 1988 [Электронный ресурс]. — Режим доступа: https://www.net.t-labs.tu-berlin.de/~petr/ADC-07/papers/DLS88.pdf.
93. *Castro M., Liskov B. H.* Practical Byzantine Fault Tolerance and Proactive Recovery // ACM Transactions on Computer Systems, volume 20, number 4, pages 396–461,

November 2002 [Электронный ресурс]. — Режим доступа: http://zoo.cs.yale.edu/classes/cs426/2012/bib/castro02practical.pdf.

94. *Oki B. M., Liskov B. H.* Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems // 7th ACM Symposium on Principles of Distributed Computing (PODC), August 1988 [Электронный ресурс]. — Режим доступа: http://www.cs.princeton.edu/courses/archive/fall11/cos518/papers/viewstamped.pdf.
95. *Liskov B. H., Cowling J.* Viewstamped Replication Revisited // Massachusetts Institute of Technology, Tech Report MIT-CSAIL-TR-2012-021, July 2012 [Электронный ресурс]. — Режим доступа: http://pmg.csail.mit.edu/papers/vr-revisited.pdf.
96. *Lamport L.* The Part-Time Parliament // ACM Transactions on Computer Systems, volume 16, number 2, pages 133–169, May 1998 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/part-time-parliament/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Flamport%2Fpubs%2Flamport-paxos.pdf.
97. *Lamport L.* Paxos Made Simple // ACM SIGACT News, volume 32, number 4, pages 51–58, December 2001 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Flamport%2Fpubs%2Fpaxos-simple.pdf.
98. *Chandra T. D., Griesemer R., Redstone J.* Paxos Made Live — An Engineering Perspective // 26th ACM Symposium on Principles of Distributed Computing (PODC), June 2007 [Электронный ресурс]. — Режим доступа: http://www.read.seas.harvard.edu/~kohler/class/08w-dsi/chandra07paxos.pdf.
99. *Renesse van R.* Paxos Made Moderately Complex. March 2011 [Электронный ресурс]. — Режим доступа: http://www.cs.cornell.edu/home/rvr/Paxos/paxos.pdf.
100. *Ongaro D.* Consensus: Bridging Theory and Practice // PhD Thesis, Stanford University, August 2014 [Электронный ресурс]. — Режим доступа: https://github.com/ongardie/dissertation.
101. *Howard H., Schwarzkopf M., Madhavapeddy A., Crowcroft J.* Raft Refloated: Do We Have Consensus? // ACM SIGOPS Operating Systems Review, volume 49, number 1, pages 12–21, January 2015 [Электронный ресурс]. — Режим доступа: http://www.cl.cam.ac.uk/~ms705/pub/papers/2015-osr-raft.pdf.
102. *Medeiros A.* ZooKeeper's Atomic Broadcast Protocol: Theory and Practice // Aalto University School of Science, March 20, 2012 [Электронный ресурс]. — Режим доступа: http://www.tcs.hut.fi/Studies/T-79.5001/reports/2012-deSouzaMedeiros.pdf.
103. *Renesse van R., Schiper N., Schneider F. B.* Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab // IEEE Transactions on Dependable and Secure Computing, volume 12, number 4, pages 472–484, September 2014 [Электронный ресурс]. — Режим доступа: http://arxiv.org/abs/1309.5671.
104. *Portnoy W.* Lessons Learned from Implementing Paxos // blog.willportnoy.com, June 14, 2012 [Электронный ресурс]. — Режим доступа: http://blog.willportnoy.com/2012/06/lessons-learned-from-paxos.html.

105. *Howard H., Malkhi D., Spiegelman A.* Flexible Paxos: Quorum Intersection Revisited. August 24, 2016 [Электронный ресурс]. — Режим доступа: https://arxiv.org/abs/1608.06696.

106. *Howard H., Crowcroft J.* Coracle: Evaluating Consensus at the Internet Edge // Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), August 2015 [Электронный ресурс]. — Режим доступа: http://www.sigcomm.org/sites/default/files/ccr/papers/2015/August/2829988-2790010.pdf.

107. *Kingsbury K.* Call Me Maybe: Elasticsearch 1.5.0. April 27, 2015 [Электронный ресурс]. — Режим доступа: https://aphyr.com/posts/323-call-me-maybe-elasticsearch-1-5-0.

108. *Kelly I.* BookKeeper Tutorial. October 2014 [Электронный ресурс]. — Режим доступа: https://github.com/ivankelly/bookkeeper-tutorial.

109. *Fournier C.* Consensus Systems for the Skeptical Architect // Craft Conference, Budapest, Hungary, April 2015 [Электронный ресурс]. — Режим доступа: http://www.ustream.tv/recorded/61483409.

110. *Birman K. P.* A History of the Virtual Synchrony Replication Model // Replication: Theory and Practice, Springer LNCS volume 5959, chapter 6, pages 91–120, 2010 [Электронный ресурс]. — Режим доступа: https://www.truststc.org/pubs/713.html.

Часть III
Производные данные

В частях I и II этой книги мы собрали воедино все основные понятия, касающиеся распределенных баз данных, начиная от размещения данных на диске и заканчивая пределами распределенной согласованности при наличии сбоев. Однако все это время мы исходили из предположения, что в приложении имеется только одна БД.

На практике информационные системы обычно бывают гораздо сложнее. В большом приложении часто приходится получать и обрабатывать данные самыми разными способами, и ни одна база не способна удовлетворить всем этим требованиям. Поэтому в приложениях обычно используется сочетание нескольких информационных хранилищ, индексов, кэшей, аналитических систем и т. п. и реализованы механизмы перемещения данных из одного хранилища в другое.

В заключительной части этой книги мы рассмотрим темы, касающиеся интеграции нескольких разных информационных систем — они могут иметь разные информационные модели и быть оптимизированы для разных шаблонов доступа — в одну согласованную архитектуру приложения. Поставщики часто игнорируют этот аспект системного проектирования и утверждают, что их продукт способен удовлетворить все ваши потребности. На самом деле интеграция разнотипных систем является одной из самых важных задач, которую необходимо решить при построении нетривиального приложения.

III.1. Системы записи и производные данные

На высоком уровне системы хранения и обработки информации могут быть сгруппированы в две обширные категории.

- *Системы записи*. Известны также как *источник правды*. Система записи содержит надежную версию данных. Когда появляются новые данные, например, в результате пользовательского ввода, они сначала записываются сюда. Каждый факт представлен ровно один раз (представление обычно *нормализуется*). В случае каких-либо расхождений с другой системой правильным (по определению) считается значение в системе записи.
- *Производные информационные системы*. Данные в такой системе являются результатом получения неких уже существующих данных из другой системы, которые каким-то образом были преобразованы или обработаны. В случае потери производные данные можно воссоздать из исходного источника. Классическим примером такой системы является кэш: данные можно получить из кэша при их наличии там. Но если он не содержит нужной информации, то можно обратиться за ней в исходную базу данных. К этой категории также относятся денормализованные значения, индексы и материализованные представления. В системах выдачи рекомендаций прогнозируемые сводные данные часто генерируются на основе журналов использования.

С технической точки зрения производные данные являются *избыточными* в том смысле, что дублируют существующую информацию. Однако часто бывает важно обеспечить хорошую скорость выполнения запросов чтения. Они обычно *денормализованы*. Можно создать несколько наборов данных на основе одного источника, и это позволит просматривать данные с разных точек зрения.

Не все системы делают четкое различие между системами записи и производными данными по своей архитектуре. Но это очень полезное различие, поскольку делает более понятным прохождение данных через систему: в нем четко указывается, какие входы и выходы имеет каждая часть системы и как они зависят друг от друга.

Большинство баз данных, систем хранения информации и языков запросов не являются по своей сути системами записи или производными системами. База — это всего лишь инструмент: как вы ее используете, зависит только от вас. Различие между системой записи и производной информационной системой зависит не от инструмента, а от того, как вы применяете его в своем приложении.

Понимание того, откуда получены те или иные данные, позволит вам построить ясную системную архитектуру — в противном случае она может быть весьма запутанной. Мы постоянно будем возвращаться к этой теме в текущей части книги.

III.2. Обзор глав

Для начала в главе 10 мы рассмотрим пакетно-ориентированные системы обработки данных, такие как MapReduce. Мы увидим, что они предоставляют хорошие инструменты и являются основой для построения больших информационных систем. В главе 11 применим эти идеи к информационным потокам, что позволит уменьшить задержки при обработке данных. В главе 12, завершающей эту книгу, мы рассмотрим идеи того, как можно использовать изученные инструменты для создания надежных, масштабируемых и удобных в сопровождении приложений.


МОРЕ ПРОИЗВОДНЫХ ДАННЫХ
Mahout
ДЕЛЬТА МАШИННОГО ОБУЧЕНИЯ
К потоковой обработке (глава 11)
К царству аналитики (глава 3) и интеграции данных (глава 12)
Terrapin
Механизмы рекомендаций
Voldemort
Flink
Spark
СИСТЕМЫ, УПРАВЛЯЕМЫЕ ПОТОКАМИ ДАННЫХ
РЕКА ПАКЕТНЫХ ВЫХОДНЫХ ДАННЫХ
ОБРАБОТКА
Hive
Tez
GraphX
ПОИСКОВЫЕ ИНДЕКСЫ
Impala
БАЗЫ ДАННЫХ С МАССОВО-ПАРАЛЛЕЛЬНОЙ АРХИТЕКТУРОЙ
РАСПРЕДЕЛЕННЫЕ ФАЙЛОВЫЕ СИСТЕМЫ
ИТЕРАТИВНАЯ
Giraph
Solr
Потоки операций MapReduce
Гавань ETL
ОДНОУЗЛОВАЯ ОБРАБОТКА
GraphChi
Unix-утилиты

10 Пакетная обработка

> Хорошая система не может зависеть от отдельного человека. После того как первоначальная разработка завершена и обеспечена достаточная надежность, начинается настоящее тестирование — приходят люди с разными точками зрения и начинают ставить на ней свои эксперименты.
>
> *Дональд Кнут*

В двух предыдущих частях этой книги мы рассмотрели множество различных *запросов* с их *откликами* и *результатами*. Такой стиль обработки данных принят во многих современных информационных системах: вы запрашиваете некие сведения или посылаете инструкцию в расчете на то, что через некоторое время система даст ответ. Так работают базы данных, кэши, поисковые индексы, веб-серверы и многие другие системы.

В *онлайновых* системах, где браузер запрашивает страницу или сервис обращается к удаленному API, обычно предполагается, что запрос исходит от пользователя-человека, и этот пользователь ждет ответа. Он не захочет ждать слишком долго, и, как следствие, в таких системах большое внимание уделяется *времени отклика* (см. подраздел «Описание производительности» раздела 1.3).

Благодаря Интернету с его растущим количеством API на базе HTTP/REST взаимодействие в стиле «запрос-отклик» стало таким распространенным, что его уже воспринимают как нечто само собой разумеющееся. Но следует помнить: это

не единственный способ построения систем. Есть и другие варианты, каждый со своими достоинствами. Выделяют три типа систем.

- *Сервисы (онлайновые системы)*. Сервис ожидает запросов и инструкций, поступающих от клиента. После того как запрос получен, сервис пытается обработать его максимально быстро и возвращает отклик. Последний обычно является первичной мерой производительности сервиса. Кроме того, очень важна доступность (если клиент не может получить доступ к сервису, то пользователь, скорее всего, получит сообщение об ошибке).
- *Системы пакетной обработки (автономные системы)*. Такая система принимает большое количество данных, запускает *задачу* для их обработки и выдает некие данные на выход. Подобные задачи обычно выполняются довольно долго (от нескольких минут до нескольких дней), так что пользователь, как правило, не ждет их окончания. Зато пакетные задачи часто запускаются по расписанию через определенные промежутки времени (например, раз в день). Главной мерой производительности пакетной задачи обычно является *пропускная способность* (время, необходимое для обработки входного набора данных установленного размера). В этой главе мы рассмотрим именно системы пакетной обработки.
- *Системы поточной обработки (системы почти реального времени)*. Поточная обработка представляет собой нечто среднее между онлайновой и автономной (пакетной) обработкой данных (именно поэтому ее иногда называют обработкой *почти реального времени* или *первоочередной* обработкой). Как и при пакетной обработке, эта система принимает данные на входе и генерирует выходные данные (вместо того чтобы отвечать на запросы). Однако в случае появления событий поточные задачи реагируют на них, в то время как пакетные работают только с фиксированным набором входных данных. Благодаря этому отличию время ожидания у поточных систем меньше, чем у аналогичных пакетных. О том, как построить систему поточной обработки на основании пакетной, написано в главе 11.

Как вы увидите далее в этой главе, пакетная обработка является важной составляющей при построении надежных, масштабируемых и удобных в сопровождении приложений. Например, опубликованный в 2001 году алгоритм пакетной обработки MapReduce [1] был (возможно, преждевременно) назван «алгоритмом, который обеспечивает масштабируемость Google» [2]. Впоследствии он был внедрен во многие информационные системы с открытым исходным кодом, в том числе Hadoop, CouchDB и MongoDB.

По сравнению с системами параллельной обработки, которые создавались для складов данных много лет [3, 4], MapReduce — довольно низкоуровневая модель программирования. Однако она является большим шагом вперед в смысле масштабирования обработки, достигаемого на обычном аппаратном обеспечении. И хотя важность системы MapReduce сейчас идет на убыль [5], ей все еще стоит уделить

внимание, поскольку она дает ясное представление о том, почему и как следует использовать пакетную обработку.

В сущности, пакетная обработка — очень старая форма вычислений. Задолго до того, как были изобретены программируемые цифровые компьютеры, полуавтоматический вариант пакетной обработки применялся в машинах, принимавших перфокарты, таких как Холеритовы вычислительные машины, использовавшиеся при переписи населения США в 1890 году [6] для вычисления сводной статистики по большим объемам входных данных. MapReduce необыкновенно похожа на электромеханические перфокартные машины IBM, которые широко применялись для обработки бизнес-данных в 1940-х и 1950-х годах [7]. История часто повторяется.

В этой главе мы рассмотрим MapReduce и еще несколько алгоритмов и систем пакетной обработки, а также изучим, как они используются в современных информационных системах. Но для начала обсудим обработку данных стандартными средствами Unix. Даже если вы с ними уже знакомы, напоминание о философии Unix того стоит: идеи и уроки Unix легко переносятся на большие, гетерогенные распределенные информационные системы.

10.1. Пакетная обработка средствами Unix

Начнем с простого примера. Предположим, у нас есть веб-сервер, который добавляет строку в файл журнала каждый раз, когда выполняет запрос. Например, при использовании стандартного формата журнала доступа nginx строка журнала может выглядеть так:

```
216.58.210.78 - - [27/Feb/2015:17:55:11 +0000] "GET /css/typography.css HTTP/1.1"
200 3377 "http://martin.kleppmann.com/" "Mozilla/5.0 (Macintosh; Intel Mac OS X
10_9_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2214.115 Safari/537.36"
```

(На самом деле это одна строка, разбитая на несколько для удобства чтения.) Здесь много информации. Чтобы ее интерпретировать, нужно посмотреть на формат журнала, который выглядит следующим образом:

```
$remote_addr - $remote_user [$time_local] "$request"
$status $body_bytes_sent "$http_referer" "$http_user_agent"
```

Итак, из этой одной строки журнала следует, что 27 февраля 2015 года в 17:55:11 по UTC сервер получил запрос на файл `/css/typography.css` с клиентского IP-адреса 216.58.210.78. Пользователь не был аутентифицирован, поэтому переменной `$remote_user` было присвоено значение «дефис» (`-`). Статус ответа был 200 (то есть запрос был успешным), а ответ занял 3377 байт. Клиент использовал браузер Chrome 40, и тот загрузил данный файл, поскольку он был указан на странице с URL http://martin.kleppmann.com/.

Простой анализ журнала

Существует ряд программных средств, способных строить на основе таких файлов журналов красивые отчеты о трафике вашего сайта. Но ради практики мы построим подобный отчет, используя только базовые инструменты Unix. Например, предположим, что мы хотим выбрать пять самых популярных страниц сайта. В командной среде Unix это можно сделать так:[1]

```
cat /var/log/nginx/access.log |  (1)
  awk '{print $7}' |  (2)
  sort             |  (3)
  uniq -c          |  (4)
  sort -r -n       |  (5)
  head -n 5           (6)
```

(1) Прочитать файл журнала.

(2) Разделить каждую строку на поля с помощью пробелов и вывести только седьмое поле каждой строки — именно там находятся нужные нам URL, в данном случае это `/css/typography.css`.

(3) Упорядочить список запрошенных URL по алфавиту. Если какой-либо адрес был запрошен *n* раз, то после сортировки файла строки с этим URL будут идти подряд.

(4) С помощью команды `uniq` проверить, являются ли две соседние строки одинаковыми, и отфильтровать повторяющиеся строки на входе. Опция `-c` позволяет вывести счетчик, в котором для каждого URL указано, сколько раз этот адрес появился на входе.

(5) Вторая команда `sort` сортирует по числу (`-n`) в начале каждой строки, то есть по количеству запросов этого URL. Результаты представляются в порядке убывания (`-r`), то есть первым идет адрес с наибольшим количеством запросов.

(6) Оставить только первые пять строк (`-n 5`) входных данных, отбросив остальное.

Результат этой серии команд выглядит примерно так:

```
4189 /favicon.ico
3631 /2013/05/24/improving-security-of-ssh-private-keys.html
2124 /2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html
1369 /
 915 /css/typography.css
```

[1] Некоторым нравится указывать на то, что cat здесь не требуется, так как входной файл можно указать сразу в виде аргумента awk. Однако так линейный конвейер более заметен.

Тем, кто не очень хорошо знаком с инструментами Unix, представленная выше командная строка, вероятно, покажется несколько запутанной, однако ее возможности невероятно велики. Она способна обработать несколько гигабайтов журнальных записей за считаные секунды, и ее можно легко настроить на различные виды анализа в соответствии с вашими потребностями. Например, для удаления из отчета файлов CSS аргумент команды `awk` нужно заменить на `'$7 !~ /\.css$/ {print $7}'`, а чтобы вместо самых посещаемых страниц получить самые популярные клиентские IP-адреса, — на `'{print $1}'` и т. д.

В этой книге не хватит места для подробного изучения инструментов Unix, но они того стоят. Неожиданно много видов анализа данных можно провести всего за несколько минут с помощью команд `awk`, `sed`, `grep`, `sort`, `uniq` и `xargs`, и они работают на удивление быстро [8].

Что писать: цепь команд или программу?

Вместо цепи команд Unix можно написать простую программу, выполняющую те же действия. Например, на Ruby она может выглядеть так:

```
counts = Hash.new(0) ❶

File.open('/var/log/nginx/access.log') do |file|
  file.each do |line|
    url = line.split[6] ❷
    counts[url] += 1 ❸
  end
end

top5 = counts.map{|url, count| [count, url] }.sort.reverse[0...5] ❹
top5.each{|count, url| puts "#{count} #{url}" } ❺
```

❶ `counts` — хеш-таблица, в которой хранится счетчик количества появлений каждого URL. По умолчанию этот счетчик равен нулю.

❷ Из каждой строки журнала извлекаем седьмое поле, разделенное пробелами, — это URL. (Номер элемента массива равен 6, потому что в Ruby массивы индексируются с нуля.)

❸ Увеличиваем счетчик для данного адреса в текущей строке журнала.

❹ Упорядочиваем содержимое хеш-таблицы по убыванию значения счетчика и берем первые пять записей.

❺ Выводим эти верхние пять записей.

Данная программа не столь лаконична, как конвейер Unix, но довольно легко читается. Какой из двух вариантов вы предпочтете — отчасти дело вкуса. Однако, кроме внешних синтаксических различий, существует весомая разница в способе выполнения, и она станет очевидной, если выполнить анализ большого файла.

Сортировка или агрегация в памяти?

Сценарий Ruby сохраняет URL в памяти в виде хеш-таблицы, где каждому адресу соответствует количество обращений к нему. В конвейере Unix такой хеш-таблицы нет, вместо нее используется сортировка списка URL, в котором они просто повторяются столько раз, сколько встречаются.

Какой вариант лучше? Это зависит от количества разных URL. Для большинства сайтов малого и среднего размера, вероятно, можно сохранить все уникальные адреса и их счетчики примерно в 1 Гбайт памяти. В данном примере *рабочий набор* задачи (объем памяти для произвольного доступа) зависит только от количества уникальных URL: даже если некоему адресу в журнале соответствует миллион записей, то в хеш-таблице это все равно одна запись: URL и размер счетчика. При достаточно малом рабочем наборе хранимая в памяти хеш-таблица хорошо работает даже на ноутбуке.

Однако если рабочий набор задачи превышает объем доступной памяти, то сортировка позволяет эффективно использовать диски. Здесь работает тот же принцип, который мы обсуждали в подразделе «SS-таблицы и LSM-деревья» раздела 3.1: можно отсортировать фрагменты данных в памяти и записать на диск в виде сегментных файлов, а затем объединить несколько отсортированных сегментов в более крупный отсортированный файл. В этом алгоритме есть последовательные шаблоны доступа, хорошо работающие на дисках. (Помните оптимизацию для последовательного ввода-вывода, которая была повторяющейся темой в главе 3? Этот шаблон появляется и здесь.)

Утилита `sort` в GNU Coreutils (Linux) автоматически обрабатывает массивы большого размера, записывая их на диск, и распараллеливает сортировку по нескольким ядрам процессора [9]. Таким образом, рассмотренная нами простая цепь команд Unix легко масштабируется для больших наборов данных, не перегружая память. Очевидно, в этом случае узким местом будет скорость чтения исходного файла с диска.

Философия Unix

То, что мы в предыдущем примере смогли легко проанализировать файл журнала, используя цепь команд, не случайно: в действительности это одна из ключевых идей, заложенных в основу Unix, и она до сих пор на удивление актуальна. Присмотримся к Unix внимательнее, чтобы позаимствовать еще некоторые идеи [10].

Дуг Макилрой (Doug McIlroy), изобретатель цепочек Unix, первоначально в 1964 году описал их так [11]: «У нас должны быть способы подключения программ по принципу садового шланга, когда в другом сегменте возникает необходимость обработать данные другим способом. Это также способ ввода-вывода». Аналогия

с шлангом сохранилась, и идея объединения программ в конвейер стала частью того, что теперь известно как *философия Unix*: набор правил программирования, который стал популярным среди разработчиков и пользователей Unix. Философия была описана в 1978 году [12, 13].

- Пусть каждая программа делает что-то одно, и делает это хорошо. Для решения новой задачи напишите новую программу, а не усложняйте старую, добавляя новые «функции».
- Пусть выходные данные каждой программы могут служить входными данными для другой, пока неизвестной программы. Не засоряйте вывод посторонней информацией. Избегайте жестких табличных или двоичных форматов ввода. Не настаивайте на интерактивном вводе.
- Проектируйте и разрабатывайте программное обеспечение, даже операционные системы, так, чтобы протестировать их как можно раньше, в идеале — в течение нескольких недель. Без колебаний отказывайтесь от грубо написанных частей, переделывайте их.
- Чтобы упростить программирование, используйте инструменты вместо неквалифицированной помощи, даже если для их создания приходится отвлечься от основной задачи, а впоследствии — отказаться от некоторых из них.

Этот подход — автоматизация, быстрое прототипирование, инкрементная итерация, поощрение экспериментов и разбивка крупных проектов на управляемые блоки — очень близок к современным принципам Agile и DevOps. Удивительно мало изменений за 40 лет.

Инструмент `sort` — отличный пример программы, хорошо выполняющей одну задачу. Возможно, в ней сортировка реализована лучше, чем в большинстве стандартных библиотек для языков программирования (которые не помещаются на диске и не используют многопоточность даже когда это оправданно). Однако сам по себе `sort` не очень полезен. Он становится высокоэффективным только в сочетании с другими инструментами Unix, такими как `uniq`.

Оболочка Unix, подобная `bash`, позволяет легко *объединять* маленькие программы в удивительно эффективные средства обработки данных. Многие из этих программ написаны разными группами людей, но их можно объединить различными способами. Как в Unix реализована такая возможность?

Единый интерфейс

Чтобы выходные данные одной программы служили входными для другой, эти программы должны использовать один и тот же формат данных, другими словами — иметь совместимый интерфейс. Для подачи выходных данных *любой* программы на вход *любой* программы *все* программы должны применять один и тот же интерфейс ввода/вывода.

В Unix этот интерфейс является файлом (точнее, файловым дескриптором). Файл — это просто упорядоченная последовательность байтов. Благодаря такому простому интерфейсу могут быть представлены самые разные вещи: настоящий файл в файловой системе, канал связи с другим процессом (сокет Unix, `stdin`, `stdout`), драйвер устройства (такой как `/dev/audio` или `/dev/lp0`), сокет, представляющий TCP-соединение, и т. п. Сейчас мы считаем это само собой разумеющимся, но на самом деле удивительно, какие разные вещи могут иметь единый интерфейс, благодаря которому их легко соединять[1].

По соглашению многие (но не все) Unix-программы рассматривают эту последовательность байтов как текст в формате ASCII. Мы применили указанный факт в примере с анализом журнала: `awk`, `sort`, `uniq` и `head` предполагают, что входной файл — список записей, разделенных символом `\n` (новая строка, ASCII `0x0A`). Данный символ выбран произвольно; возможно, разделитель записей ASCII `0x1E` был бы лучшим вариантом, поскольку предназначен именно для этой цели [14], но в любом случае тот факт, что все эти программы стандартизованы и рассчитаны на использование одного и того же разделителя записей, позволяет им взаимодействовать.

Синтаксический анализ каждой записи (строки ввода) не так однозначен. Инструменты Unix обычно разбивают строку на поля, разделенные символами пробела или табуляции. Кроме того, иногда используется формат CSV (разделение запятыми), разделение вертикальными линиями и другие кодировки. Даже у такого простого инструмента, как `xargs`, имеется полдюжины параметров командной строки, определяющих, как следует читать его входные данные.

Единый интерфейс в виде ASCII-текста, как правило, работает, но это не всегда выглядит красиво: в примере анализа журнала для извлечения URL мы использовали формат `{print $7}`, что не очень хорошо читается. В идеале это должно было бы выглядеть примерно так: `{print $ request_url}`. Позже мы еще вернемся к данной идее.

---

[1] Другой пример единого интерфейса — URL и HTTP, основа Интернета. URL идентифицирует конкретный объект (ресурс) на сайте; на любой адрес можно ссылаться с любого сайта. Таким образом, с помощью браузера пользователь легко переходит по ссылкам между сайтами, хотя серверы принадлежат самым разным организациям. Сейчас этот принцип представляется очевидным, но в свое время он стал ключевым озарением, позволившим сделать Интернет тем, чем он является сегодня. Предшествовавшие ему системы были не столь однородными: например, электронные доски объявлений (bulletin board systems, BBS) имели собственные номера телефона и формат скорости передачи. Ссылка с одной BBS на другую представлялась в виде номера телефона и настроек модема. Пользователь должен был повесить трубку, набрать номер другой BBS, а затем вручную найти нужную ему информацию. Подключить напрямую какой-либо фрагмент контента одной BBS к другой было невозможно.

Пускай он и неидеален, но даже спустя десятилетия унифицированный интерфейс Unix все еще вызывает восхищение. Немногие компоненты программного обеспечения взаимодействуют и сочетаются друг с другом настолько хорошо, как инструменты Unix: например, невозможно так же легко проанализировать содержимое учетной записи электронной почты и историю покупок в Интернете, передать результат в электронную таблицу, опубликовать его в социальной сети или в «Википедии». Сегодня программы, способные работать вместе так же слаженно, как инструменты Unix, — скорее исключение, чем норма.

Даже БД с *одной и той же моделью данных* часто не позволяют легко извлечь данные из одной базы и передать в другую. Отсутствие интеграции приводит к раздробленности и хаотичности данных.

Отделение логики от передачи данных

Еще одной характерной особенностью инструментов Unix является использование стандартных потоков ввода (`stdin`) и вывода (`stdout`). Если запустить программу без входных параметров, то `stdin` поступает с клавиатуры, а `stdout` выводится на экран. Однако также можно вводить данные из файла и/или перенаправлять вывод в файл. Конвейер Unix позволяет присоединить `stdout` одного процесса к `stdin` другого (с небольшим буфером в памяти, не записывая весь промежуточный поток данных на диск).

При необходимости программа может читать и записывать файлы напрямую, но принципы Unix работают лучше, если программа не беспокоится о конкретных путях файлов, просто применяя `stdin` и `stdout`. В этом случае пользователь оболочки способен подключать вход и выход любым способом; программа «не знает», не заботится о том, откуда поступают входные данные и куда направляются выходные. (Это можно назвать формой *слабого связывания*, *поздней привязки* [15] или *инверсией управления* [16].) Отделение ввода/вывода от логики программы позволяет легко объединять небольшие инструменты в более крупные системы.

Вы даже можете писать собственные программы и объединять их со стандартными инструментами операционной системы. Чтобы программу можно было использовать в конвейерах обработки данных, она лишь должна читать входные данные из `stdin` и записывать вывод в `stdout`. Так, для анализа журнала можно написать инструмент, преобразующий строки из пользовательского файла в более разумные идентификаторы браузера либо же заменяющий IP-адреса на коды стран, и просто подключить его к конвейеру. Программе `sort` безразлично, откуда поступают данные: от другой части операционной системы или от написанной вами программы.

Однако с помощью `stdin` и `stdout` можно делать далеко не все. Например, написать программу, которая нуждается в нескольких потоках ввода и вывода, возможно, но

сложно. Нельзя передать вывод программы сетевому соединению [17, 18][1]. Если программа напрямую открывает файлы для чтения и записи, или запускает другую программу в качестве подпроцесса, или открывает сетевое соединение, то этот поток ввода-вывода подключается самой программой. Его тоже можно настроить (например, с помощью параметров командной строки), но гибкость подключения потоков ввода и вывода в оболочке снижается.

Прозрачность и экспериментирование

Отчасти причиной эффективности инструментов Unix является то, что с ними легко контролировать процесс.

- Входные файлы Unix-команд обычно не меняются. Таким образом, можно запускать команды многократно, с различными параметрами командной строки, не повреждая входные файлы.
- В любой момент можно прервать выполнение конвейера, вывести результаты в `less` и проверить, имеют ли они ожидаемую форму. Эта возможность проверки очень удобна при отладке.
- Можно сохранить выходные данные одного этапа конвейера в файл и затем использовать его как входной для второго этапа. Такая комбинация позволит перезапустить впоследствии второй этап без повторного запуска всего конвейера.

Таким образом, хотя инструменты Unix довольно примитивны и просты по сравнению с оптимизаторами запросов в реляционных базах данных, тем не менее они по-прежнему удивительно полезны, особенно для экспериментов.

Однако самым большим ограничением для инструментов Unix является то, что они запускаются только на одной машине, — и тут на помощь приходят такие инструменты, как Hadoop.

10.2. MapReduce и распределенные файловые системы

Модель программирования MapReduce подобна инструментам Unix, но распространяется на много компьютеров — потенциально на тысячи. Как и в случае с Unix, это довольно примитивный, грубый, но удивительно эффективный инструмент. Задача MapReduce подобна процессу Unix: принимает один или несколько потоков ввода и производит один или несколько потоков вывода.

---

[1] Если только не задействовать специальный инструмент, такой как netcat или curl. Unix начал с попытки представить все в виде файлов, но API сокетов BSD отошел от этого соглашения [17]. Исследовательские операционные системы Plan 9 и Inferno более последовательны в применении файлов: в них TCP-соединение представлено в виде файла /net/tcp [18].

Как и большинство инструментов Unix, задача MapReduce обычно не изменяет входные данные и не имеет побочных эффектов — только генерирует выходные. Последние записываются один раз, последовательно (уже записанная часть файла при этом не изменяется).

В то время как инструменты Unix используют `stdin` и `stdout` в качестве потоков ввода и вывода, задачи MapReduce читают и записывают файлы в распределенной файловой системе. В реализации Hadoop MapReduce эта файловая система называется HDFS (Hadoop Distributed File System, распределенная файловая система Hadoop), реализация файловой системы Google — Google File System, GFS, с открытым исходным кодом [19].

Кроме HDFS, есть и другие распределенные файловые системы, такие как GlusterFS и Quantcast File System (QFS) [20]. Сервисы хранения объектов, подобные Amazon S3, Azure Blob Storage и OpenStack Swift [21], во многом схожи[1]. В этой главе мы будем использовать для примера главным образом HDFS, но рассматриваемые принципы применимы к любой распределенной файловой системе.

В отличие от общего диска в архитектурах *Network Attached Storage* (NAS) и *Storage Area Network* (SAN), в основе HDFS лежит принцип «*ничего общего*» (см. введение в часть II). Хранилище с общим диском реализуется в виде централизованного устройства хранения данных, для чего часто требуется специальное оборудование и сетевая инфраструктура, такая как Fibre Channel. Принцип «ничего общего», напротив, не требует специального оборудования — только компьютеров, объединенных в обычную сеть центра обработки и хранения данных.

HDFS состоит из процесса демона, запускаемого на каждом компьютере. Он предоставляет сетевой сервис, позволяющий другим узлам сети получить доступ к файлам, хранящимся на этом компьютере (при условии, что на каждой машине общего назначения в центре обработки и хранения данных есть закрепленные за ним диски). Центральный сервер, называемый *NameNode*, отслеживает, на каком компьютере хранятся те или иные блоки файлов. Таким образом, концептуально HDFS создает одну большую файловую систему, использующую дисковое пространство всех компьютеров, на которых установлен демон.

Чтобы не зависеть от сбоев компьютера и диска, файловые блоки дублируются на нескольких машинах. Репликация может означать просто несколько копий одних и тех же данных на нескольких машинах, как в главе 5, или же схему *удаляющего*

---

[1] Одно из отличий заключается в том, что при использовании HDFS вычислительные задачи могут размещаться на том компьютере, где хранится копия определенного файла, тогда как в хранилищах объектов память и вычисления обычно разделены. Если пропускная способность сети сравнительно невелика, то чтение с локального диска предпочтительнее по производительности. Однако обратите внимание: в случае удаляющего кодирования преимущество локальности теряется, поскольку для восстановления исходного файла необходимо объединить данные с нескольких машин [20].

кодирования, такую как коды Рида — Соломона, позволяющие восстанавливать потерянные данные с меньшими затратами на хранение, чем полная репликация [20, 22]. Эти методы подобны RAID, что обеспечивает избыточность на нескольких дисках, подключенных к одному компьютеру. Разница заключается вот в чем: в распределенной файловой системе доступ к файлам и их репликация выполняются через обычную сеть центра обработки и хранения данных, без применения специального оборудования.

HDFS хорошо масштабируется: на момент написания этой книги самые крупные системы на базе этой файловой системы насчитывали десятки тысяч машин общей емкостью хранения данных в сотни петабайт [23]. Такой масштаб стал возможным благодаря тому, что стоимость хранения данных и доступа к HDFS с помощью обычного аппаратного обеспечения и ПО с открытым исходным кодом намного ниже, чем стоимость такого же объема памяти на специализированных устройствах хранения данных [24].

Выполнение задач в MapReduce

MapReduce — среда разработки, позволяющая писать код для обработки больших наборов данных в распределенной файловой системе, такой как HDFS. Проще всего это понять на том же примере анализа журнала веб-сервера, который был рассмотрен в подразделе «Простой анализ журнала» раздела 10.1. Обработка данных в MapReduce очень похожа на этот пример.

1. Прочитать набор входных файлов и разделить их на *записи*. В случае журнала веб-сервера каждая запись является одной строкой журнала (то есть `\n` служит разделителем записей).
2. Вызвать функцию сопоставления, чтобы извлечь ключ и значение для каждой входной записи. В предыдущем примере роль функции сопоставления выполняла команда `awk '{print $7}'`: она извлекала URL (`$7`) в качестве ключа и оставляла значение пустым.
3. Отсортировать все пары «ключ — значение» по ключу. В случае журнала это выполняется с помощью первой команды `sort`.
4. Для всех отсортированных пар «ключ — значение» вызвать функцию сжатия. Если ключ повторяется несколько раз, то в отсортированном списке эти записи идут последовательно, так что их значения легко объединить и не хранить в памяти лишние строки. В предыдущем примере сжатие реализовано с помощью команды `uniq -c`, которая подсчитывает количество смежных записей с одним и тем же ключом.

Четыре представленных этапа могут выполняться в одной задаче MapReduce. На этапах 2 (сопоставление) и 4 (сжатие) надо написать свой код обработки данных. Этап 1 (разбиение файлов на записи) реализуется синтаксическим анализатором формата ввода. Этап 3, сортировка, выполняется в MapReduce по

умолчанию — вам не нужно писать его код, потому что выходные данные функции сопоставления, передаваемые на сжатие, всегда отсортированы.

Чтобы создать задачу MapReduce, необходимо реализовать две функции обратного вызова, сопоставление и сжатие, которые ведут себя следующим образом (см. также подраздел «Выполнение запросов с помощью MapReduce» раздела 2.2).

- *Сопоставление*. Функция сопоставления вызывается один раз для каждой входной записи, ее задача — извлечь ключ и значение записи. Для каждого набора входных данных может быть сгенерировано любое количество пар «ключ — значение» (в том числе ни одной). При переходе от одной входной записи к следующей никакие промежуточные состояния не сохраняются, каждая запись обрабатывается независимо от других.
- *Сжатие*. Среда разработки MapReduce принимает пары «ключ — значение», созданные функцией сопоставления, объединяет все значения, соответствующие одному ключу, и вызывает для каждой такой группы значений функцию сжатия. Последняя может создавать выходные записи (например, количество вхождений одного и того же URL).

В примере с журналом веб-сервера на с. 452 в качестве шага 5 была использована вторая команда `sort`, которая располагала URL в порядке убывания запросов. В MapReduce при необходимости второй сортировки можно написать вторую задачу и подать на ее вход выходные данные первой задачи. Таким образом, роль функции сопоставления заключается в том, чтобы привести данные к виду, удобному для сортировки, а роль функции сжатия — в обработке отсортированных данных.

Распределенные вычисления в MapReduce

Основное отличие MapReduce от конвейеров команд Unix заключается в том, что в MapReduce можно распараллеливать вычисления на нескольких машинах, и для этого не требуется писать код, явно реализующий параллелизм. Сопоставление и сжатие обрабатывают записи одну за другой; им нет нужды знать, откуда поступают записи и куда выводятся, вследствие чего среда разработки может взять на себя задачу перемещения данных между машинами.

Для сопоставления и сжатия в распределенных вычислениях можно использовать стандартные инструменты Unix [25], но чаще они реализуются в виде функций, написанных на выбранном языке программирования. В Hadoop MapReduce сопоставление и сжатие представлены в виде классов Java, каждый из которых реализует соответствующий интерфейс. В MongoDB и CouchDB сопоставление и сжатие — это функции JavaScript (см. подраздел «Выполнение запросов с помощью MapReduce» раздела 2.2).

На рис. 10.1 показан поток данных в задаче Hadoop MapReduce. Его распараллеливание основано на секционировании (см. главу 6): входные данные для задачи обычно представлены в виде каталога HDFS, каждый файл или файловый блок которого

считается самостоятельной секцией, обрабатываемой функцией сопоставления в качестве отдельной записи (на рис. 10.1 обозначены как m 1, m 2 и m 3).

Объем каждого входного файла обычно составляет сотни мегабайт. Планировщик задач MapReduce (на схеме не показан) старается запустить функцию сопоставления на каждом компьютере, где хранится копия входного файла, если только там достаточно оперативной памяти и ресурсов процессора для выполнения сопоставления [26]. Этот принцип известен под названием *«где данные, там вычисления»* [27]: он сохраняет копию входного файла в сети, уменьшая нагрузку на сеть и увеличивая локальность.

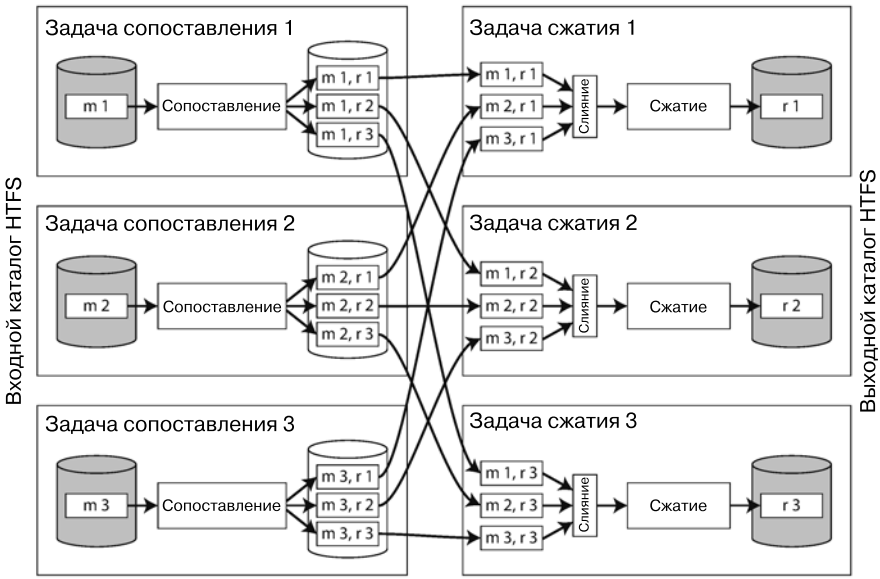


Рис. 10.1. Работа MapReduce с тремя картографами и тремя редукторами

В большинстве случаев код приложения, которое должно выполняться на этапе сопоставления, еще отсутствует на том компьютере, где ему следует выполняться, поэтому среда MapReduce сначала копирует код (например, в случае программы Java — файлы JAR) на соответствующие машины. Затем она запускает задачу сопоставления и начинает чтение входного файла, передавая функции сопоставления по одной записи. Результатом сопоставления является набор из пар «ключ — значение».

Этап сжатия также секционирован между разными машинами. Если количество задач сопоставления определяется количеством входных файловых блоков, то количество задач сжатия определяется автором задачи (и не всегда равно количеству задач сопоставления). Чтобы все пары «ключ — значение» с одним и тем же ключом попали на обработку одной и той же функции сжатия, в среде разработки

используется хеш ключей. По нему система определяет, какой задаче сжатия надо передать определенную пару «ключ — значение» (см. раздел «Секционирование по хешу ключа» раздела 6.2).

Пары «ключ — значение» должны быть отсортированы, но набор данных, скорее всего, слишком велик для применения обычных алгоритмов сортировки на одной машине. Вместо этого сортировка выполняется поэтапно. Вначале каждая задача сопоставления разбивает свой результат на части для сжатия на основе хеша ключей. Каждая такая секция записывается в отсортированный файл на локальном диске того компьютера, где выполняется сопоставление. Для этого используется техника, аналогичная описанной в подразделе «SS-таблицы и LSM-деревья» раздела 3.1.

Всякий раз, когда функция сопоставления завершает чтение входного файла и записывает отсортированные выходные файлы, планировщик MapReduce уведомляет функции сжатия о том, что они могут начать получать входные файлы от этой функции сопоставления. Функции сжатия подключаются к функциям сопоставления и загружают файлы с отсортированными парами «ключ — значение» — каждая для своей секции. Процесс секционирования по принадлежности к конкретной функции сжатия, сортировки и копирования разделов данных с этапа сопоставления на этап сжатия называется *перетасовкой* [26] (неточный термин — в отличие от перетасовки колоды карт в MapReduce нет ничего случайного).

Функция сжатия берет файлы после сопоставления и объединяет их, сохраняя порядок сортировки. Таким образом, если разные функции сопоставления создали записи с одинаковыми ключами, то в объединенном входном файле сжатия такие записи будут смежными.

Функция сжатия вызывается по ключу. И итератор последовательно сканирует все записи с одинаковым ключом (которые не всегда полностью помещаются в памяти). Для обработки этих записей функция сжатия может использовать любую логику и генерировать любое количество выходных записей. Последние помещаются в файл распределенной файловой системы (обычно один экземпляр на локальном диске машины, на которой работает функция сжатия, с репликами на других машинах).

Потоки MapReduce

Круг проблем, решаемых с помощью одной задачи MapReduce, ограничен. Возвращаясь к примеру анализа журнала, благодаря одной задаче MapReduce можно узнать количество просмотров страницы с заданным URL, но не самые популярные адреса, поскольку здесь требуется второй этап сортировки.

Поэтому задачи MapReduce часто объединяются в *потоки*, где выходные данные одной задачи становятся входными для следующей. В среде разработки Hadoop

MapReduce нет специальной поддержки рабочих процессов, так что цепочки создаются неявно, по имени каталога: первая задача настраивается на запись выходных данных в выбранный каталог HDFS, а следующая — на чтение входных данных из него. С точки зрения MapReduce это две независимые задачи.

Таким образом, цепочки задач MapReduce похожи не столько на конвейеры команд Unix (передающие выходные данные одного процесса на вход другого напрямую, используя только небольшой буфер в памяти), сколько на последовательность команд, выходные данные которых каждый раз записываются во временный файл, а из него каждая следующая команда считывает данные как входные. Такая система имеет свои преимущества и недостатки; мы обсудим их в подразделе «Материализация промежуточного состояния» раздела 10.3.

Выходные данные пакетной задачи считаются корректными только при ее успешном завершении (MapReduce отбрасывает частичный вывод задачи при неудачном завершении). Поэтому очередная задача потока может начать выполняться только после того, как предыдущие задачи, то есть те, что создают ее входные каталоги, успешно завершены. Для обработки этих зависимостей между выполнением задач были разработаны различные планировщики потоков для Hadoop, такие как Oozie, Azkaban, Luigi, Airflow и Pinball [28].

В этих планировщиках также реализованы функции управления, которые могут быть полезны при обработке большого количества пакетных задач. При создании рекомендательных систем [29] часто встречаются потоки, состоящие из 50–100 задач MapReduce, а в больших организациях разные команды могут выполнять разные задачи, использующие результаты друг друга. Для управления такими сложными потоками данных необходимы соответствующие инструменты.

Высокоуровневые инструменты для Hadoop, такие как Pig [30], Hive [31], Cascading [32], Crunch [33] и FlumeJava [34], позволяют создавать и автоматически объединять потоки, состоящие из нескольких этапов MapReduce.

Объединение и группировка на этапе сжатия

Мы обсуждали объединения в главе 2 в контексте моделей данных и языков запросов, однако не углубились в вопрос, как именно реализованы объединения. Пора снова обратиться к этой теме.

Во многих наборах данных реализованы взаимосвязи между записями: *внешние ключи* в реляционной модели, *ссылки на документ* в модели документов, *ребра* в модели графов. Объединения необходимы в тех случаях, когда некий код должен иметь доступ к записям, принадлежащим обеим сторонам этой ассоциации (как записи, содержащей ссылку, так и записи, на которую ссылаются). Как обсуждалось

в главе 2, денормализация позволяет уменьшить потребность в объединениях, но, как правило, не исключает ее полностью[1].

При выполнении в базе данных запроса, задействующего небольшое количество записей, для быстрого поиска интересующих записей обычно применяется *индекс* (см. главу 3). Если же в запросе использованы объединения, то может потребоваться несколько проходов индекса. Однако в MapReduce нет понятия индекса, по крайней мере не в обычном смысле.

Когда задача MapReduce получает набор файлов в качестве входных данных, она считывает содержимое всех этих файлов полностью; в БД такая операция называется *полным сканированием таблицы*. Если вы хотите прочитать только небольшое количество записей, то полное сканирование таблицы является чрезмерно затратным по сравнению с поиском по индексу. Однако в аналитических запросах (см. раздел 3.2) обычно приходится делать выводы, обрабатывая большое количество записей. В этом случае сканирование всего ввода вполне оправданно, особенно при наличии возможности распараллелить обработку данных на несколько машин.

Говоря об объединениях в контексте пакетной обработки, мы имеем в виду соответствие для всех элементов некоторого множества в наборе данных. Например, мы предполагаем, что задача обрабатывает данные для всех пользователей одновременно, а не просто ищет информацию для конкретного пользователя (для чего было бы гораздо эффективнее применять индекс).

Пример: анализ активности пользователя

На рис. 10.2 показан типичный пример объединения при пакетной обработке данных. Слева представлен журнал событий, описывающих действия зарегистрированных пользователей на сайте (известных как *события активности* или *история посещений*), а справа — база данных пользователей. Можно рассматривать этот пример как часть схемы типа «звезда» (см. подраздел «“Звезды” и “снежинки”: схемы для аналитики» раздела 3.2): журнал событий — таблица фактов, а база данных пользователей — одно из измерений.

Аналитическая задача может потребовать установить соответствие между действиями пользователя и информацией его профиля: например, если там указаны его возраст или дата рождения, то система способна определить, какие страницы

[1] Объединения, о которых говорится в настоящей книге, обычно являются эквивалентными. Это наиболее распространенный тип объединения, где одна запись связана с другими по признаку одинакового значения в определенном поле (например, ID). Отдельные базы данных поддерживают более общие типы объединений — например, вместо оператора равенства стоит оператор «меньше», но объем книги не позволяет рассмотреть эти варианты.

наиболее популярны у разных возрастных групп. Однако в событиях активности указан только ID пользователя, а не полная информация из его профиля. Очевидно, что включать всю информацию из профиля в каждое событие было бы слишком расточительным. Таким образом, необходимо объединить события активности с данными из БД о профилях пользователей.

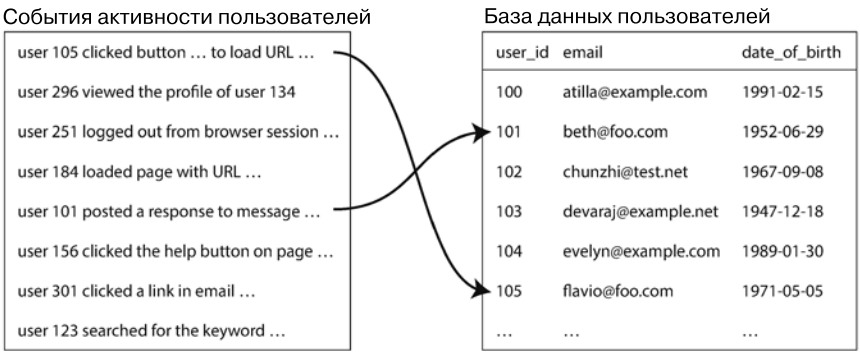


Рис. 10.2. Объединение между журналом активности пользователей и их базой данных

Простейшая реализация такого объединения — просматривать события активности по одному, каждый раз делая запрос в базу данных (на удаленном сервере) для каждого идентификатора пользователя, указанного в событии. Это возможно, но, скорее всего, производительность будет очень низкой: скорость обработки ограничена временем прохождения от сервера базы и назад, эффективность локального кэша очень зависит от распределения данных, одновременное выполнение большого количество запросов может привести к тому, что БД быстро перегрузится [35].

Чтобы обеспечить хорошую пропускную способность в пакетном процессе, вычисления должны выполняться (насколько возможно) локально, на одной машине. Запросы произвольного доступа по сети для каждой обрабатываемой записи совершаются слишком медленно. Более того, при запросе к удаленной БД пакетное задание становится недетерминированным, поскольку, пока оно выполняется, данные в удаленной базе могут измениться.

Поэтому лучше взять копию базы данных пользователей (например, создать резервную с помощью процесса ETL — см. подраздел «Складирование данных» раздела 3.2) и разместить ее в той же распределенной файловой системе, что и журнал активности пользователей. Тогда у нас будет один набор файлов HDFS с БД пользователей и другой — с записями об их активности, и можно применить MapReduce, чтобы собрать соответствующие записи в одном месте и эффективно их обработать.

Объединение с сортировкой слияния

Напомню, что задачей функции сопоставления является извлечение ключа и значения из каждой входной записи. В случае, показанном на рис. 10.2, ключом служит ID пользователя: один набор функций сопоставления просматривает события активности (извлекая идентификатор в качестве ключа и событие активности в качестве значения), а второй — базу данных пользователей (извлекая ID пользователя в качестве ключа и его дату рождения в качестве значения). Этот процесс показан на рис. 10.3.

Когда среда разработки MapReduce сортирует пары «ключ — значение», полученные в результате сопоставления, и затем разделяет их по ключам, эффект заключается в том, что все записи о событиях и пользователях с одним и тем же ID пользователя идут подряд и в таком виде поступают на вход функции сжатия. Задача MapReduce может даже отсортировать записи таким образом, чтобы на сжатие всегда сначала поступала запись из базы данных пользователей, а затем — события активности в порядке метки времени. Этот метод известен как *вторичная сортировка* [26].

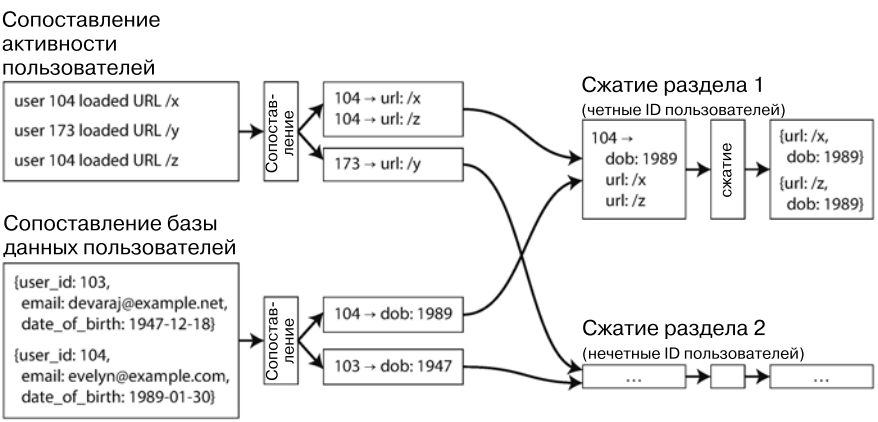


Рис. 10.3. Объединение с сортировкой слияния по ID пользователя на этапе сжатия. Если входные наборы данных разделены на несколько файлов, то каждый из них может обрабатываться несколькими функциями сжатия параллельно

Затем функция сжатия может легко выполнить логику объединения: функция сжатия вызывается один раз для каждого ID пользователя. Благодаря вторичной сортировке ожидается, что первым входным значением будет дата рождения из БД пользователя. Функция сжатия сохраняет дату рождения в локальной переменной, а затем перебирает все события активности для данного ID, выводя пары значений *«просмотренный URL — возраст в годах»*. Последующие задачи MapReduce могут затем посчитать распределение возраста посетителей для каждого URL и объединить адреса по возрастным группам.

Поскольку функция сжатия обрабатывает все записи для определенного ID пользователя за один проход, она хранит в памяти только одну запись пользователя за раз и никогда не делает запросы по сети. Этот алгоритм известен как *объединение с сортировкой слияния*, поскольку выходные данные функции сопоставления сортируются по ключу, а функции сжатия соединяют отсортированные списки записей, полученные с обеих сторон объединения.

Размещение связанных данных в одном месте

При объединении с сортировкой слияния благодаря сопоставлению и сортировке гарантируется, что все данные, необходимые для выполнения операции объединения по конкретному ID пользователя, окажутся в одном и том же месте и потребуется только один вызов функции сжатия. Благодаря тому что все необходимые данные подобраны заранее, сжатие может быть реализовано в виде довольно простого однопотокового кода, перебирающего записи с высокой скоростью и низкими издержками памяти.

Можно считать, что в этой архитектуре функции сопоставления «отправляют сообщения» функциям сжатия. Когда функция сопоставления генерирует пару «ключ — значение», ключ играет роль адреса, по которому должно быть доставлено значение. И хотя ключ представляет собой произвольную строку (а не фактический сетевой адрес, такой как IP или номер порта), он ведет себя подобно адресу: все пары «ключ — значение» с одним ключом будут доставлены в один пункт назначения (одной и той же функции сжатия).

С помощью модели программирования MapReduce удалось отделить аспекты коммуникации в физической сети (получение данных на конкретном компьютере) от логики приложения (обработка данных после их получения). Такое разделение в корне отличается от типичного использования БД, где запрос на получение данных из базы часто обрабатывается где-то глубоко внутри кода приложения [36]. Поскольку модель MapReduce поддерживает все варианты сетевых коммуникаций, она также избавляет приложение от необходимости обрабатывать частичные отказы, такие как сбой другого узла: MapReduce прозрачно повторяет неудачно завершенные задачи, не затрагивая логику приложения.

GROUP BY

Кроме объединений, есть еще одно типичное применение шаблона «размещение связанных данных в одном месте» — группировка записей по заданному ключу (как в операторе `GROUP BY` в SQL). Все записи с одним и тем же ключом объединяются в группу, после чего в каждой из них часто выполняется какое-либо обобщение, например:

- подсчет количества записей в каждой группе (в нашем примере — количество просмотров страниц, что можно было бы описать как агрегацию `COUNT (*)` в SQL);

- добавление значений в конкретном поле (`SUM (имя поля)`) в SQL);
- выбор k первых записей в соответствии с некоторой функцией ранжирования.

Простейшей реализацией подобной группировки в MapReduce является такая настройка функций сопоставления, чтобы при генерации пар «ключ — значение» использовался определенный ключ группировки. Затем в процессе разбиения и сортировки все записи с одним ключом объединяются в одной функции сжатия. Таким образом, при реализации в MapReduce группировка и объединение на первый взгляд выглядят весьма похоже.

Еще одно распространенное применение группировки — сопоставление всех событий для одной сессии с целью узнать последовательность действий пользователя. Этот процесс называется *сессионизацией* [37]. Например, с помощью такого анализа можно определить, какие пользователи чаще совершают покупки: те, кто уже видел новую версию сайта, или те, кто видел только старую (тестирование A/B), а также рассчитать, насколько действенным является тот или иной маркетинговый ход.

Если пользовательские запросы обрабатываются несколькими веб-серверами, то события, касающиеся каждого пользователя, скорее всего, разбросаны в файлах журналов разных серверов. Можно реализовать сессионизацию, задействуя в качестве ключа группировки cookie сессии, ID пользователя или другой подобный идентификатор, и собрать все события активности для конкретного человека в одном месте, в то время как события разных людей будут размещены по различным разделам.

Обработка асимметрии

Если данных, связанных с одним ключом, очень много, то шаблон «собрать все записи с одним ключом в одном месте» ломается. Например, в социальной сети большинство пользователей могут быть связаны с несколькими сотнями человек, но у небольшого количества знаменитостей могут быть миллионы подписчиков. Такие непропорционально активные записи БД называют *ключевыми объектами* [38] или *горячими ключами*.

Сбор всех данных об активности знаменитости (например, ответы на ее публикации) в одной функции сжатия может привести к значительной *асимметрии*, (такие места также называют *«горячими точками»*): один редуктор должен обработать значительно больше записей, чем другие (см. подраздел «Асимметричные нагрузки и разгрузка горячих точек» раздела 6.2). Поскольку задача MapReduce завершается только после того, как будут выполнены все функции сопоставления и сжатия, все последующие задачи могут начать работу только после окончания самого медленного сжатия.

Если среди входных данных объединения есть горячие ключи, существует несколько алгоритмов, которые можно использовать для их компенсации. Например,

метод *асимметричного объединения* в Pig сначала выполняет выборку образцов, чтобы определить, какие ключи являются горячими [39]. При выполнении фактического объединения функции сопоставления отправляют все записи, относящиеся к горячим ключам, одной из нескольких функций сжатия, выбранных случайным образом (в отличие от обычной MapReduce, которая выбирает функцию сжатия однозначно, по хешу ключа). Для остальных входных данных объединения записи, относящиеся к горячему ключу, должны быть реплицированы для *всех* функций сжатия, обрабатывающих этот ключ [40].

Таким образом обработка горячего ключа распределяется между несколькими функциями сжатия. Это позволяет лучше распараллеливать задачу ценой необходимости дублировать входные данные для нескольких функций сжатия. В Crunch реализован аналогичный метод *сегментированного объединения*, но он требует, чтобы горячие ключи указывались явно, без выборки образцов. Указанный метод также очень похож на тот, который обсуждался в подразделе «Асимметричные нагрузки и разгрузка горячих точек» раздела 6.2, с рандомизацией для разгрузки горячих ключей в секционированной БД.

В Hive реализован альтернативный подход к оптимизации асимметрии. Он требует, чтобы горячие ключи указывались явно в метаданных таблицы. Записи, относящиеся к этим ключам, хранятся в особых файлах, отдельно от остальных. При выполнении объединения в данной таблице для горячих ключей используется объединение на этапе сопоставления (см. следующий подраздел).

При группировке и сборе записей по горячему ключу можно провести группировку в два этапа. На первом MapReduce отправляет записи случайной функции сжатия, так что каждая из них выполняет группировку по подмножеству записей для горячего ключа и выводит для каждого ключа более компактное агрегированное значение. На втором MapReduce объединяет значения, полученные от всех функций сжатия, в одно для каждого ключа.

Объединения на этапе сопоставления

Алгоритмы объединения, описанные в предыдущем подразделе, выполняют фактическое объединение в функциях сжатия и соответственно называются *объединениями на этапе сжатия*. Функции сопоставления берут на себя подготовку входных данных: извлечение ключей и значений из каждой входной записи, передачу пар «ключ — значение» функциям сжатия и сортировку по ключу.

Преимущество объединения на этапе сжатия в том, что не нужно делать какие-либо предположения о входных данных: какими бы ни были их свойства и структура, при сопоставлении данные будут подготовлены для объединения. Однако есть и недостаток: вся эта сортировка, копирование для сжатия и объединение входных данных сжатия могут потребовать довольно больших вычислительных затрат.

В зависимости от доступного объема буферов памяти при прохождении по этапам MapReduce данные могут несколько раз записываться на диск [37].

Впрочем, когда *удается* сделать определенные предположения о входных данных, можно ускорить объединение, используя так называемое *соединение на этапе сопоставления*. При таком подходе применяется сокращенная задача MapReduce, в которой нет сжатия и сортировки. Вместо этого каждая функция сопоставления просто считывает один блок входных файлов из распределенной файловой системы и записывает в файловую систему один выходной файл — и больше ничего.

Широковещательное объединение по хешу

Самое простое объединение на этапе сопоставления применяется в том случае, когда большой набор данных объединяется с малым. Для этого последний должен полностью помещаться в памяти в каждой функции сопоставления.

Например, представим: в примере, показанном на рис. 10.2, база данных пользователей достаточно мала, чтобы помещаться в памяти. В таком случае функция сопоставления может сначала считать эту БД из распределенной файловой системы в находящуюся в памяти хеш-таблицу, а затем сканировать события активности и просто находить ID пользователя для каждого события в хеш-таблице[1].

В этом случае тоже может быть несколько задач сопоставления: по одной для каждого входного файлового блока при наличии большого набора входных данных для объединения (в примере на рис. 10.2 события активности — большого набора входных данных). Каждая из функций сопоставления полностью загружает малый набор данных в память.

Этот простой, но эффективный алгоритм называется *широковещательным объединением по хешу*: слово *«широковещательное»* здесь отражает тот факт, что каждая функция сопоставления для раздела большого набора данных считывает всю информацию о малом наборе (так что он, в сущности, «широковещательно» передается всем разделам большого), а слово *«хеш»* отражает использование хеш-таблицы. Этот метод объединения поддерживается в Pig (под названием «реплицируемое объединение»), Hive (MapJoin), Cascading и Crunch. Он также применяется в механизмах запросов складов данных, таких как Impala [41].

Вместо того чтобы загружать малый набор данных в хранящуюся в памяти хеш-таблицу, можно сохранить его на локальном диске в виде индекса, предназначенного

[1] В этом примере предполагается: для каждого ключа в хеш-таблице имеется ровно одна запись, что, вероятно, верно в случае базы данных пользователей (ID однозначно идентифицирует пользователя). В общем случае хеш-таблица может содержать несколько записей с одинаковыми ключами, и оператор объединения выводит все совпадения для текущего ключа.

только для чтения [42]. Часто используемые составляющие этого индекса будут оставаться в кэше страниц операционной системы, вследствие чего при таком подходе поиск с произвольным доступом может выполняться почти так же быстро, как и в случае хранения хеш-таблицы в памяти, но без требования, чтобы весь набор данных помещался в памяти.

Секционированное хеш-объединение

Если входные данные для объединения на этапе сопоставления разделены одинаково, то объединение по хешу может выполняться для каждого раздела в отдельности. В случае, показанном на рис. 10.2, можно организовать для событий активности и базы данных пользователей разбивку на разделы по последнему десятичному разряду в ID (по десять записей на раздел). Например, функция сопоставления 3 сначала загружает в хеш-таблицу всех пользователей с идентификатором, заканчивающимся на 3, а затем просматривает все события активности для каждого из этих пользователей.

Если разделение выполнено правильно, то вы можете быть уверены: все записи, которые требуется объединить, расположены в одном и том же нумерованном разделе и, следовательно, каждой функции сопоставления достаточно прочитать только один раздел из каждого входного набора данных. Преимущество такого подхода в том, что каждая функция сопоставления загружает в свою хеш-таблицу меньшее количество данных.

Описанный подход работает только в случае, когда оба набора входных данных для объединения имеют одинаковое количество разделов, причем записи заносятся в разделы на основе одного и того же ключа и одной и той же хеш-функции. Это разумное предположение, если входные данные генерируются предыдущими задачами MapReduce, уже выполнившими такую группировку.

В Hive разделенное хеш-объединение называется *сегментным объединением на этапе сопоставления* [37].

Объединение слиянием на этапе сопоставления

Если входные наборы данных не только разделены одинаково, но и *сортируются* по одному ключу, то применяется другой вариант объединения на этапе сопоставления. В таком случае неважно, помещаются ли входные данные в памяти, поскольку функция сопоставления может выполнять ту же операцию слияния, которая обычно выполнялась на этапе сжатия: читая по одной записи из обоих входных файлов в порядке возрастания значения ключа и сопоставляя записи с одинаковыми ключами.

Если объединение слияния на этапе сопоставления возможно, то это, очевидно, означает, что при совершении предыдущих задач MapReduce входные наборы

данных были приведены к этому секционированному и отсортированному виду. В принципе, такое объединение можно было бы выполнить в предыдущей задаче на этапе сокращения. Тем не менее иногда целесообразно произвести объединение слияния в отдельной задаче, где используется только сопоставление, — например, когда секционированные и отсортированные наборы входных данных, помимо этого конкретного объединения, необходимы и для других целей.

Потоки MapReduce с объединением на этапе сопоставления

Когда результат объединения MapReduce подается на вход следующей задачи, выбор между объединением на этапе сопоставления или сжатия влияет на структуру выходных данных. Результат объединения на этапе сжатия секционирован и отсортирован по ключу объединения, тогда как результат объединения на этапе сопоставления секционирован и отсортирован аналогично большему набору входных данных (поскольку задача сопоставления запускается для каждого файлового блока большого набора входных данных, независимо от того, используется ли разделенное или широковещательное соединение).

Как уже обсуждалось, объединения на этапе сопоставления также позволяют сделать больше предположений о размере, сортировке и разбиении входных наборов данных. Знание о физическом размещении наборов в распределенной файловой системе становится важным при оптимизации стратегий объединения: недостаточно знать только формат кодировки и имя каталога, хранящего данные; необходимо знать и количество разделов и ключи, по которым данные секционируются и сортируются.

В экосистеме Hadoop такие метаданные о разделении наборов данных часто хранятся в HCatalog и метахранилище Hive [37].

Выходные данные пакетных потоков

Мы много говорили о различных алгоритмах для реализации потоковых задач MapReduce, но до сих пор пренебрегали важным вопросом: каков результат всей обработки, после того как она будет закончена? Почему все эти задачи выполняются в первую очередь?

В случае запросов к базе данных мы различали цели обработки транзакций (OLTP) и аналитические цели (см. раздел 3.2). Мы увидели, что OLTP-запросы обычно перебирают небольшое количество записей по ключу и затем применяют индексы, чтобы представить их пользователю (например, на веб-странице). Аналитические запросы, напротив, часто просматривают большое количество записей, выполняя группировки и объединения, а выходные данные нередко имеют форму отчета: график, показывающий изменение метрики во времени или десять лучших элементов,

отобранных по определенному показателю, либо разделение некоего множества на категории. Получателем такого отчета часто является аналитик или менеджер, которому необходимо принимать бизнес-решения.

В каком случае применима пакетная обработка? Это не обработка транзакций, не аналитика. Она ближе к последней, поскольку пакетный процесс обычно сканирует большие порции входного набора данных. Однако поток задач MapReduce — это не то же самое, что SQL-запрос, используемый для аналитических целей (см. подраздел «Сравнение Hadoop и распределенных баз данных» текущего раздела). Выходные данные пакетного процесса часто являются не отчетом, а несколько другой структурой.

Построение поисковых индексов

Первоначально Google использовал MapReduce для построения индексов в своей поисковой системе. Это было реализовано как поток из 5–10 заданий MapReduce [1]. И хотя позже Google отказался от применения MapReduce для данной цели [43], сам факт помогает понять MapReduce, посмотрев на нее с точки зрения построения поискового индекса. (Даже сегодня Hadoop MapReduce остается хорошим способом создания индексов для Lucene/Solr [44].)

В пункте «Полнотекстовый поиск и нечеткие индексы» подраздела «Другие индексные структуры» раздела 3.1 мы кратко рассмотрели, как работает индекс полнотекстового поиска, такой как Lucene: это файл (словарь терминов), позволяющий быстро найти конкретное ключевое слово и ID всех документов, его содержащих (список проводок). Описанное — очень упрощенный вид поискового индекса: на практике он требует различных дополнительных данных, чтобы ранжировать результаты поиска по релевантности, исправлять орфографические ошибки, использовать синонимы и т. п., но принцип сохраняется.

Если необходимо выполнить полнотекстовый поиск по фиксированному набору документов, то пакетный процесс — очень эффективный способ построения индекса: функции сопоставления разбивают набор документов, каждая функция сжатия строит индекс для своего раздела, индексные файлы записываются в распределенную файловую систему. Построение таких секционированных по документам индексов (см. раздел 6.3) очень хорошо распараллеливается.

Поскольку запрос в индекс поиска по ключевому слову является операцией только чтения, индексные файлы не меняются после их создания.

Если проиндексированный набор документов периодически меняется, то один из вариантов — периодически повторять процесс индексирования для всего набора документов и по его окончании заменять все старые индексные файлы на новые. При изменении лишь немногих документов такой подход может потребовать слишком больших вычислительных затрат, но его преимущество в том, что про-

цесс индексирования очень прост для понимания: на вход подаются документы, на выходе получаем индекс.

В качестве альтернативы можно наращивать индекс поэтапно. Как описано в главе 3, чтобы добавлять, удалять и обновлять документы в индексе, Lucene записывает новые файлы сегментов и асинхронно объединяет и уплотняет файлы сегментов в фоновом режиме. Подробнее такую постепенную обработку мы обсудим в главе 11.

Хранилища пар «ключ — значение» как выходные данные пакетного процесса

Поисковые индексы — лишь один пример возможного результата процесса пакетной обработки. Другим распространенным применением этой обработки является создание систем машинного обучения, таких как классификаторы (например, спам-фильтры, обнаружение аномалий, распознавание изображений) и системы рекомендаций (например, людей, которых вы можете знать, товаров, потенциально способных вас заинтересовать, и т. п. [29]).

Результатом выполнения этих пакетных задач часто является база данных: например, такая, к которой можно делать запросы по ID пользователя и получать рекомендованных для него друзей, или база, к которой можно делать запросы по ID продукта и получать списки связанных продуктов [45].

Запросы к таким БД обычно поступают из веб-приложений, обрабатывающих запросы пользователей, которые обычно отделены от инфраструктуры Hadoop. Каким образом результат пакетного процесса возвращается в базу данных, откуда веб-приложение может его получить?

Самый очевидный вариант — использование клиентской библиотеки для выбранной базы данных, включающей в себя функции сопоставления или сжатия, и сохранение результатов пакетной задачи прямо на сервере базы, по одной записи за раз. Алгоритм будет работать (если брандмауэр разрешает прямой доступ из среды Hadoop к базам данных), но это плохая идея по нескольким причинам.

- Как уже обсуждалось в контексте объединений, отдельный сетевой запрос для каждой записи выполняется на порядки медленнее, чем обычная пропускная способность пакетной задачи. Даже если клиентская библиотека поддерживает пакетную обработку, производительность, вероятно, будет низкой.
- Задачи MapReduce часто запускают множество операций конкурентно. Если все функции сопоставления и сжатия одновременно будут записывать данные в одну и ту же выходную базу с ожидаемой скоростью пакетного процесса, то эта БД может быть легко перегружена и ее производительность для выполнения запросов пострадает. Это, в свою очередь, способно вызвать проблемы эксплуатации в других частях системы [35].

- Как правило, для выходных данных MapReduce предоставляет простую гарантию «все или ничего»: при успешном выполнении задачи ее результатом является результат однократного совершения всех операций, даже если часть из них не выполнялась и их пришлось повторить; в случае неудачного завершения всей задачи выходные данные не генерируются. Тем не менее запись во внешнюю систему из задачи создает внешне заметные побочные эффекты, которые невозможно скрыть таким образом. Поэтому приходится позаботиться о том, чтобы результаты частично выполненных задач, а также особенности попыток выполнить задачи Hadoop и спекулятивного исполнения были видны другим системам.

Гораздо лучшее решение — создать новую базу данных *внутри* пакетной задачи и записать ее в виде файлов в выходной каталог задачи в распределенной файловой системе, как поисковые индексы в предыдущем подразделе. Впоследствии эти файлы с данными не изменяются и могут быть загружены все вместе на серверы, обрабатывающие запросы на чтение (без записи). Для создания файлов базы данных в задачах MapReduce используются такие хранилища типа «ключ — значение», как Voldemort [46], Terrapin [47], ElephantDB [48] и HBase (в режиме пакетной загрузки) [49].

Создание таких файлов базы данных — хорошее применение MapReduce: сопоставление для извлечения ключа и сортировка по этому ключу является значительной частью работы по созданию индекса. Поскольку большинство подобных хранилищ пар «ключ — значение» предназначены только для чтения (файлы могут быть записаны лишь один раз, при выполнении пакетной задачи, и затем не изменяются), структуры данных довольно просты. Например, они не требуют WAL (см. пункт «Обеспечение надежности B-деревьев» подраздела «B-деревья» раздела 3.1).

При загрузке данных в систему Voldemort сервер продолжает обслуживать запросы к старым файлам данных, в то время как новые файлы копируются из распределенной файловой системы на локальный диск сервера. Когда копирование завершено, сервер автоматически переключается на запросы к новым файлам. Если в процессе что-то пойдет не так, то он снова вернется к старым файлам, поскольку они все еще существуют и не изменялись [46].

Философия выходных данных пакетного процесса

Рассмотренная ранее в этой главе философия Unix (см. подраздел «Философия Unix» раздела 10.1) побуждает экспериментировать, очень четко описывая поток данных: программа считывает свои входные данные и записывает выходные. При этом входные данные не изменяются, а любые выходные предыдущей программы полностью заменяются новыми; никаких других побочных эффектов не существует. Это значит, что можно повторно запускать команду сколько угодно раз, настраивать или отлаживать ее, не нарушая состояния системы.

Обработка выходных данных с помощью задач MapReduce соответствует этой философии. Рассматривая входные данные как неизменные и избегая побочных эффектов

(таких как запись во внешние БД), пакетные задачи не только обеспечивают хорошую производительность, но и становятся намного проще в обслуживании.

- При ошибке в коде, когда результат ошибочен или поврежден, достаточно вернуться к предыдущей версии кода и повторить задачу, и результат снова будет правильным. Или, что еще проще, можно сохранить старые выходные данные в другом каталоге и всего лишь вернуться к нему. Базы данных с транзакциями чтения и записи не обладают таким свойством: если выполнить ошибочный код, который запишет некорректные данные в базу, то возврат к правильному коду не приведет к исправлению данных в базе. (Идея возможности восстановить данные после выполнения ошибочного кода называется *толерантностью к человеческим ошибкам* [50].)
- Следствием такой легкости отката является то, что разработка функций выполняется быстрее, чем в среде, где ошибки могут нанести непоправимый ущерб. Этот принцип *минимизации необратимости* полезен для разработки программного обеспечения Agile [51].
- В случае сбоя на этапе сопоставления или сжатия среда разработки MapReduce автоматически перенастраивает задачу и запускает ее снова с теми же входными данными. Если отказ произошел из-за ошибки в коде, то сбой повторится и после нескольких попыток это приведет к сбою задания. Но если отказ вызван кратковременной проблемой, то ошибка будет исправлена. Такие автоматические повторные попытки безопасны только потому, что входные данные неизменны, а результаты неудачных задач MapReduce отбрасывает.
- Один и тот же набор файлов может использоваться как входные данные для различных задач, включая мониторинг с вычислением метрик и оценкой того, имеет ли результат ожидаемые характеристики (например, путем сравнения с результатами предыдущего прохода и измерения несоответствий).
- Подобно инструментам Unix, в MapReduce логика отделена от коммуникаций (настройка каталогов ввода и вывода). Это обеспечивает разделение проблем и позволяет повторно использовать код: одна команда разработчиков может сосредоточиться на реализации задачи, хорошо выполняющей конкретную функцию, в то время как другие команды будут решать, где и когда выполнять эту задачу.

Итак, принципы проектирования, хорошо зарекомендовавшие себя в Unix, похоже, хорошо работают и для Hadoop. Но между Unix и Hadoop есть различия. Например, поскольку большинство инструментов Unix предполагает нетипизированные текстовые файлы, они уделяют большое внимание синтаксическому анализу ввода (см. пример анализа журнала в начале главы, где для извлечения URL используется `{print $7}`). В Hadoop некоторые из этих малоценных синтаксических преобразований устранены благодаря более структурированным форматам файлов. Так, часто применяются Avro (см. подраздел «Avro» раздела 4.1) и Parquet (см. раздел 3.3), поскольку обеспечивают эффективное кодирование по заданной схеме и позволяют со временем ее развивать (см. главу 4).

Сравнение Hadoop и распределенных баз данных

Как мы уже видели, система Hadoop похожа на распределенную версию Unix с файловой системой HDFS и MapReduce в роли причудливой реализации процесса Unix (который всегда запускает утилиту `sort` между этапами сопоставления и сжатия). Мы увидели, как с помощью этих простейших операций можно реализовать различные операции объединения и группировки.

Опубликованная статья о MapReduce [1] была в некотором смысле совсем не новой. Все алгоритмы обработки и параллельных объединений, обсуждавшиеся выше, уже были реализованы в так называемых базах данных с *массовым параллелизмом* (MPP) более десяти лет назад [3, 40]. Например, пионерами в этой области были процессор БД Gamma, Teradata и Tandem NonStop SQL [52].

Самое большое различие заключается в том, что базы данных MPP сосредотачиваются на параллельном выполнении аналитических SQL-запросов на компьютерном кластере, в то время как сочетание MapReduce и распределенной файловой системы [19] дает нечто гораздо большее, чем универсальная операционная система, способная запускать произвольные программы.

Разнообразие хранилищ данных

Базы данных требуют структурировать данные в соответствии с определенной моделью (например, реляционной или документной), тогда как файлы в распределенной файловой системе — это просто последовательности байтов, которые могут быть записаны с помощью любой модели данных и кодирования. Они могут содержать наборы записей базы или же тексты, изображения, видео, показания датчиков, разреженные матрицы, векторы свойств, последовательности генома и любые другие данные.

Коротко говоря, система Hadoop открыла возможность записывать в HDFS любые данные и только потом выяснять, как их обрабатывать [53]. В базах данных MPP, наоборот, обычно требуется сначала тщательно проработать структуру данных и шаблоны запросов и только потом импортировать данные в собственный формат хранения.

С точки зрения пуриста может показаться, что такое тщательное моделирование перед импортом предпочтительнее: это говорит о наличии у пользователей БД более качественных данных для работы. Однако на практике выясняется: быстро получить данные, даже представленные в странном, трудном для применения, необработанном формате, зачастую более ценно, чем пытаться заранее выбрать идеальную модель данных [54].

Эта идея аналогична складу данных (см. подраздел «Складирование данных» раздела 3.2): важно просто собрать данные из разных частей крупной системы в одном месте, поскольку тогда можно объединять различные наборы, которые ранее были

несопоставимыми. Тщательная проработка схемы, требуемая базами данных MPP, замедляет этот централизованный сбор данных. Сбор необработанной информации, с тем чтобы позже позаботиться о ее структуре, позволяет его ускорить (концепция, иногда называемая озером данных или корпоративным концентратором информации [55]).

Неизбирательный сбор данных смещает нагрузку по их интерпретации: вместо того чтобы заставить производителя набора привести его в стандартизованный формат, интерпретация данных становится задачей их потребителя (подход «схема при чтении» [56]; см. пункт «Гибкость схемы в документной модели» подраздела «Реляционные и документоориентированные базы данных сегодня» раздела 2.1). Это может быть преимуществом, если производитель и потребители — разные люди с несходными приоритетами. Возможно, вместо одной идеальной модели данных существует несколько разных точек зрения на информацию, в зависимости от целей. Простой сбор данных в их исходной форме позволяет выполнять несколько подобных преобразований. Такой подход был назван *принципом суши*: сырые данные лучше обработанных [57].

Таким образом, система Hadoop часто использовалась для реализации процессов ETL (см. подраздел «Складирование данных» раздела 3.2): данные из систем обработки транзакций сбрасываются в распределенную файловую систему в некоем необработанном виде, где затем с помощью задач MapReduce они обрабатываются, преобразуются в реляционную форму и импортируются в складе данных MPP для аналитических целей. Моделирование данных не отменяется, но выполняется на своем этапе, отдельно от их сбора. Такое разделение возможно благодаря тому, что распределенная файловая система поддерживает данные в любом формате.

Разнообразие моделей обработки

Базы данных MPP — монолитные, тесно интегрированные части программного обеспечения, обеспечивающие структуру хранилища информации на диске, планирование запросов, их расписание и выполнение. Поскольку эти компоненты настраиваются и оптимизируются для конкретных потребностей базы, система в целом способна достичь очень хорошей производительности для тех типов запросов, для которых она разработана. Кроме того, язык запросов SQL позволяет составлять выразительные запросы с изящной семантикой без необходимости писать код, что делает его доступным для графических инструментов, используемых бизнес-аналитиками (например, Tableau).

Однако не все виды обработки можно рационально сформулировать в виде SQL-запроса. Например, в системах машинного обучения и рекомендаций, или в индексах полнотекстового поиска с моделями ранжирования релевантности, или при анализе изображений, скорее всего, потребуется более общая модель обработки данных. Эти виды обработки зачастую очень зависят от конкретного приложения

(например, системы машинного обучения, модели естественного языка для машинного перевода, функции оценки риска для прогнозирования мошенничества), так что неизбежно требуют написания кода, а не только запросов.

Система MapReduce позволяет инженерам легко запускать разработанный ими код для больших наборов данных. При наличии пары HDFS-MapReduce *возможно* построить систему выполнения SQL-запросов поверх нее. Именно это сделано в проекте Hive [31]. Однако можно также написать много других форм пакетных процессов, которые не выражаются в виде SQL-запросов.

Впоследствии было обнаружено, что для отдельных видов обработки возможности MapReduce слишком ограниченны, а производительность низкая. Так появилось много других моделей обработки, и в первую очередь Hadoop (часть из них мы рассмотрим в разделе 10.3). Двух моделей, SQL и MapReduce, недостаточно: нужно больше разных моделей! Вследствие открытости платформы Hadoop удалось реализовать целый ряд подходов, которые не были бы возможны в рамках монолитной базы данных MPP [58].

Важно отметить: все эти модели обработки могут работать в одном общем вычислительном кластере, с доступом к одним и тем же файлам в распределенной файловой системе. В Hadoop не нужно импортировать данные в несколько специализированных систем для разных видов обработки: система достаточно гибкая, чтобы поддерживать разнообразный набор задач в одном кластере. Отсутствие необходимости постоянно перемещать данные намного облегчает извлечение из них ценной информации и упрощает проведение экспериментов с новыми моделями обработки.

Экосистема Hadoop включает в себя базы данных OLTP с произвольным доступом, такие как HBase (см. подраздел «SS-таблица и LSM-деревья» раздела 3.1) и аналитические базы данных MPP, такие как Impala [41]. Ни та ни другая не используют MapReduce, но обе задействуют HDFS для хранения информации. Несмотря на то что это совершенно разные подходы к доступу и обработке данных, они могут сосуществовать и интегрироваться в одной системе.

Проектирование на случай частых сбоев

При сравнении MapReduce с базой данных MPP обращают на себя внимание еще два принципиальных различия: обработка ошибок и применение памяти и диска. Пакетные процессы менее чувствительны к ошибкам, чем онлайновые системы, поскольку в случае сбоя не сразу влияют на пользователей и их всегда можно перезапустить.

Если узел дает сбой во время выполнения запроса, то большинство баз данных MPP прерывают выполнение всего запроса и либо предлагают пользователю отправить запрос повторно, либо перезапускают его автоматически [3]. Поскольку запросы обычно выполняются в течение нескольких секунд или максимум не-

скольких минут, то этот способ обработки ошибок является приемлемым, ведь стоимость повторной попытки не слишком велика. Базы данных MPP также предпочитают хранить как можно больше данных в памяти (например, задействуя хеш-объединения), чтобы уменьшить затраты времени на чтение диска.

MapReduce, напротив, допускает ошибки на этапах сопоставления или сжатия. Соответствующая операция просто совершается повторно, не затрагивая выполнение остальной задачи. Кроме того, активно используется запись данных на диск — отчасти для большей отказоустойчивости, отчасти из такого предположения: набор данных в любом случае слишком велик, чтобы поместиться в памяти.

Метод MapReduce подходит для более крупных задач: обрабатывающих так много данных и работающих так долго, что за это время, скорее всего, произойдет по крайней мере один сбой. В таком случае повторная загрузка всей задачи из-за сбоя одной операции будет чересчур расточительной. Даже если восстановление с точностью до отдельной операции приводит к дополнительным расходам, замедляющим обработку при отсутствии ошибок, это может быть разумным компромиссом в случае достаточно высокой вероятности сбоя при выполнении задачи.

Но насколько реалистичны эти предположения? В большинстве кластеров сбои происходят не очень часто — пожалуй, даже настолько редко, что большинство задач выполняется без них. Насколько оправданны столь значительные накладные расходы ради отказоустойчивости?

Чтобы понять причины экономного использования памяти и восстановления на уровне операций в MapReduce, полезно взглянуть на ту среду, для которой изначально создавалась система MapReduce. У Google есть многофункциональные центры обработки и хранения информации, где операционные онлайновые сервисы и автономные пакетные задания выполняются на одних и тех же машинах. Каждой задаче предоставляются определенные ресурсы (ядра процессора, оперативная память, дисковое пространство и т. п.), активизируемые с помощью контейнеров. У каждой задачи также есть приоритет: если задаче с более высоким приоритетом нужно больше ресурсов, то задачи с более низким приоритетом, выполняемые на том же компьютере, могут быть прерваны (выгружены), чтобы освободить ресурсы. Вдобавок приоритет определяет цену вычислительных ресурсов: команды разработчиков должны оплачивать ресурсы, которые они используют, и процессы с более высоким приоритетом стоят дороже [59].

Эта архитектура допускает чрезмерную нагрузку на непроизводственные (низкоприоритетные) вычислительные ресурсы, поскольку система при необходимости может их освободить. В свою очередь, избыточные ресурсы позволяют лучше использовать вычислительные мощности и повысить их эффективность по сравнению с системами, где разделяются производственные и непроизводственные задачи. Однако, поскольку задачи MapReduce выполняются с низким приоритетом, есть риск, что они будут прерваны в любое время, так как их ресурсы потребуются для процесса с более высоким приоритетом. Пакетные задачи эффективно «собирают

объедки под столом», задействуя любые вычислительные ресурсы, остающиеся после процессов с высоким приоритетом.

В Google вероятность того, что задача MapReduce, работающая в течение часа, будет прервана с целью освободить ресурсы для процесса с более высоким приоритетом, составляет примерно 5 %. Это более чем на порядок выше вероятности сбоя из-за проблем с оборудованием, перезагрузки компьютера или по другим причинам [59]. При такой возможности прерывания, если задача состоит из 100 операций, каждая из которых выполняется в течение 10 минут, вероятность того, что по крайней мере одна из них будет прервана до завершения, составляет более 50 %.

Именно поэтому система MapReduce рассчитана на частые неожиданные прерывания задач: не потому, что аппаратное обеспечение ненадежно, а для возможности произвольного прерывания процессов с целью лучше использовать ресурсы в вычислительном кластере.

Среди кластерных планировщиков с открытым исходным кодом приоритетное переключение процессов менее распространено. Функция CapacityScheduler в YARN поддерживает приоритетное переключение для равномерного распределения ресурсов в разных запросах [58], но на момент написания [60] в YARN, Mesos и Kubernetes приоритетное переключение процессов не поддерживалось. В среде, где выполнение задач прерывается не так часто, эти принципы MapReduce менее важны. В следующем разделе мы рассмотрим некоторые альтернативы MapReduce, в которых приняты другие архитектурные решения.

10.3. За пределами MapReduce

В конце 2000-х годов среда разработки MapReduce стала очень популярной и вызвала бурные обсуждения, но это всего лишь одна из многих возможных моделей программирования распределенных систем. В зависимости от объема данных, их структуры и типа обработки другие инструменты могут оказаться более эффективными для выполнения вычислений.

Тем не менее в данной главе мы уделили много внимания MapReduce, поскольку это полезный инструмент для обучения, очень ясная и простая абстракция над распределенной файловой системой. Под простотой здесь имеется в виду понимание того, что делает система, а не простота использования. Наоборот, реализация обработки с помощью «чистых» API MapReduce является довольно сложной и трудоемкой, например, все алгоритмы объединения приходится реализовать самостоятельно [37].

Ответом на трудность непосредственного использования MapReduce стали различные модели программирования более высокого уровня (Pig, Hive, Cascading, Crunch), созданные как абстракции на базе MapReduce. Если вы понимаете, как работает MapReduce, то их довольно легко изучить, а их построения более высо-

кого уровня значительно упрощают реализацию многих типичных задач пакетной обработки.

Однако и в самой модели обработки MapReduce существуют проблемы, которые нельзя устранить добавлением еще одного уровня абстракции. При отдельных видах вычислений они приводят к низкой производительности. С одной стороны, MapReduce очень надежна: ее можно применять для обработки практически сколь угодно больших объемов данных в ненадежной многопользовательской системе с частыми прерываниями задач, и работа все равно будет выполнена (пускай и медленно). С другой стороны, есть инструменты, работающие на порядок быстрее для некоторых видов вычислений.

В оставшейся части главы мы рассмотрим часть этих альтернатив для пакетной обработки информации. В главе 11 перейдем к потоковой обработке, которую можно рассматривать как еще один способ ускорения пакетной обработки.

Материализация промежуточного состояния

Как уже говорилось, каждая задача MapReduce не зависит от других задач. Главными точками соприкосновения задачи с внешним миром являются ее каталоги входных и выходных данных в распределенной файловой системе. Чтобы выходные данные одной задачи стали входными для другой, нужно сделать выходной каталог первой задачи входным каталогом второй, а внешний планировщик потока должен запустить вторую задачу только после того, как закончится выполнение первой.

Такая настройка является разумной, если выходные данные первой задачи представляют собой информацию, которую вы намерены широко распространить в своей организации. В таком случае нужно иметь возможность ссылаться на эту информацию по имени и многократно использовать ее в качестве входных данных для различных задач (включая задачи, созданные другими группами разработчиков). Размещение данных в известном месте распределенной файловой системы позволяет избежать связывания: задача не должна знать, кто создает для нее входные данные или кто потребляет результаты ее работы (см. пункт «Отделение логики от передачи данных» подраздела «Философия Unix» раздела 10.1).

Однако во многих случаях известно, что выходные данные задачи используются как входные только одной конкретной задачей, которая обслуживается той же командой разработчиков. В этом случае файлы в распределенной файловой системе играют роль всего лишь *промежуточного состояния*, обеспечивающего передачу данных от одной задачи к другой. В сложных процессах, применяемых для построения систем рекомендаций и состоящих из 50–100 задач MapReduce [29], встречается множество таких промежуточных состояний.

Процесс записи промежуточных состояний в файлы называется *материализацией*. (Этот термин нам встречался ранее в контексте материализованных представлений

в подразделе «Агрегирование: кубы данных и материализованные представления» раздела 3.3. Он означает следующее: нужно рассчитывать на результат некоей операции и записывать ее, вместо того чтобы вычислять его каждый раз по требованию.)

В примере с анализом журнала, рассмотренном в начале главы, наоборот, для подачи выходных данных одной команды на вход другой использовались конвейеры Unix. Вместо того чтобы полностью материализовать промежуточное состояние, конвейеры передают *поток* выходных данных на вход постепенно, применяя только небольшой буфер в памяти.

По сравнению с конвейерами Unix подход MapReduce с полностью материализованным промежуточным состоянием имеет следующие недостатки.

- Задача MapReduce запускается только после завершения всех операций предыдущих задач (генерирующих свои входные данные), в то время как процессы, объединенные в конвейер Unix, запускаются одновременно и выходные данные потребляются сразу же после их создания. Асимметрия, или различная нагрузка на разных машинах, означает, что в задаче есть несколько медленных операций, для выполнения которых требуется гораздо больше времени, чем для других. Необходимость дождаться завершения всех операций предыдущей задачи замедляет выполнение всего потока.
- Функции сопоставления часто являются избыточными: они просто читают тот же файл, который был только что написан функцией сжатия, и готовят его к следующему этапу секционирования и сортировки. Во многих случаях код функции сопоставления может быть частью предыдущей функции сжатия: если выходные данные функции сжатия были разбиты на разделы и отсортированы таким же образом, как и выходные данные функции сопоставления, то функции сжатия можно соединить напрямую, без чередования с этапом сопоставления.
- Сохранение промежуточного состояния в распределенной файловой системе означает репликацию этих файлов на нескольких узлах, что часто бывает избыточным для таких временных данных.

Подсистемы потока данных

Чтобы устранить эти проблемы с MapReduce, было разработано несколько новых механизмов для выполнения распределенных пакетных вычислений. Из них наиболее известны Spark [61, 62], Tez [63, 64] и Flink [65, 66]. Они сильно различаются по архитектуре, но имеют одну общую черту: весь рабочий процесс рассматривается как одна задача, без разделения на независимые подзадачи.

Поскольку в этих системах поток данных, проходящий через несколько этапов обработки, моделируется явно, они известны как *подсистемы потока данных*. Подобно MapReduce, их задача заключается в вызове пользовательской функции для обработки каждой записи потока. Их работа распараллеливается путем сек-

ционирования входных данных, а выходные данные каждой функции копируются по сети, чтобы стать входными для другой.

В отличие от MapReduce, эти функции не обязательно выполняют по очереди операции сопоставления и сжатия — их можно сочетать более гибкими способами. Такие функции называют *операторами*, и подсистема потока данных предоставляет следующие варианты передачи выходных данных одного оператора на вход другого.

- Один из вариантов заключается в повторном секционировании и сортировке записей по ключу, как на этапе перетасовки в MapReduce (см. пункт «Распределенные вычисления в MapReduce» подраздела «Выполнение задач в MapReduce» раздела 10.2). Это позволяет объединять результаты сортировки со слиянием и группировать данные так же, как в MapReduce.
- Еще одна возможность состоит в том, чтобы взять несколько наборов входных данных и секционировать их по одному и тому же принципу, но пропустить сортировку, сэкономив таким образом ресурсы на этапе распределенного хеш-объединения, где важно разделение записей, но их порядок не имеет значения, поскольку при построении хеш-таблицы последовательность записей все равно будет случайной.
- Для объединений с хеш-трансляцией один и тот же набор выходных данных от заданного оператора может быть отправлен на все разделы оператора объединения.

Этот стиль обработки реализован в таких исследовательских системах, как Dryad [67] и Nephele [68]. По сравнению с моделью MapReduce он имеет следующие преимущества.

- Затратные вычисления, такие как сортировка, выполняются только там, где это действительно необходимо, а не по умолчанию между каждым сопоставлением и сжатием.
- Не выполняется лишних сопоставлений, так как работа, выполняемая функциями сопоставления, часто может быть включена в предыдущий оператор сжатия (поскольку при сопоставлении секционирование набора данных не изменяется).
- Поскольку все объединения и зависимости данных в потоке объявлены явно, планировщик знает, какие данные и где требуются, и может выполнить местную оптимизацию. Например, он может попытаться разместить задачу, потребляющую те или иные данные, на том же компьютере, что и задачу, их генерирующую, чтобы передавать их через общий буфер памяти, а не копировать по сети.
- Как правило, достаточно, чтобы промежуточное состояние между операторами сохранялось в памяти или записывалось на локальный диск. Это требует меньше операций ввода-вывода, чем запись данных в HDFS (с репликацией

на несколько машин и записью каждой реплики на диск). MapReduce уже использует такую оптимизацию для выходных данных на этапе сопоставления, но в подсистемах потока данных указанная идея распространяется на все промежуточные состояния.

- Операторы начинают работу сразу, как только готовы их входные данные; нет необходимости ждать завершения всего предыдущего этапа, чтобы запустить следующий.
- Существующие процессы Java Virtual Machine (JVM) можно многократно использовать для запуска новых операторов. По сравнению с MapReduce (где запускается новая JVM для каждой задачи) это требует меньше вычислительных затрат.

Подсистемы потока данных можно использовать для тех же вычислений, что и потоки MapReduce, и благодаря описанным здесь оптимизациям они обычно выполняются гораздо быстрее. Поскольку операторы представляют собой обобщение операции сопоставления и сжатия, один и тот же код обработки способен выполняться в любой подсистеме: потоки, реализованные в Pig, Hive или Cascading, могут быть переведены с MapReduce на Tez или Spark с простой настройкой, без изменения кода [64].

Tez — очень небольшая библиотека, применяющая сервис перетасовки YARN для физического копирования данных между узлами [58]; Spark и Flink — большие структуры разработки, включающие в себя собственный уровень сетевой коммуникации, планировщик и пользовательские API. Мы обсудим эти высокоуровневые API ниже.

Отказоустойчивость

Преимущество полностью материализованного промежуточного состояния в распределенной файловой системе состоит в том, что оно может долго храниться. В MapReduce это обеспечивает отказоустойчивость: если задача дает сбой, то ее достаточно перезапустить на другом компьютере и снова прочитать тот же набор входных данных из файловой системы.

В Spark, Flink и Tez промежуточное состояние не записывается в HDFS, поэтому для устойчивости к отказам там используется другой подход: если на компьютере происходит сбой и промежуточное состояние на нем потеряно, то оно повторно вычисляется по другим данным, которые еще доступны (более раннее промежуточное состояние, при условии, что оно сохранено, в противном случае — исходные данные, которые обычно находятся в HDFS).

Для активизации повторных вычислений среда разработки должна отслеживать, как обрабатывается каждая часть информации: какие входные разделы были использованы и какие операторы были применены. В Spark задействуется абстракция

гибкого распределенного набора данных (resilient distributed dataset, RDD), которая позволяет отслеживать происхождение данных [61]; Flink проверяет состояние оператора, что позволяет возобновить его работу, если в процессе его выполнения произойдет сбой [66].

При повторной обработке данных важно знать, является ли вычисление *детерминированным*: другими словами, всегда ли оператор производит одинаковые выходные данные, если подавать одни и те же данные на вход? Этот важно в случае передачи следующим операторам некой части потерянных данных. Если оператор выполняется повторно, а пересчитанные данные не совпадают с предыдущими, которые были потеряны, то следующим операторам будет очень трудно устранить противоречия между старыми и новыми данными. При сбое недетерминированных операторов обычно также останавливают работу всех последующих операторов и снова запускают их с новыми данными.

Во избежание таких каскадных ошибок лучше сделать операторы детерминированными. Однако обратите внимание: легко ошибиться, решая, является ли поведение программы детерминированным. Например, многие языки программирования не гарантируют какого-либо конкретного порядка при переборе элементов хеш-таблицы; многие вероятностные и статистические алгоритмы явно полагаются на использование случайных чисел; любое применение системных часов или внешних источников данных является недетерминированным. Чтобы система была отказоустойчивой, подобные причины недетерминированности необходимо устранить, к примеру, генерируя псевдослучайные числа с помощью фиксированного начального числа.

Восстановление после сбоев путем повторной обработки данных не всегда является правильным решением: если промежуточные данные намного меньше исходных или если вычисления очень интенсивно используют ресурсы процессора, то, вероятно, будет дешевле материализовать промежуточные данные в файлы, чем вычислять их повторно.

Разные взгляды на материализацию

Возвращаясь к аналогии Unix, мы видим, что система MapReduce похожа на запись выходных данных каждой команды во временный файл, тогда как подсистемы потока данных подобны конвейерам Unix. В частности, явно конвейерное выполнение операций реализовано в Flink: выходные данные последовательно передаются от одного оператора к другому, обработка начинается сразу, не дожидаясь завершения ввода.

Операция сортировки неизбежно должна прочитать все входные данные, прежде чем сможет генерировать что-либо на выходе, поскольку возможно следующее: самая последняя входная запись содержит самый маленький ключ и, следовательно, должна быть самой первой выходной записью. Таким образом, любой оператор,

который требует сортировки, должен будет накапливать данные, по крайней мере временно. Но многие другие части потока могут выполняться по принципу конвейера.

Когда работа завершается, нужно записать выходные данные в какое-либо долговременное хранилище, чтобы пользователи могли их там найти и задействовать, — скорее всего, снова в распределенную файловую систему. Таким образом, в момент применения подсистемы потока данных материализованные наборы данных в HDFS по-прежнему являются по большей части входными и выходными данными задач. Как и в MapReduce, входные данные не изменяются, а выходные заменяются полностью. По сравнению с MapReduce усовершенствование заключается в том, что все промежуточные состояния тоже сохраняются в файловой системе.

Графы и итеративная обработка

В разделе 2.3 обсуждалось использование графов при моделировании данных и языков запросов по графам для поиска пересечений ребер и вершин на графе. В главе 2 главным образом обсуждалось применение OLTP-стиля: быстрое выполнение запросов для поиска небольшого количества вершин, соответствующих определенным критериям.

Интересно рассмотреть применение графов и в контексте пакетной обработки, где целью является выполнение какой-либо автономной обработки или анализа по всему графу. Данная потребность часто возникает в приложениях машинного обучения, таких как системы выдачи рекомендаций или системы ранжирования. Например, одним из самых известных алгоритмов анализа графов является PageRank [69], который оценивает популярность веб-страницы на основе того, какие еще веб-страницы ссылаются на нее. Этот алгоритм также используется в формуле, определяющей порядок представления результатов поисковыми системами.



Подсистемы потока данных, такие как Spark, Flink и Tez (см. подраздел «Материализация промежуточного состояния» текущего раздела), обычно строят последовательность операторов в задаче в виде ориентированного ациклического графа (directed acyclic graph, DAG). Это не то же самое, что графовая обработка: в подсистемах потока данных их передача от одного оператора к другому организована в виде графа, но сами они обычно состоят из кортежей реляционного типа. При графовой обработке сами данные имеют форму графа. Еще один неудачный термин, приводящий к путанице!

Многие графовые алгоритмы описываются как последовательность перемещений по ребрам, и одна вершина соединяется с другой соседней для распространения некоторой информации. Так повторяется до тех пор, пока не будет выполнено некое условие, например, не останется ребер для продвижения по ним или пока

не сойдутся отдельные метрики. В примере на рис. 2.6 был составлен список всех локаций из Северной Америки, хранящихся в базе данных, путем прохождения по ребрам, указывающим, какая локация принадлежит той или иной локации (такой тип алгоритмов называется *транзитивным замыканием*).

Можно хранить граф в распределенной файловой системе (в файлах, содержащих списки вершин и ребер), но идея «повторять до завершения» не может быть реализована в обычной MapReduce, поскольку там выполняется только один проход по данным. Поэтому такие алгоритмы часто реализуются в виде *итераций*.

1. Внешний планировщик запускает пакетный процесс для вычисления одного шага алгоритма.
2. Когда пакетный процесс завершен, планировщик проверяет, закончено ли вычисление (по условию завершения — например, не осталось ребер, по которым можно пройти, или изменения по сравнению с последней итерацией ниже некоего порога).
3. Если алгоритм не завершен, то планировщик возвращается к шагу 1 и запускает пакетный процесс еще раз.

Такой подход работает, но его реализация в MapReduce часто очень неэффективна, потому что MapReduce не учитывает итеративный характер алгоритма: система всегда будет считывать весь набор входных данных и каждый раз создавать совершенно новый выходной набор, даже если по сравнению с предыдущей итерацией изменилась только небольшая часть графа.

Модель обработки Pregel

В качестве оптимизации для пакетной обработки графов популярна модель *синхронных параллельных* вычислений (bulk synchronous parallel, BSP) [70]. В частности, она реализована в Apache Giraph [37], Spark GraphX API и Flink Gelly API [71]. Она также известна как модель Pregel, поскольку этот принцип графовой обработки был описан в документе Google Pregel [72].

Напомню, что в MapReduce функции сопоставления концептуально «отправляют сообщение» конкретным функциям сжатия, поскольку система собирает в одной точке все выходные данные сопоставления, имеющие один и тот же ключ. Подобная идея стоит и в основе Pregel: одна вершина может «отправить сообщение» другой, и эти сообщения обычно отправляются по ребрам графа.

На каждой итерации функция вызывается для каждой вершины, передавая ей все адресованные ей сообщения, как при вызове функции сжатия. Отличие от MapReduce заключается в том, что в модели Pregel вершина сохраняет в памяти свое состояние до следующей итерации, поэтому функции приходится обрабатывать только новые входящие сообщения. Если к какой-либо части графа сообщения не поступают, то никакие действия не выполняются.

Это немного похоже на акторную модель (см. пункт «Распределенные акторные фреймворки» подраздела «Поток данных передачи сообщений» раздела 4.2), если рассматривать каждую вершину в качестве актора. Только состояния вершин и сообщений между ними являются отказоустойчивыми и долговечными, а связь осуществляется за фиксированное количество проходов: на каждой итерации структура доставляет все сообщения, которые были отправлены на предыдущей итерации. Акторы обычно не имеют такой гарантии хронометража.

Отказоустойчивость

То, что связь между вершинами возможна только с помощью передачи сообщений (а не прямого запроса друг к другу), помогает повысить производительность выполнения задач Pregel, так как сообщения могут быть собраны в пакет и это позволит сократить время ожидания связи. Остается время ожидания между итерациями: поскольку модель Pregel гарантирует, что все сообщения, отправленные на данной итерации, будут доставлены на следующей, перед началом итерации предыдущая обязана полностью завершиться и все ее сообщения должны быть скопированы по сети.

Даже если используемая сеть отбросит, продублирует или произвольно задержит сообщения (см. раздел 8.2), реализации Pregel гарантируют, что на следующей итерации сообщения будут обрабатываться вершиной только один раз. Как и MapReduce, система прозрачно восстанавливается после сбоев с целью упростить модель программирования для алгоритмов, создаваемых на основе Pregel.

Такая отказоустойчивость достигается путем периодической проверки состояния всех вершин в конце итерации — другими словами, их полное состояние записывается в долговременное хранилище. Если узел выходит из строя и его состояние, записанное в памяти, теряется, то самым простым решением является откат всей обработки графа до последней контрольной точки и повторная обработка. В случае детерминированного алгоритма и регистрации сообщений можно также выборочно восстановить только тот раздел, который был потерян (как для описанных ранее подсистем потока данных) [72].

Параллельная обработка

Вершине не нужно знать, на какой физической машине она выполняется; при отправке сообщений другим вершинам она просто передает их по идентификатору вершины. За секционирование графа отвечает система — например, выбирает, на каком компьютере обрабатывается та или иная вершина и как маршрутизировать сообщения по сети, чтобы они оказались в нужном месте.

Поскольку модель программирования имеет дело только с одной вершиной за раз (иногда это называется «думать как вершина»), система может разбивать граф произвольным образом. В идеале он будет секционирован таким образом, чтобы те

вершины, которые интенсивно обмениваются данными, оказались на одном компьютере. Однако на практике создать такое оптимизированное разделение сложно, чаще система просто разбивает граф по произвольно назначенным идентификаторам вершин, не пытаясь сгруппировать тесно связанные вершины.

В результате графовые алгоритмы часто порождают много межмашинных коммуникационных издержек, а промежуточные состояния (сообщения, отправляемые между узлами сети) часто бывают больше самого графа. Затраты времени на отправку сообщений по сети могут значительно замедлить выполнение распределенных графовых алгоритмов.

По этой причине, если граф помещается в памяти одного компьютера, вполне вероятно, что одномашинный (возможно, однопоточный) алгоритм будет работать быстрее, чем распределенный пакетный процесс [73, 74]. Даже если граф не помещается в памяти, но может поместиться на дисках компьютера, одномашинная обработка с использованием среды разработки, такой как GraphChi, будет хорошим вариантом [75]. Когда же граф слишком велик для установки на одну машину, применение распределенной системы, такой как Pregel, неизбежно; в настоящее время ведутся разработки эффективно распараллеливающих графовых алгоритмов [76].

API и языки высокого уровня

За годы, прошедшие с момента первого появления MapReduce, механизмы распределенной пакетной обработки стали совершеннее. В настоящее время инфраструктура достаточно надежна, чтобы хранить и обрабатывать многие петабайты данных в кластерах, насчитывающих более 10 000 машин. Поскольку задача создать физически работающий пакетный процесс такого масштаба более или менее решена, внимание переключилось на другие области: совершенствование модели программирования, повышение эффективности обработки и расширение набора задач, решаемых с помощью этих технологий.

Как обсуждалось ранее, языки более высокого уровня и API, такие как Hive, Pig, Cascading и Crunch, стали популярными, поскольку программировать задачи вручную в MapReduce — довольно трудоемкий процесс. После появления Tez эти языки высокого уровня получили дополнительное преимущество: возможность перейти к новому механизму обработки потока данных без необходимости переписывать код задачи. В Spark и Flink также имеются собственные потоковые API высокого уровня, которые во многом черпают вдохновение в FlumeJava [34].

Эти потоковые API обычно используют для описания вычислений структурных блоков реляционного стиля: объединение наборов данных по значению некоторого поля; группировку кортежей по ключу; фильтрацию по определенным условиям; агрегирование кортежей путем вычислений, суммирования или других функций. Внутренне представленные операции выполняются с помощью различных алгоритмов объединения и группировки, обсуждавшихся в настоящей главе.

Помимо очевидного преимущества — меньшего количества кода, — эти высокоуровневые интерфейсы тоже допускают интерактивное использование, при котором можно написать в оболочке любой код анализа и зачастую даже выполнить его, чтобы пронаблюдать за его действием. Такой стиль разработки очень полезен при изучении набора данных и проведении экспериментов с разными подходами по его обработке. Это также напоминает философию Unix (см. подраздел «Философия Unix» раздела 10.1).

Эти высокоуровневые интерфейсы повышают эффективность не только работы людей с системой, но и выполнения заданий на уровне машины.

Переход к декларативным языкам запросов

Преимущество объединений в качестве реляционных операторов по сравнению с написанием кода, выполняющего объединение, заключается в том, что система может анализировать свойства входных данных для объединения и автоматически определять, какой из описанных ранее алгоритмов объединения будет наиболее подходящим для текущей задачи. В Hive, Spark и Flink есть оптимизаторы запросов по вычислительным затратам, способные сделать это и даже изменить порядок объединений, чтобы свести к минимуму количество промежуточных состояний [66, 77–79].

Выбор алгоритма объединения способен значительно влиять на скорость выполнения пакетной задачи, и хорошо, если разработчику не приходится помнить и углубляться в тонкости всех алгоритмов объединения, обсуждавшихся в настоящей главе. Это возможно в случае *декларативного* описания объединений: приложение просто объявляет, какие объединения требуются, а оптимизатор запросов решает, как их лучше выполнить. Подобная идея уже встречалась в разделе 2.2.

Однако система MapReduce и ее производные в области потоковой обработки данных очень отличаются от полностью декларативной модели запросов SQL. В основе MapReduce была заложена идея обратного вызова функций: для каждой записи или группы записей вызывается пользовательская функция (сопоставления или сжатия), которая может выполнять произвольный код для генерации выходных данных. Преимущество такого подхода состоит в возможности применения обширной экосистемы существующих библиотек для совершения таких действий, как синтаксический анализ, анализ естественных языков, распознавание изображений, выполнение числовых и статистических алгоритмов.

Свобода выбора кода — то, что давно отличает системы пакетной обработки, происходящие от MapReduce, от баз данных MPP (см. подраздел «Сравнение Hadoop и распределенных баз данных» раздела 10.2). Базы данных тоже имеют возможности для написания пользовательских функций, но часто громоздки в применении и недостаточно хорошо интегрированы с менеджерами пакетов и системами

управления зависимостями, которые широко задействованы в большинстве языков программирования (такими как Maven для Java, npm для JavaScript и Rubygems для Ruby).

Тем не менее разработчики систем обработки данных обнаружили преимущества в том, чтобы использовать декларативные функции не только в объединениях. Например, если функция обратного вызова содержит лишь простое условие фильтрации или всего-навсего выбирает поля из записи, то при ее вызове для каждой записи возникают значительные чрезмерные расходы ресурсов процессора. При декларативном описании подобных простых операций фильтрации и сопоставления оптимизатор запросов может задействовать преимущества столбцового хранения данных (см. раздел 3.3) и прочитать с диска только нужные столбцы. В Hive, Spark DataFrames и Impala также применяется векторизованное выполнение (см. пункт «Пропускная способность памяти и векторизованная обработка» подраздела «Сжатие столбцов» раздела 3.3): перебор данных в жестком внутреннем цикле, построенном с учетом возможностей кэша процессора, без вызова функций. Spark генерирует байт-код JVM [79], а Impala задействует LLVM для генерации собственного кода, выполняющего эти внутренние циклы [41].

Благодаря декларативным аспектам в API высокого уровня и оптимизаторам запросов, способным использовать их во время выполнения, системы пакетной обработки все больше становятся похожи на базы данных MPP (и иногда достигают сравнимой с ними производительности). В то же время они сохраняют гибкость вследствие возможности расширения за счет произвольного кода и чтения данных в произвольных форматах.

Специализация для разных областей

Расширяемость за счет возможности запуска произвольного кода полезна, но есть множество распространенных случаев, когда используются стандартные шаблоны обработки. Так что стоит иметь реализации общих строительных блоков для многократного применения. Традиционно базы данных MPP удовлетворяли потребности бизнес-аналитики и бизнес-отчетности, но это лишь одна из многих областей, в которых задействуется пакетная обработка.

Еще одной областью, приобретающей все большее значение, являются статистические и численные алгоритмы, используемые в приложениях машинного обучения, таких как классификационные системы и системы выдачи рекомендаций. Появляются многоразовые реализации: например, в Mahout созданы алгоритмы машинного обучения на базе MapReduce, Spark и Flink, а в MADlib — аналогичные функции внутри реляционной базы данных MPP (Apache HAWQ) [54].

Кроме того, полезны и пространственные алгоритмы, такие как *ближайшие соседи k* [80]. Они находят элементы, близкие к заданному элементу многомерного

пространства, — своего рода поиск подобия. Приближенный поиск также важен в алгоритмах анализа генома, которым нужно найти строки, похожие на заданную, но не идентичные ей [81].

Системы пакетной обработки применяются для распределенного выполнения алгоритмов из все более широкого круга областей. После того как в таких системах появились встроенные функциональные возможности и высокоуровневые декларативные операторы, а базы данных MPP становятся более программируемыми и гибкими, они начинают все больше походить друг на друга: в конце концов, и те и другие являются системами для хранения и обработки данных.

10.4. Резюме

В настоящей главе мы рассмотрели тему пакетной обработки. Мы начали с изучения инструментов Unix, таких как `awk`, `grep` и `sort`, и увидели, как философия этих инструментов переносится в MapReduce и более современные системы потоковой обработки данных. Часть из описанных принципов проектирования заключается в том, что входные данные неизменяемы, а выходные предназначены служить входными для другой (пока неизвестной) программы; для решения сложных задач создаются небольшие инструменты, которые «хорошо делают что-то одно».

В мире Unix предусмотрен стандартный интерфейс, позволяющий соединять одну программу с другой и основанный на применении файлов и каналов; в MapReduce этот интерфейс является распределенной файловой системой. Мы увидели, что системы потоковой обработки данных добавляют собственные конвейероподобные механизмы передачи данных, чтобы избежать материализации промежуточных состояний в распределенной файловой системе, но исходные и окончательные выходные данные задачи по-прежнему обычно хранятся в HDFS.

Распределенные системы пакетной обработки данных решают две основные проблемы.

- *Секционирование*. В MapReduce функции сопоставления размещаются в секциях в соответствии с входными файловыми блоками. Выходные данные переупорядочиваются, сортируются и объединяются в разделы сжатия, количество которых определяется в зависимости от задачи. Цель этого процесса — разместить все связанные данные, например все записи с одинаковым ключом, в одной секции.

  Системы потоковой обработки данных, созданные на базе MapReduce, стараются избегать сортировки там, где это не требуется, но в остальном широко используют принцип секционирования.
- *Отказоустойчивость*. MapReduce часто записывает данные на диск, что упрощает восстановление после неудачно совершенной операции без повторного выполнения всей задачи, но замедляет выполнение при отсутствии сбоев.

Системы потоковой обработки данных реже используют материализацию промежуточных состояний и держат больше данных в памяти. Это значит, что при выходе узла из строя им приходится выполнять больше повторных вычислений. Детерминированные операторы сокращают объем данных, которые необходимо пересчитывать.

Мы обсудили несколько алгоритмов объединения в MapReduce, большинство из которых используются в базах данных MPP и системах потоковой обработки данных. Они также служат хорошей иллюстрацией того, как работают алгоритмы секционирования.

- *Объединение с сортировкой слияния.* Каждый из наборов входных данных обрабатывается функцией сопоставления, которая извлекает ключ объединения. При секционировании, сортировке и объединении все записи с одинаковым ключом направляются в одну и ту же функцию сжатия. Затем эта функция может выводить объединенные записи.
- *Широковещательное объединение по хешу.* Когда один из двух входных наборов данных мал, он не секционируется и может быть полностью загружен в хеш-таблицу. Это позволяет запускать функцию сопоставления для каждого раздела большого входного набора данных, загружая хеш-таблицу с меньшим набором данных в каждый обработчик, и затем сканировать большой набор, извлекая из него по одной записи и каждый раз запрашивая для нее хеш-таблицу.
- *Секционированные хеш-объединения.* Если два набора входных данных секционированы одинаково (по одному и тому же ключу, одной и той же хеш-функции, с одинаковым количеством разделов), то объединение с помощью хеш-таблицы может применяться независимо для каждого раздела.

Распределенные системы пакетной обработки имеют преднамеренно ограниченную модель программирования: функции обратного вызова (сопоставления и сжатия) считаются не меняющими состояния и не имеющими внешних видимых побочных эффектов, кроме генерируемых ими выходных данных. Это ограничение позволяет системе скрыть некоторые проблемы жестких распределенных систем за абстракцией: в случае сбоев и сетевых проблем задачи можно безопасно повторить, а выходные данные неудачно завершенных задач отбрасываются. Если несколько задач для какой-либо секции выполнены успешно, то только одна из них генерирует фактически видимые выходные данные.

Благодаря такой системе при разработке кода для пакетной обработки не приходится беспокоиться о механизмах отказоустойчивости: система сама гарантирует состояние конечного результата задачи таким же, как если бы никаких ошибок не произошло, хотя в действительности различные операции, возможно, пришлось повторить. Эти надежные семантические структуры намного надежнее, чем те, что обычно применяются в онлайн-сервисах, обрабатывающих пользовательские запросы, и имеют побочные результаты обработки запроса в виде информации, записанной в базу данных.

Характерная особенность задачи пакетной обработки такова: она считывает входные данные и генерирует выходные, не изменяя исходную информацию, — другими словами, выход отделен от входа. Крайне важно, что входные данные *ограничены*: имеют известный фиксированный размер (например, состоят из набора файлов журнала, скопированных в определенный момент времени, или копии содержимого базы данных). Поскольку набор входных данных ограничен, задача знает, когда закончено его чтение, и завершается в этот момент.

В следующей главе мы перейдем к потоковой обработке, в которой входные данные *не ограничены*, то есть у нас все еще есть задача, но на ее входе — бесконечные потоки информации. В этом случае работа никогда не будет завершена, поскольку на ее вход в любой момент могут поступить новые данные. Мы увидим, что потоковая и пакетная обработки несколько похожи, но предположение о неограниченных входных потоках данных многое меняет в способе построения системы.

10.5. Библиография

1. *Dean J., Ghemawat S.* MapReduce: Simplified Data Processing on Large Clusters // 6th USENIX Symposium on Operating System Design and Implementation (OSDI), December 2004 [Электронный ресурс]. — Режим доступа: https://research.google.com/archive/mapreduce.html.
2. *Spolsky J.* The Perils of JavaSchools. December 25, 2005 [Электронный ресурс]. — Режим доступа: https://www.joelonsoftware.com/2005/12/29/the-perils-of-javaschools-2/.
3. *Babu S., Herodotou H.* Massively Parallel Databases and MapReduce Systems // Foundations and Trends in Databases, volume 5, number 1, pages 1–104, November 2013 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/massively-parallel-databases-and-mapreduce-systems/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F206464%2Fdb-mr-survey-final.pdf.
4. *DeWitt D. J., Stonebraker M.* MapReduce: A Major Step Backwards // originally published at databasecolumn.vertica.com, January 17, 2008 [Электронный ресурс]. — Режим доступа: https://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html.
5. *Robinson H.* The Elephant Was a Trojan Horse: On the Death of MapReduce at Google // June 25, 2014 [Электронный ресурс]. — Режим доступа: http://the-paper-trail.org/blog/the-elephant-was-a-trojan-horse-on-the-death-of-map-reduce-at-google/.
6. The Hollerith Machine // United States Census Bureau [Электронный ресурс]. — Режим доступа: https://www.census.gov/history/www/innovations/technology/the_hollerith_tabulator.html.

7. IBM 82, 83, and 84 Sorters Reference Manual // Edition A24-1034-1, International Business Machines Corporation, July 1962 [Электронный ресурс]. — Режим доступа: http://www.textfiles.com/bitsavers/pdf/ibm/punchedCard/Sorter/A24-1034-1_82-83-84_sorters.pdf.
8. *Drake A.* Command-Line Tools Can Be 235x Faster than Your Hadoop Cluster. January 25, 2014 [Электронный ресурс]. — Режим доступа: https://aadrake.com/command-line-tools-can-be-235x-faster-than-your-hadoop-cluster.html.
9. GNU Coreutils 8.23 Documentation // Free Software Foundation, Inc., 2014 [Электронный ресурс]. — Режим доступа: https://www.gnu.org/software/coreutils/manual/html_node/index.html.
10. *Kleppmann M.* Kafka, Samza, and the Unix Philosophy of Distributed Data. August 5, 2015 [Электронный ресурс]. — Режим доступа: http://martin.kleppmann.com/2015/08/05/kafka-samza-unix-philosophy-distributed-data.html.
11. *McIlroy D.* Internal Bell Labs memo. October 1964. Cited in: Dennis M. Richie: Advice from Doug McIlroy [Электронный ресурс]. — Режим доступа: http://cm.bell-labs.com/cm/cs/who/dmr/mdmpipe.pdf.
12. *McIlroy M. D., Pinson E. N., Tague B. A.* UNIX Time-Sharing System: Foreword // The Bell System Technical Journal, volume 57, number 6, pages 1899–1904, July 1978 [Электронный ресурс]. — Режим доступа: https://archive.org/details/bstj57-6-1899.
13. *Raymond E. S.* The Art of UNIX Programming. — Addison-Wesley, 2003. http://www.catb.org/~esr/writings/taoup/html/.
14. *Duncan R.* Text File Formats — ASCII Delimited Text — Not CSV or TAB Delimited Text. October 31, 2009 [Электронный ресурс]. — Режим доступа: https://ronaldduncan.wordpress.com/2009/10/31/text-file-formats-ascii-delimited-text-not-csv-or-tab-delimited-text/.
15. *Kay A.* Is 'Software Engineering' an Oxymoron? [Электронный ресурс]. — Режим доступа: http://tinlizzie.org/~takashi/IsSoftwareEngineeringAnOxymoron.pdf.
16. *Fowler M.* InversionOfControl. June 26, 2005 [Электронный ресурс]. — Режим доступа: https://martinfowler.com/bliki/InversionOfControl.html.
17. *Bernstein D. J.* Two File Descriptors for Sockets [Электронный ресурс]. — Режим доступа: http://cr.yp.to/tcpip/twofd.html.
18. *Pike R., Ritchie D. M.* The Styx Architecture for Distributed Systems // Bell Labs Technical Journal, volume 4, number 2, pages 146–152, April 1999 [Электронный ресурс]. — Режим доступа: http://doc.cat-v.org/inferno/4th_edition/styx.
19. *Ghemawat S., Gobioff H., Leung S.-T.* The Google File System // 19th ACM Symposium on Operating Systems Principles (SOSP), October 2003 [Электронный ресурс]. — Режим доступа: https://static.googleusercontent.com/media/research.google.com/ru//archive/gfs-sosp2003.pdf.

20. *Ovsiannikov M., Rus S., Reeves D., et al.* The Quantcast File System // Proceedings of the VLDB Endowment, volume 6, number 11, pages 1092–1101, August 2013 [Электронный ресурс]. — Режим доступа: http://db.disi.unitn.eu/pages/VLDBProgram/pdf/industry/p808-ovsiannikov.pdf.
21. OpenStack Swift 2.6.1 Developer Documentation // OpenStack Foundation, March 2016 [Электронный ресурс]. — Режим доступа: https://docs.openstack.org/swift/latest/.
22. *Zhang Z., Wang A., Zheng K., et al.* Introduction to HDFS Erasure Coding in Apache Hadoop. September 23, 2015 [Электронный ресурс]. — Режим доступа: http://blog.cloudera.com/blog/2015/09/introduction-to-hdfs-erasure-coding-in-apache-hadoop/.
23. *Cnudde P.* Hadoop Turns 10. February 5, 2016 [Электронный ресурс]. — Режим доступа: http://yahoohadoop.tumblr.com/post/138739227316/hadoop-turns-10.
24. *Baldeschwieler E.* Thinking About the HDFS vs. Other Storage Technologies. July 25, 2012 [Электронный ресурс]. — Режим доступа: https://hortonworks.com/blog/thinking-about-the-hdfs-vs-other-storage-technologies/.
25. *Gregg B.* Manta: Unix Meets Map Reduce. June 25, 2013 [Электронный ресурс]. — Режим доступа: http://dtrace.org/blogs/brendan/2013/06/25/manta-unix-meets-map-reduce/.
26. *White T.* Hadoop: The Definitive Guide, 4th edition. O'Reilly Media, 2015.
27. *Gray J. N.* Distributed Computing Economics // Microsoft Research Tech Report MSR-TR-2003-24, March 2003 [Электронный ресурс]. — Режим доступа: https://arxiv.org/ftp/cs/papers/0403/0403019.pdf.
28. *Trencséni M.* Luigi vs Airflow vs Pinball. February 6, 2016 [Электронный ресурс]. — Режим доступа: http://bytepawn.com/luigi-airflow-pinball.html.
29. *Sumbaly R., Kreps J., Shah S.* The 'Big Data' Ecosystem at LinkedIn // ACM International Conference on Management of Data (SIGMOD), July 2013 [Электронный ресурс]. — Режим доступа: https://www.slideshare.net/s_shah/the-big-data-ecosystem-at-linkedin-23512853.
30. *Gates A. F., Natkovich O., Chopra S., et al.* Building a High-Level Dataflow System on Top of Map-Reduce: The Pig Experience // 35th International Conference on Very Large Data Bases (VLDB), August 2009 [Электронный ресурс]. — Режим доступа: http://www.vldb.org/pvldb/2/vldb09-1074.pdf.
31. *Thusoo A., Sarma J. S., Jain N., et al.* Hive — A Petabyte Scale Data Warehouse Using Hadoop // 26th IEEE International Conference on Data Engineering (ICDE), March 2010 [Электронный ресурс]. — Режим доступа: http://i.stanford.edu/~ragho/hive-icde2010.pdf.
32. Cascading 3.0 User Guide // Concurrent, Inc., January 2016 [Электронный ресурс]. — Режим доступа: http://docs.cascading.org/cascading/3.0/userguide/.
33. Apache Crunch User Guide // Apache Software Foundation [Электронный ресурс]. — Режим доступа: https://crunch.apache.org/user-guide.html.

34. *Chambers C., Raniwala A., Perry F., et al.* FlumeJava: Easy, Efficient Data-Parallel Pipelines // 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2010 [Электронный ресурс]. — Режим доступа: https://static.googleusercontent.com/media/research.google.com/ru//pubs/archive/35650.pdf.

35. *Kreps J.* Why Local State is a Fundamental Primitive in Stream Processing. July 31, 2014 [Электронный ресурс]. — Режим доступа: https://www.oreilly.com/ideas/why-local-state-is-a-fundamental-primitive-in-stream-processing.

36. *Kleppmann M.* Rethinking Caching in Web Apps. October 1, 2012 [Электронный ресурс]. — Режим доступа: http://martin.kleppmann.com/2012/10/01/rethinking-caching-in-web-apps.html.

37. *Grover M., Malaska T., Seidman J., Shapira G.* Hadoop Application Architectures. — O'Reilly Media, 2015. http://shop.oreilly.com/product/0636920033196.do.

38. *Ajoux P., Bronson N., Kumar S., et al.* Challenges to Adopting Stronger Consistency at Scale // 15th USENIX Workshop on Hot Topics in Operating Systems (HotOS), May 2015 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/system/files/conference/hotos15/hotos15-paper-ajoux.pdf.

39. *Manjunath S.* Skewed Join. 2009 [Электронный ресурс]. — Режим доступа: https://wiki.apache.org/pig/PigSkewedJoinSpec.

40. *DeWitt D. J., Naughton J. F., Schneider D. A., Seshadri S.* Practical Skew Handling in Parallel Joins // 18th International Conference on Very Large Data Bases (VLDB), August 1992 [Электронный ресурс]. — Режим доступа: http://www.vldb.org/conf/1992/P027.PDF.

41. *Kornacker M., Behm A., Bittorf V., et al.* Impala: A Modern, Open-Source SQL Engine for Hadoop // 7th Biennial Conference on Innovative Data Systems Research (CIDR), January 2015 [Электронный ресурс]. — Режим доступа: http://pandis.net/resources/cidr15impala.pdf.

42. *Monsch M.* Open-Sourcing PalDB, a Lightweight Companion for Storing Side Data. October 26, 2015 [Электронный ресурс]. — Режим доступа: https://engineering.linkedin.com/blog/2015/10/open-sourcing-paldb--a-lightweight-companion-for-storing-side-da.

43. *Peng D., Dabek F.* Large-Scale Incremental Processing Using Distributed Transactions and Notifications // 9th USENIX conference on Operating Systems Design and Implementation (OSDI), October 2010 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Peng.pdf.

44. Cloudera Search User Guide // Cloudera, Inc., September 2015 [Электронный ресурс]. — Режим доступа: http://www.cloudera.com/documentation/cdh/5-1-x/Search/Cloudera-Search-User-Guide/Cloudera-Search-User-Guide.html.

45. *Wu L., Shah S., Choi S., et al.* The Browsemaps: Collaborative Filtering at LinkedIn // 6th Workshop on Recommender Systems and the Social Web (RSWeb), October 2014

[Электронный ресурс]. — Режим доступа: http://ls13-www.cs.tu-dortmund.de/homepage/rsweb2014/papers/rsweb2014_submission_3.pdf.

46. *Sumbaly R., Kreps J., Gao L., et al.* Serving Large-Scale Batch Computed Data with Project Voldemort // 10th USENIX Conference on File and Storage Technologies (FAST), February 2012 [Электронный ресурс]. — Режим доступа: http://static.usenix.org/events/fast12/tech/full_papers/Sumbaly.pdf.
47. *Sharma V.* Open-Sourcing Terrapin: A Serving System for Batch Generated Data. September 14, 2015 [Электронный ресурс]. — Режим доступа: https://medium.com/@Pinterest_Engineering/open-sourcing-terrapin-a-serving-system-for-batch-generated-data-7aa2f38c4472.
48. *Marz N.* ElephantDB. May 30, 2011 [Электронный ресурс]. — Режим доступа: https://www.slideshare.net/nathanmarz/elephantdb.
49. *Cryans J.-D. (JD)* How-to: Use HBase Bulk Loading, and Why. September 27, 2013 [Электронный ресурс]. — Режим доступа: http://blog.cloudera.com/blog/2013/09/how-to-use-hbase-bulk-loading-and-why/.
50. *Marz N.* How to Beat the CAP Theorem. October 13, 2011 [Электронный ресурс]. — Режим доступа: http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.
51. *Dishman M. B., Fowler M.* Agile Architecture // O'Reilly Software Architecture Conference, March 2015 [Электронный ресурс]. — Режим доступа: https://conferences.oreilly.com/software-architecture/sa2015/public/schedule/detail/40388.
52. *DeWitt D. J., Gray J. N.* Parallel Database Systems: The Future of High Performance Database Systems // Communications of the ACM, volume 35, number 6, pages 85–98, June 1992 [Электронный ресурс]. — Режим доступа: http://www.cs.cmu.edu/~pavlo/courses/fall2013/static/papers/dewittgray92.pdf.
53. *Kreps J.* But the multi-tenancy thing is actually really really hard // tweetstorm, October 31, 2014 [Электронный ресурс]. — Режим доступа: https://twitter.com/jaykreps/status/528235702480142336.
54. *Cohen J., Dolan B., Dunlap M., et al.* MAD Skills: New Analysis Practices for Big Data // Proceedings of the VLDB Endowment, volume 2, number 2, pages 1481–1492, August 2009 [Электронный ресурс]. — Режим доступа: http://www.vldb.org/pvldb/2/vldb09-219.pdf.
55. *Terrizzano I., Schwarz P., Roth M., Colino J. E.* Data Wrangling: The Challenging Journey from the Wild to the Lake // 7th Biennial Conference on Innovative Data Systems Research (CIDR), January 2015 [Электронный ресурс]. — Режим доступа: http://cidrdb.org/cidr2015/Papers/CIDR15_Paper2.pdf.
56. *Roberts P.* To Schema on Read or to Schema on Write, That Is the Hadoop Data Lake Question. July 2, 2015 [Электронный ресурс]. — Режим доступа: http://adaptivesystemsinc.com/blog/to-schema-on-read-or-to-schema-on-write-that-is-the-hadoop-data-lake-question/.
57. *Johnson B., Adler J.* The Sushi Principle: Raw Data Is Better // Strata+Hadoop World, February 2015 [Электронный ресурс]. — Режим доступа: https://vimeo.com/123985284.

58. *Vavilapalli V. K., Murthy A. C., Douglas C., et al.* Apache Hadoop YARN: Yet Another Resource Negotiator // 4th ACM Symposium on Cloud Computing (SoCC), October 2013 [Электронный ресурс]. — Режим доступа: https://54e57bc8-a-62cb3a1a-s-sites.googlegroups.com/site/2013socc/home/program/a5-vavilapalli.pdf?attachauth=ANoY7com4l4oY_8Yh2px8uFDflJUQgIciVWZvfsnJP4jFKRE7CiEW67HmTpvMi6Ipg6_pxMJmhcTULe2SnBME5gTBoti0cp8URSN8HszXnesenCAh5waTwnITu9_erS67RTW0uELvx09EknRY3qr63ah90c42xfNdb7ex8pxI6WqUbWzX1FZnVf0FTo9nJOwZQaf-5bIisWN_o05bQcHVIVPq9NEkiqExSpA9tulTz70zhPnO9VECgg%3D&attredirects=0.

59. *Verma A., Pedrosa L., Korupolu M., et al.* Large-Scale Cluster Management at Google with Borg // 10th European Conference on Computer Systems (EuroSys), April 2015 [Электронный ресурс]. — Режим доступа: https://research.google.com/pubs/pub43438.html.

60. *Schwarzkopf M.* The Evolution of Cluster Scheduler Architectures. March 9, 2016 [Электронный ресурс]. — Режим доступа: http://www.firmament.io/blog/scheduler-architectures.html.

61. *Zaharia M., Chowdhury M., Das T., et al.* Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing // 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI), April 2012 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf.

62. *Karau H., Konwinski A., Wendell P., Zaharia M.* Learning Spark. O'Reilly Media, 2015.

63. *Saha B., Shah H.* Apache Tez: Accelerating Hadoop Query Processing // Hadoop Summit, June 2014 [Электронный ресурс]. — Режим доступа: https://www.slideshare.net/Hadoop_Summit/w-1205phall1saha.

64. *Saha B., Shah H., Seth S., et al.* Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications // ACM International Conference on Management of Data (SIGMOD), June 2015 [Электронный ресурс]. — Режим доступа: http://home.cse.ust.hk/~weiwa/teaching/Fall15-COMP6611B/reading_list/Tez.pdf.

65. *Tzoumas K.* Apache Flink: API, Runtime, and Project Roadmap. January 14, 2015 [Электронный ресурс]. — Режим доступа: https://www.slideshare.net/KostasTzoumas/apache-flink-api-runtime-and-project-roadmap.

66. *Alexandrov A., Bergmann R., Ewen S., et al.* The Stratosphere Platform for Big Data Analytics // The VLDB Journal, volume 23, number 6, pages 939–964, May 2014 [Электронный ресурс]. — Режим доступа: https://ssc.io/pdf/2014-VLDBJ_Stratosphere_Overview.pdf.

67. *Isard M., Budiu M., Yu Y., et al.* Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks // European Conference on Computer Systems (EuroSys), March 2007 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/project/dryad/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fprojects%2Fdryad%2Feurosys07.pdf.

68. *Warneke D., Kao O.* Nephele: Efficient Parallel Data Processing in the Cloud // 2nd Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS), November 2009 [Электронный ресурс]. — Режим доступа: https://stratosphere2.dima.tu-berlin.de/assets/papers/Nephele_09.pdf.

69. *Page L., Brin S., Motwani R., Winograd T.* The PageRank Citation Ranking: Bringing Order to the Web // Stanford InfoLab Technical Report 422, 1999 [Электронный ресурс]. — Режим доступа: http://ilpubs.stanford.edu:8090/422/.

70. *Valiant L. G.* A Bridging Model for Parallel Computation // Communications of the ACM, volume 33, number 8, pages 103–111, August 1990 [Электронный ресурс]. — Режим доступа: https://dl.acm.org/citation.cfm?id=79181.

71. *Ewen S., Tzoumas K., Kaufmann M., Markl V.* Spinning Fast Iterative Data Flows // Proceedings of the VLDB Endowment, volume 5, number 11, pages 1268–1279, July 2012 [Электронный ресурс]. — Режим доступа: http://vldb.org/pvldb/vol5/p1268_stephanewen_vldb2012.pdf.

72. *Malewicz G., Austern M. H., Bik A. J. C., et al.* Pregel: A System for Large-Scale Graph Processing // at ACM International Conference on Management of Data (SIGMOD), June 2010 [Электронный ресурс]. — Режим доступа: https://kowshik.github.io/JPregel/pregel_paper.pdf.

73. *McSherry F., Isard M., Murray D. G.* Scalability! But at What COST? // 15th USENIX Workshop on Hot Topics in Operating Systems (HotOS), May 2015 [Электронный ресурс]. — Режим доступа: http://www.frankmcsherry.org/assets/COST.pdf.

74. *Gog I., Schwarzkopf M., Crooks N., et al.* Musketeer: All for One, One for All in Data Processing Systems // 10th European Conference on Computer Systems (EuroSys), April 2015 [Электронный ресурс]. — Режим доступа: http://www.cl.cam.ac.uk/research/srg/netos/camsas/pubs/eurosys15-musketeer.pdf.

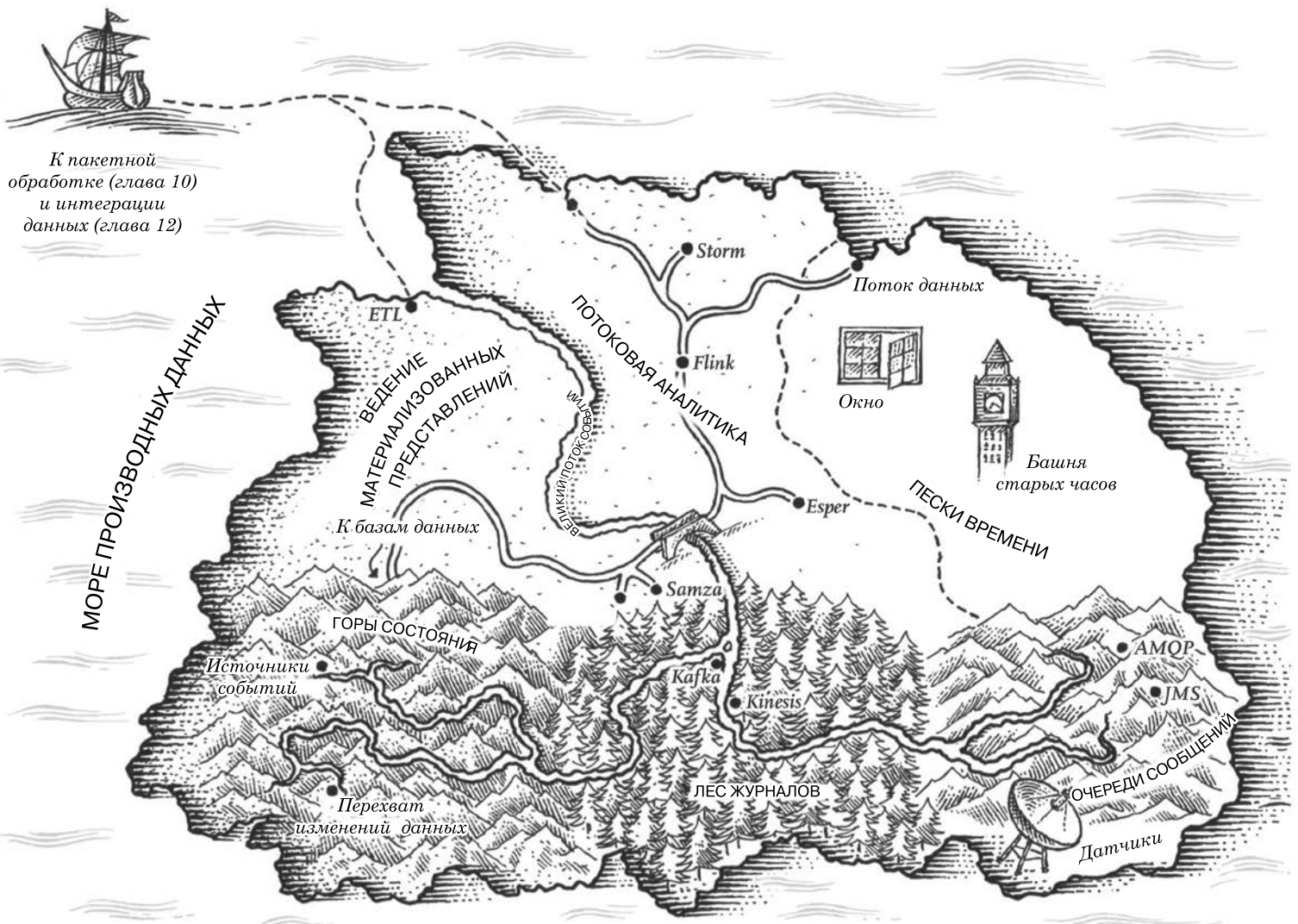
75. *Kyrola A., Blelloch G., Guestrin C.* GraphChi: Large-Scale Graph Computation on Just a PC // 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI), October 2012 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/system/files/conference/osdi12/osdi12-final-126.pdf.

76. *Lenharth A., Nguyen D., Pingali K.* Parallel Graph Analytics // Communications of the ACM, volume 59, number 5, pages 78–87, May 2016 [Электронный ресурс]. — Режим доступа: https://cacm.acm.org/magazines/2016/5/201591-parallel-graph-analytics/abstract.

77. *Hüske F.* Peeking into Apache Flink's Engine Room. March 13, 2015 [Электронный ресурс]. — Режим доступа: http://flink.apache.org/news/2015/03/13/peeking-into-Apache-Flinks-Engine-Room.html.

78. *Mokhtar M.* Hive 0.14 Cost Based Optimizer (CBO) Technical Overview. March 2, 2015 [Электронный ресурс]. — Режим доступа: https://hortonworks.com/blog/hive-0-14-cost-based-optimizer-cbo-technical-overview/.

79. *Armbrust M., Xin R. S., Cheng L., et al.* Spark SQL: Relational Data Processing in Spark // ACM International Conference on Management of Data (SIGMOD), June 2015 [Электронный ресурс]. — Режим доступа: http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf.
80. *Blazevski D.* Planting Quadtrees for Apache Flink. March 25, 2016 [Электронный ресурс]. — Режим доступа: https://blog.insightdatascience.com/planting-quadtrees-for-apache-flink-b396ebc80d35.
81. *White T.* Genome Analysis Toolkit: Now Using Apache Spark for Data Processing. April 6, 2016 [Электронный ресурс]. — Режим доступа: http://blog.cloudera.com/blog/2016/04/genome-analysis-toolkit-now-using-apache-spark-for-data-processing/.


К пакетной обработке (глава 10) и интеграции данных (глава 12)
МОРЕ ПРОИЗВОДНЫХ ДАННЫХ
ETL
ВЕДЕНИЕ МАТЕРИАЛИЗОВАННЫХ ПРЕДСТАВЛЕНИЙ
Storm
Flink
ПОТОКОВАЯ АНАЛИТИКА
ВЕЛИКИЙ ПОТОК СОБЫТИЙ
Поток данных
Окно
Башня старых часов
ПЕСКИ ВРЕМЕНИ
Esper
К базам данных
Samza
ГОРЫ СОСТОЯНИЯ
Источники событий
Kafka
Kinesis
AMQP
JMS
ОЧЕРЕДИ СООБЩЕНИЙ
ЛЕС ЖУРНАЛОВ
Перехват изменений данных
Датчики

11 Потоковая обработка

> Любая работающая сложная система неизбежно выросла из работающей простой системы. Верно и обратное: сложная система, разработанная с нуля, никогда не будет работать, и ее невозможно привести в рабочее состояние.
>
> *Джон Голл. Systemantics (1975)*

В главе 10 обсуждалась пакетная обработка — технологии, при которых считывался набор файлов в качестве входных данных и затем создавался новый набор в качестве выходных данных. Выходная информация является формой *производных данных* — набора, который при необходимости можно воссоздать, повторно запустив пакетный процесс. Мы увидели, как эта простая, но эффективная идея может быть использована для создания поисковых индексов, систем выдачи рекомендаций, аналитики и т. п.

Однако в главе 10 было сделано серьезное допущение: входные данные ограничены, например, их размер известен и конечен. Поэтому пакетный процесс знает, когда чтение таких данных завершено. Например, при операции сортировки, которая является центральной для MapReduce, сначала считываются все входные данные и только потом начинается генерирование выходной информации. Ведь может случиться так, что у самой последней входной записи окажется самый маленький ключ и, следовательно, она станет самой первой выходной записью, поэтому начинать вывод раньше, чем будет прочитан весь поток ввода, не имеет смысла.

На практике поток данных часто не ограничен, информация приходит постепенно, с течением времени. Пользователи генерируют информацию сегодня, как и вчера, и продолжат это делать завтра. Процесс никогда не закончится, если только вы

не решили закрыть бизнес. Поэтому, строго говоря, набор данных никогда не будет «окончательным» [1]. Пакетным процессорам приходится искусственно делить информацию на фиксированные временные промежутки: например, в конце каждого дня обрабатывать данные, полученные за этот день, или в конце часа — данные, поступившие за час.

Проблема с ежедневными пакетными процессами заключается в том, что изменения входной информации отражаются на выходе только через день — для многих нетерпеливых пользователей это слишком долго. Для уменьшения задержки можно чаще повторять обработку: например, обрабатывать данные раз в секунду или даже непрерывно, полностью отказавшись от разделения времени на фиксированные интервалы. Просто обрабатывать события по мере их возникновения. Это и есть идея *потоковой обработки*.

Вообще говоря, потоком называют данные, которые становятся доступными постепенно, с течением времени. Эта концепция находит самое разное применение: в `stdin` и `stdout` Unix, в языках программирования («ленивые списки») [2], в API файловых систем (таких как Java `FileInputStream`), в TCP-соединениях, технологиях доставки аудио- и видеосодержимого через Интернет и т. п.

В этой главе мы рассмотрим *потоки событий* как механизм управления данными: аналог рассмотренных в предыдущей главе пакетных данных, но только без ограничений и с возможностью поэтапной обработки. Сначала мы изучим представление, хранение и передачу потоков по сети. В разделе 11.2 рассмотрим взаимосвязь между потоками и базами данных. И наконец, в разделе 11.3 познакомимся с методами и инструментами для непрерывной обработки потоков, а также способами их использования для создания приложений.

11.1. Передача потоков событий

В мире пакетной обработки входные и выходные данные задачи являются файлами (возможно, в распределенной файловой системе). Как выглядит их потоковый эквивалент?

Когда входные данные представляют собой файл (последовательность байтов), первый шаг обработки обычно заключается в том, чтобы проанализировать его и превратить в последовательность записей. В контексте обработки потока записи обычно называют *событиями*, но, в сущности, это одно и то же: небольшой, автономный, неизменный объект, хранящий детали произошедшего в какой-то момент времени. Обычно событие содержит временную метку, показывающую, в какое время суток оно произошло (см. подраздел «Монотонные часы и часы истинного времени» раздела 8.3).

Произошедшее событие может быть действием пользователя, например просмотром страницы или сделанной покупкой. Оно может также генерироваться компьютером, например быть результатом периодического измерения от датчика температуры

или показателем нагрузки на процессор. В примере, рассмотренном в разделе 10.1, каждая строка журнала веб-сервера является событием.

Событие может быть представлено в виде текстовой строки, или кода JSON, или, вероятно, в некой двоичной форме, как было показано в главе 4. Такое представление позволяет сохранять события, например записывая их в файл, вставляя в реляционную таблицу или занося в документную базу данных. Оно также позволяет передавать событие по сети на другой узел для последующей обработки.

При пакетной обработке файл записывается один раз и затем может считываться многократно различными задачами. В терминологии потоков аналогичная ситуация происходит с событием: оно генерируется один раз *инициатором* (также известным как *издатель* или *отправитель*), а затем может обрабатываться разными *потребителями* (*подписчиками* или *получателями*) [3]. В файловой системе имя файла идентифицирует набор взаимосвязанных записей; в потоковой системе связанные события обычно объединяются в *тему* или *поток*.

В принципе, чтобы установить связь между инициаторами и потребителями, достаточно файла или БД: инициатор записывает в хранилище данных каждое сгенерированное им событие, а потребители периодически опрашивают хранилище на предмет того, не появились ли там новые события с момента последнего опроса. Это, по сути, тот же пакетный процесс, обрабатывающий полученные за день данные в конце каждого дня.

Однако при переходе к непрерывной обработке с короткими задержками опрос становится слишком затратным, если хранилище данных не предназначено для такого использования. Чем чаще проводится опрос, тем ниже процент запросов, возвращающих новые события и, следовательно, выше накладные расходы. Вместо этого лучше отправлять потребителям уведомления о появлении новых событий.

Традиционные базы данных не предназначены для поддержки такого механизма уведомлений: в реляционных БД обычно есть *триггеры*, способные реагировать на изменения (например, вставку строки в таблицу), но их возможности очень ограниченны. Можно сказать, что в архитектуре баз триггеры были реализованы задним числом [4, 5]. Вместо этого для доставки уведомлений о событиях были разработаны специализированные инструменты.

Системы обмена сообщениями

Общий принцип уведомления потребителей о новых событиях заключается в использовании *системы обмена сообщениями*: инициатор отправляет сообщение, содержащее событие, которое затем рассылается потребителям. Мы уже упоминали о таких системах в подразделе «Поток данных передачи сообщений» раздела 4.2, но теперь рассмотрим их более подробно.

Простейшая реализация системы обмена сообщениями — прямой канал связи между инициатором и потребителем, такой как конвейер Unix или TCP-соединение.

Однако большинство систем обмена сообщениями идут дальше этой базовой модели. В частности, конвейеры Unix и TCP-соединения способны соединить только одного отправителя с одним получателем, тогда как система обмена сообщениями позволяет нескольким узлам-инициаторам отправлять сообщения в одну и ту же тему, и нескольким потребительским узлам — получать их из данной темы.

В рамках такой модели *публикации/подписки* различные системы используют широкий спектр подходов, единой технологии на все случаи жизни не существует. Для классификации этих систем очень полезно задать себе следующие два вопроса.

1. *Что произойдет, если инициаторы будут отправлять сообщения быстрее, чем потребители смогут их обрабатывать?* Вообще говоря, существует три варианта: система будет отбрасывать сообщения, буферизовать их в виде очереди или использовать *контроль обратного потока* (также известный как *управление потоком*, то есть запрет инициатору отправлять новые сообщения). Например, в конвейерах Unix и TCP-соединениях применяется этот контроль: в них есть небольшой буфер фиксированного размера, если он заполняется, то отправитель блокируется до тех пор, пока получатель не выберет данные из буфера (см. пункт «Перегруженность сети и очереди» подраздела «Время ожидания и неограниченные задержки» раздела 8.2).

   При буферизации сообщений в виде очереди важно понять происходящее в случае роста очереди. Что случится, если она выйдет за пределы памяти: сбой системы или запись сообщений на диск? И как потом доступ к диску повлияет на скорость обмена сообщениями [6]?
2. *Что произойдет, если узел сети даст сбой или будет временно недоступен? Будут ли сообщения потеряны?* Как и в случае с базами данных, ради надежности может быть реализована некая комбинация записи на диск и/или репликации (см. врезку «Репликация и сохраняемость» в пункте «Сохраняемость» подраздела «Смысл аббревиатуры ACID» раздела 7.1), которая потребует определенных затрат. При допустимой иногда потере сообщений, вероятно, удастся получить более высокую пропускную способность и более низкую задержку на одном и том же оборудовании.

Будет ли приемлема потеря сообщений — очень зависит от приложения. Например, в случае с показаниями датчиков и метриками, которые передаются периодически, одно случайно потерянное измерение, возможно, не так важно, поскольку вскоре будет передано обновленное значение. Однако будьте осторожны: при потере сразу многих сообщений текущие показания окажутся неверны и это обнаружится не сразу [7]. Если же события подсчитываются, то их надежная доставка более важна, поскольку каждое потерянное сообщение означает неправильные показания счетчиков.

Хорошим свойством систем пакетной обработки, рассмотренных в главе 10, является то, что они гарантируют надежность: неудачно завершенные задачи автоматически повторяются, а их частичные результаты автоматически отбрасываются. То есть в случае неудачи выходные данные будут такими же, как и при отсутствии сбоев; это упрощает модель программирования. Позже мы рассмотрим, как получить аналогичные гарантии для потоковой обработки.

Прямой обмен сообщениями между инициаторами и потребителями

В некоторых системах обмена сообщениями используется прямая связь по сети между инициаторами и потребителями, без узлов-посредников.

- В финансовой сфере для таких потоков, как каналы фондового рынка, где важна низкая латентность, широко применяется групповая рассылка UDP [8]. Хотя сам по себе протокол UDP ненадежен, протоколы на уровне приложений способны восстанавливать потерянные пакеты (инициатор должен запоминать пакеты, которые отправил, чтобы иметь возможность передавать их повторно по требованию).
- В библиотеках обмена сообщениями без посредников, таких как ZeroMQ [9] и nanomsg, использован аналогичный подход. В них публикация и подписка на сообщения осуществляются по протоколу TCP или IP multicast.
- В StatsD [10] и Brubeck [7] задействован ненадежный обмен сообщениями по протоколу UDP для сбора и мониторинга показателей датчиков со всех компьютеров сети. (В протоколе StatsD показания счетчика признаются достоверными только в том случае, если все сообщения получены; использование UDP делает все показания в лучшем случае приблизительными [11]. См. также врезку «TCP и UDP» в пункте «Перегруженность сети и очереди» подраздела «Время ожидания и неограниченные задержки» раздела 8.2.)
- Если пользователь предоставляет сервис по сети, то инициатор, чтобы отправить сообщение потребителю, может сделать прямой запрос по протоколу HTTP или RPC (см. подраздел «Поток данных через сервисы: REST и RPC» раздела 4.2). Эта идея лежит в основе webhooks [12] — паттерна, в котором URL обратного вызова сервиса регистрируется другим сервисом и делает запрос на этот адрес всякий раз, когда происходит событие.

Эти системы прямого обмена сообщениями хорошо работают в ситуациях, для которых они разработаны, но обычно требуют, чтобы в коде приложения учитывалась возможность потери сообщения. Круг проблем, с которыми они могут справляться сами, весьма ограничен: даже если протоколы обнаруживают и повторно передают потерянные в сети пакеты, обычно предполагается, что инициаторы и потребители постоянно подключены к сети.

Если потребитель отключится, то может пропустить сообщения, отправленные, пока он был вне доступа. Отдельные протоколы позволяют инициатору повторить сообщения в случае неудачной доставки, но в случае сбоя последнего этот подход может не сработать, так как буфер сообщений, которые должны были быть переданы повторно, будет потерян.

Брокеры сообщений

Вместо прямого обмена сообщениями часто используется отправка через *брокеров сообщений* (также известная как *очередь сообщений*). По существу, это своего рода база данных, оптимизированная для обработки потоков сообщений [13]. Он работает как сервер, а инициаторы и потребители подключаются к нему как клиенты. Инициаторы пишут сообщения брокеру, а потребители получают сообщения, читая их у него.

Благодаря централизации данных у брокеров такие системы более легко справляются с ситуациями, когда клиенты появляются и исчезают (подключаются, разъединяются, терпят сбои), а задача обеспечения надежности возлагается на брокера. Одни брокеры сохраняют сообщения только в памяти, другие (в зависимости от конфигурации) записывают их на диск, чтобы не потерять в случае сбоя. Столкнувшись с медленным потребителем, они позволяют обеспечить неограниченную очередь (вместо сброса сообщений или контроля обратного потока, хотя такие варианты тоже возможны при соответствующей настройке).

Результатом очередей также является то, что потребители обычно *асинхронны*: инициатор, отправляя сообщение, ждет только подтверждения от брокера о буферизации этого сообщения. Он не ждет, пока сообщение будет обработано потребителями. Доставка им произойдет в какой-то неопределенный будущий момент — часто за долю секунды, но иногда значительно позже, в случае большой очереди.

Что лучше: брокер сообщений или база данных?

Некоторые брокеры сообщений могут даже участвовать в двухфазных протоколах фиксации транзакций с использованием XA или JTA (см. подраздел «Распределенные транзакции на практике» раздела 9.4). Эта особенность делает их подобными по своей природе базам данных, хотя между ними все равно остаются следующие важные практические различия.

- Информация в БД обычно хранится до тех пор, пока не будет явно удалена, тогда как большинство брокеров автоматически удаляют сообщение после того, как оно было успешно доставлено потребителям. Такие брокеры сообщений не подходят для долговременного хранения данных.
- Поскольку брокеры быстро удаляют сообщения, большинство из них рассчитано на относительно небольшой рабочий набор данных — другими словами,

предполагается, что очереди сравнительно коротки. Если из-за медленных потребителей брокеру приходится буферизовать много сообщений (возможно, сохраняя сообщения на диске ввиду того, что они больше не помещаются в памяти), то на обработку каждого сообщения уходит больше времени, а общая пропускная способность может снизиться [6].

- Базы обычно поддерживают вторичные индексы и различные способы поиска данных, в то время как брокеры сообщений обеспечивают какой-либо способ подписки на подмножество тем, соответствующих определенному шаблону. Несмотря на то что это разные механизмы, оба, по существу, являются способами, позволяющими клиенту выбрать интересующую его часть данных.
- Результат запроса в БД обычно основан на ее текущем состоянии в данный момент времени; если другой клиент впоследствии что-то запишет в базу и это изменит результат запроса, то первый клиент не узнает об устаревании его предыдущего результата (при условии, что не повторяет запрос или не будет опрашивать базу об изменениях). Брокеры сообщений, напротив, не поддерживают произвольные запросы, но уведомляют клиентов об изменении данных (то есть о появлении новых сообщений).

Это традиционное представление о брокерах сообщений реализовано в таких стандартах, как JMS [14] и AMQP [15], и в таких программах, как RabbitMQ, ActiveMQ, HornetQ, Qpid, TIBCO Enterprise Message Service, IBM MQ, Azure Service Bus и Google Cloud Pub/Sub [16].

Когда потребителей несколько

Когда несколько потребителей читают сообщения в одной теме, используются два основных паттерна обмена сообщениями.

- *Распределение нагрузки* (рис. 11.1, *а*). Каждое сообщение доставляется *одному* потребителю, вследствие чего потребители могут распределить между собой обработку сообщений в теме. Брокер может доставлять сообщения потребителям произвольно. Данный паттерн полезен, когда обработка сообщений требует значительных ресурсов, и поэтому желательно распараллелить обработку между потребителями. (В AMQP распределение нагрузки среди нескольких клиентов реализуется за счет того, что они обращаются к общей очереди, а в JMS это называется *общей подпиской*.)
- *Разветвление* (см. рис. 11.1, *б*). Каждое сообщение доставляется *всем* потребителям. Разветвление позволяет нескольким потребителям независимо друг от друга «настраиваться» на одну и ту же рассылку сообщений — потоковый эквивалент нескольких пакетных задач, читающих общий входной файл. (В JMS эта технология реализована в виде подписки на темы, а в AMQP — в виде обмена потоков.)

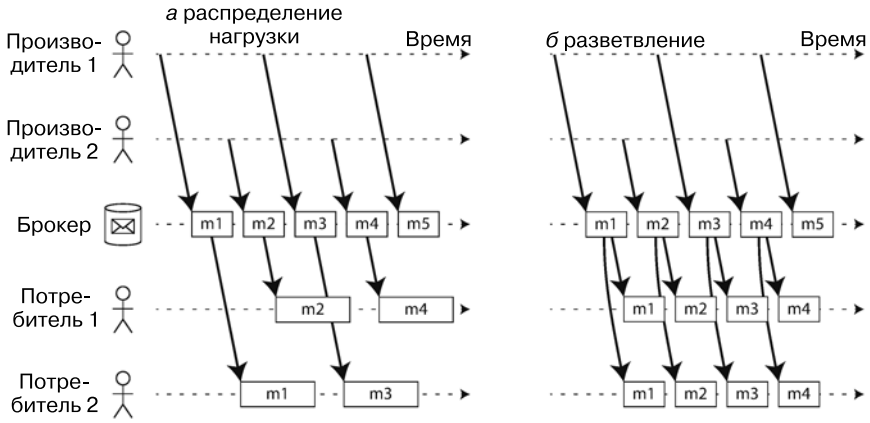


Рис. 11.1. Паттерны обмена сообщениями: *а* — распределение нагрузки: совместная обработка темы потребителями; *б* — разветвление: каждое сообщение передается нескольким потребителям

Эти два шаблона можно сочетать: например, две группы потребителей могут подписаться на тему, так что каждая станет принимать все сообщения, но внутри групп каждое сообщение будет поступать только на один из узлов.

Подтверждение и повторная доставка

В любой момент у потребителя может случиться сбой. Вероятна ситуация, когда брокер доставил сообщение, но потребитель его так и не обработал или обработал лишь частично перед сбоем. Чтобы сообщение не было потеряно, брокеры используют *подтверждения*: клиент должен явно сообщить брокеру о завершении обработки сообщения, и тогда брокер удалит сообщение из очереди.

Если соединение с клиентом было закрыто или время ожидания истекло, а брокер так и не получил подтверждение, то он предполагает, что сообщение не было обработано, и поэтому передает сообщение другому пользователю. (Обратите внимание: в действительности сообщение *может быть* полностью обработано, но подтверждение потерялось в сети. Для обработки данного случая требуется протокол подтверждения малых изменений, описанный в подразделе «Распределенные транзакции на практике» раздела 9.4.)

В сочетании с распределением нагрузки такая технология повторной доставки интересно влияет на последовательность сообщений. На рис. 11.2 потребители обычно обрабатывают сообщения в порядке отправки инициаторами. Однако предположим, что у потребителя 2 во время обработки сообщения m3 происходит сбой, в то время как потребитель 1 обрабатывает сообщение m4. Тогда неподтвержденное сообщение m3 повторно отправляется потребителю 1, в результате

чего тот обрабатывает сообщения в порядке m4, m3, m5. Таким образом, сообщения m3 и m4 доставляются не в том порядке, в каком они были отправлены инициатором 1.

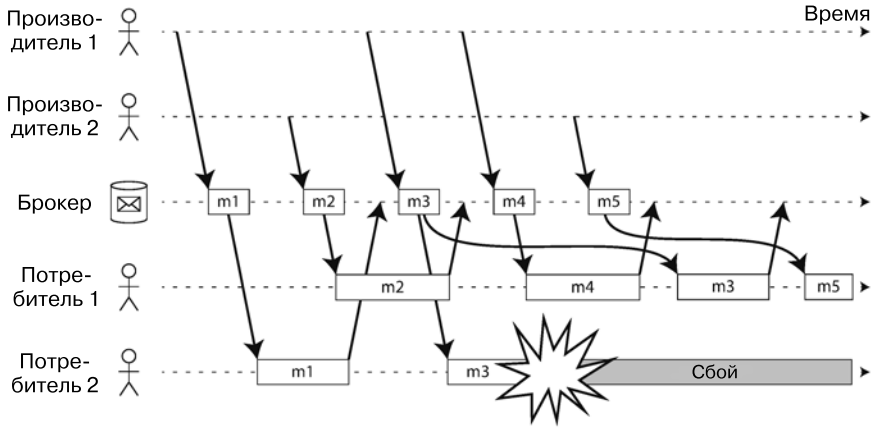


Рис. 11.2. У потребителя 2 во время обработки сообщения m3 произошел сбой, поэтому сообщение повторно отправляется потребителю 1 и обрабатывается позже, чем следующие за ним сообщения

Даже если брокер сообщений пытается сохранить их последовательность (как того требуют стандарты JMS и AMQP), то сочетание распределения нагрузки с повторной доставкой неизбежно приводит к нарушению порядка сообщений. Чтобы избежать этой проблемы, можно использовать отдельную очередь для каждого потребителя (то есть не прибегать к распределению нагрузки). Если сообщения независимы друг от друга, то нарушение их последовательности не является проблемой, но при наличии между сообщениями причинно-следственных связей их порядок может быть важен, как станет ясно далее в настоящей главе.

Секционирование журналов

Отправка пакета по сети или запрос сетевого сервиса — обычно кратковременная операция, не оставляющая за собой каких-либо длительных следов. Их можно сохранять в постоянной памяти (методом захвата и регистрации пакетов), но обычно этого не делают. Даже брокеры сообщений, записывающие сообщения на диск, удаляют их сразу после доставки потребителям, поскольку сообщения по своей природе — кратковременные объекты.

В основе баз данных и файловых систем лежит противоположный подход: предполагается, что все записанное в БД или файл внесено туда навсегда, по крайней мере пока кто-нибудь явно его не удалит.

Указанное различие имеет большое влияние на то, как создаются производные данные. Как обсуждалось в главе 10, ключевой особенностью пакетных процессов является возможность запускать их многократно, экспериментируя с этапами обработки и не рискуя при этом повредить входные данные (поскольку они доступны только для чтения). При обмене сообщениями AMQP/JMS такой подход не работает: после подтверждения брокер удаляет доставленное сообщение, вследствие чего нельзя снова запустить программу-потребитель и получить тот же результат.

Если добавить в систему обмена сообщениями нового потребителя, то он обычно начнет получать сообщения, отправленные после того, как был зарегистрирован. Все предыдущие сообщения уже исчезли и не могут быть восстановлены — в отличие от файловых систем и баз, в которых все новые клиенты могут читать все данные, как бы давно они ни были записаны (до тех пор пока приложение их явно не перезапишет или не удалит).

Почему бы не создать гибрид, сочетающий надежный подход хранения данных, свойственный базам, и оповещения с малой задержкой, свойственные системам обмена сообщениями? Именно эта идея лежит в основе *брокеров сообщений на базе журналирования*.

Использование журналов для хранения сообщений

Журнал — это просто хранящаяся на диске последовательность записей с разрешением только на добавление записей. В главе 3 уже обсуждались журналы в контексте журнализированных подсистем хранения данных и журналов упреждающей записи, а в главе 5 было рассмотрено применение журналов при репликации.

Такая же структура может быть использована и для реализации брокера сообщений: инициатор отправляет сообщение, добавляя в конец журнала, а потребитель получает его, последовательно читая журнал. Если потребитель достигает конца журнала, то ждет уведомления о добавлении нового сообщения. В сущности, именно так работает инструмент Unix `tail -f`, отслеживающий файлы, в которые добавляются данные.

Чтобы обеспечить журналу более высокую пропускную способность, чем допускают возможности одного диска, подойдет *секционирование* (см. главу 6). Когда разделы размещаются на разных машинах, каждый такой раздел становится отдельным журналом, который может быть прочитан и записан независимо от других. Тему можно определить как группу разделов, обслуживающих однотипные сообщения. Такой подход проиллюстрирован на рис. 11.3.

Внутри каждого раздела брокер назначает каждому сообщению монотонно увеличивающийся порядковый номер *(смещение)*. На рис. 11.3 смещения сообщений

показаны в виде номеров в рамках. Такие порядковые номера оправданны, потому что раздел открыт только для добавления записей, поэтому сообщения внутри него всегда упорядочены. На разные разделы данная гарантия не распространяется.

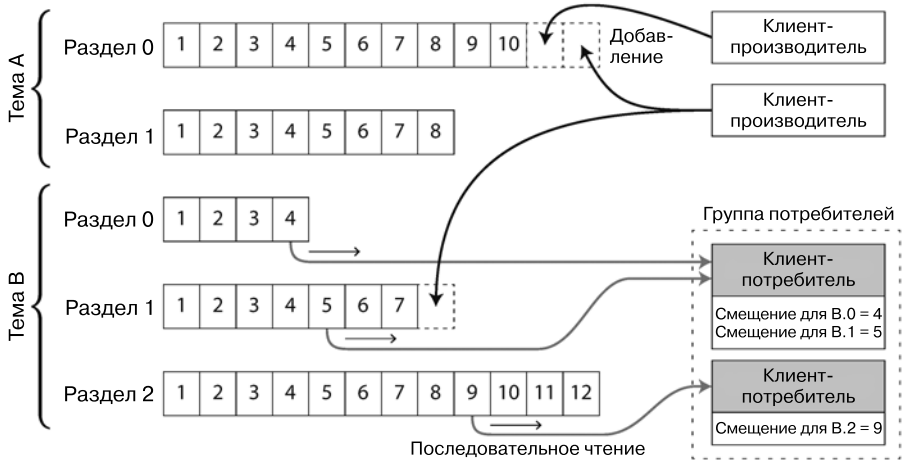


Рис. 11.3. Инициаторы отправляют сообщения, добавляя их в файл, относящийся к разделу темы, а потребители последовательно читают эти файлы

По указанной технологии работают такие брокеры сообщений на основе журналирования, как Apache Kafka [17, 18], Amazon Kinesis Streams [19] и Twitter DistributedLog [20, 21]. Google Cloud Pub/Sub имеет сходную архитектуру, но вместо абстракции журнала предоставляет API в стиле JMS [16]. Несмотря на то что эти брокеры записывают все сообщения на диск, они способны обеспечивать пропускную способность до миллионов сообщений в секунду за счет секционирования на нескольких машинах и высокую отказоустойчивость — благодаря репликации сообщений [22, 23].

Что лучше: журналы или традиционный обмен сообщениями?

Журнал — простейший способ обеспечить рассылку сообщений: несколько пользователей могут читать журнал независимо друг от друга, при этом прочитанные сообщения не удаляются из него. Для обеспечения равномерного распределения нагрузки в группе потребителей брокер, вместо того чтобы направлять отдельные сообщения клиентам-потребителям, может назначать целые разделы узлам в потребительской группе.

Затем каждый клиент получает *все* сообщения из тех раздела, который был ему назначен. Обычно, когда потребителю назначается раздел журнала, он читает

сообщения последовательно, простым однопоточным способом. Этот простейший вариант распределения нагрузки имеет следующие недостатки:

- количество узлов-потребителей, совместно обрабатывающих одну тему, не может быть больше количества разделов журнала в этой теме, поскольку сообщения из каждого раздела доставляются в один и тот же узел[1];
- если какое-то сообщение медленно обрабатывается, то задерживает обработку следующих сообщений этого раздела (блокировка заголовка строки, см. подраздел «Описание производительности» раздела 1.3).

Таким образом, в случаях, когда для обработки сообщений требуются значительные ресурсы и желательно ее распараллелить по принципу «сообщение за сообщением», причем их порядок не особенно важен, предпочтительным является стиль брокера сообщений JMS/AMQP. Однако в ситуациях с высокой пропускной способностью, при быстрой обработке сообщений и необходимости соблюдать их последовательность применение журнала очень эффективно.

Смещения потребителей

Последовательное использование раздела позволяет легко определить, какие сообщения были обработаны: все из них, смещение которых меньше, чем текущее смещение потребителя, уже обработаны, а сообщения с большим смещением еще не были просмотрены. Таким образом, брокеру не приходится отслеживать подтверждения для каждого сообщения — достаточно лишь периодически записывать смещения потребителей. Такое сокращение объема вспомогательных операций, возможности пакетной и конвейерной обработки помогают увеличить пропускную способность систем на основе журналирования.

На практике это смещение очень похоже на регистрационный номер транзакции в журнале, часто встречающийся при репликации базы данных с одним ведущим узлом и обсуждавшийся в подразделе «Создание новых ведомых узлов» раздела 5.1. При репликации БД регистрационный номер транзакции в журнале позволяет повторно подключиться к ведущему узлу в случае его отключения и возобновить репликацию без потери записей. Точно такой же принцип используется и здесь: брокер сообщений ведет себя как база-ведущий, а потребитель — как ведомая база.

При выходе потребительского узла из строя его разделы назначаются другому узлу из данной группы потребителей и тот начинает обрабатывать сообщения от

[1] Можно создать схему распределения нагрузки, в которой два пользователя обрабатывают один раздел. Притом оба читают весь набор сообщений, но один рассматривает сообщения только с четными смещениями, а другой — только с нечетными. Или же можно распределить обработку сообщений среди пула потоков, но это усложняет управление смещением для потребителей. Вообще говоря, однопоточная обработка раздела предпочтительнее, а для достижения большего параллелизма подойдет увеличение количества разделов.

последнего записанного смещения. Если потребитель их обработал, но не успел записать смещения, то после перезапуска они будут обработаны вторично. Способы решения этой проблемы мы обсудим позже в настоящей главе.

Использование дискового пространства

При постоянном ведении журнала рано или поздно наступит момент, когда на диске перестанет хватать места. Чтобы освободить дисковое пространство, на практике журнал делится на сегменты; время от времени старые сегменты удаляются или перемещаются в архивное хранилище. (Позже мы обсудим более сложный способ освобождения дискового пространства.)

Это значит следующее: если медленный потребитель не справляется со скоростью поступления сообщений и настолько отстает, что его потребительское смещение указывает на удаленный сегмент, то он пропустит некоторые сообщения. По сути, журнал представляет собой буфер ограниченного размера, отбрасывающий старые сообщения по мере заполнения. Такой буфер также называют *круговым* или *кольцевым*. Однако вследствие своего нахождения на диске он может быть довольно большим.

Сделаем приблизительный подсчет. На момент написания этой книги емкость типичного большого жесткого диска составляла 6 Тбайт, а скорость последовательной записи — 150 Мбайт/с. Если записывать сообщения с максимально возможной скоростью, то диск заполнится примерно через 11 часов. Таким образом, он способен служить буфером для сообщений за 11 часов, после чего начнет перезаписывать старые сообщения. При использовании нескольких дисков и машин это соотношение не изменится. На практике редко прибегают к полной скорости записи диска, так что журнал обычно способен хранить буфер за несколько дней или даже недель.

Независимо от того, как долго хранятся сообщения, скорость записи журнала остается более или менее постоянной, поскольку все сообщения в любом случае записываются на диск [18]. В этом кроется отличие от систем обмена сообщениями: они по умолчанию сохраняют сообщения в памяти и записывают их на диск только в том случае, если очередь становится слишком длинной. Такие системы бывают быстрыми для коротких очередей и становятся намного медленнее, когда начинают записывать данные на диск, их скорость зависит от объема сохраненной истории.

Когда потребители не успевают за инициаторами

В начале подраздела «Системы обмена сообщениями» текущего раздела рассматривались три варианта действий на тот случай, если потребитель не успевает за той скоростью, с которой инициаторы отправляют сообщения: удаление, буферизация или контроль обратного потока. В этой системе понятий журналирование — вариант буферизации с большим буфером фиксированного размера (ограниченным доступной емкостью диска).

При отставании потребителя настолько сильном, что требуемые ему сообщения старше сохраненных на диске, он не сможет прочитать эти сообщения: брокер отбрасывает сообщения, хранящиеся слишком долго, если размер буфера не позволяет хранить их дольше. Можно следить за тем, насколько потребитель отстает от начала журнала, и выдать предупреждение, если отставание значительное. Поскольку буфер достаточно велик, у оператора довольно много времени, чтобы исправить ситуацию и позволить медленному потребителю сократить отставание прежде, чем он начнет пропускать сообщения.

Но даже если потребитель отстал слишком сильно и начал пропускать сообщения, данная проблема затрагивает только одного его; обслуживание других потребителей не нарушается. Это большое операционное преимущество: можно экспериментировать с журналом инициаторов в целях разработки, тестирования и отладки, не беспокоясь о нарушении всей работы предприятия. Допустим, потребитель отключится или даст сбой; тогда он всего лишь перестает потреблять ресурсы — и от него останется только его потребительское смещение.

Такая архитектура резко отличается от традиционных брокеров сообщений, где нужно тщательно удалять запросы, чьи потребители были отключены, — в противном случае они продолжают напрасно накапливать сообщения и отнимать память у потребителей, которые все еще активны.

Повторение старых сообщений

Как уже отмечалось, в брокерах сообщений типа AMQP и JMS обработка и подтверждение сообщений является деструктивной операцией, поскольку в результате брокер удаляет эти сообщения. В брокере сообщений на основе журналирования, наоборот, обработка больше похожа на чтение из файла: это операция только чтения, не меняющая журнал.

Единственный побочный эффект обработки, кроме генерации потребителем выходных данных, — увеличение потребительского смещения. Но смещением управляет потребитель, поэтому при необходимости им легко манипулировать: например, можно запустить копию потребителя со вчерашними смещениями и записать выходные данные в другое место, чтобы заново обработать сообщения последнего дня. Такую операцию можно повторять многократно, изменяя код обработки.

Указанное обстоятельство делает обработку сообщений на основе журналирования похожей на пакетные процессы, описанные в предыдущей главе, где выходные данные четко отделяются от входных с помощью повторяемого процесса преобразования. Это позволяет широко экспериментировать и упрощает восстановление после ошибок и сбоев, предоставляя хороший инструмент для интеграции потоков данных внутри организации [24].

11.2. Базы данных и потоки

Мы провели ряд сравнений между брокерами сообщений и базами данных. По традиции их принято считать отдельными категориями инструментов, но мы увидели, что брокеры сообщений на основе журналирования успешно восприняли некоторые идеи из БД в применении к обмену сообщениями. Можно пойти и в обратном направлении: взять идеи из области обмена сообщениями и потоков и применить их к базам.

Как уже отмечалось, событие — запись о том, что произошло в некий момент времени. Это может быть не только действие пользователя (например, ввод поискового запроса) или чтение показаний датчика, но и *запись в базу данных*. Факт записи в БД является событием, которое может быть зафиксировано, сохранено и обработано. Таким образом, связь между базами и потоками глубже, чем просто физическое хранилище журналов на диске, — она довольно фундаментальна.

Фактически журнал репликации (см. подраздел «Реализация журналов репликации» раздела 5.1) представляет собой поток событий записи в базу данных, создаваемый ведущим узлом в процессе обработки транзакции. Ведомые применяют этот поток записей к своим копиям БД и, таким образом, получают свою точную копию. События в журнале репликации описывают изменения данных.

В подразделе «Рассылка общей последовательности» раздела 9.3 мы также столкнулись с принципом *репликации конечных автоматов*, который гласит: если каждое событие представляет собой запись в базу данных и каждая реплика обрабатывает одни и те же события в одном и том же порядке, то все реплики будут завершены в том же конечном состоянии. (Обработка события считается неделимой операцией.) Это просто еще один случай потоков событий!

В этом разделе мы сначала рассмотрим проблему, возникающую в гетерогенных системах данных, а затем исследуем пути ее решения, перенося идеи из потоков событий в базы.

Синхронизация систем

Читая эту книгу, вы уже, вероятно, поняли, что единой системы, которая бы удовлетворяла всем требованиям хранения, запросов и обработки данных, не существует. На практике большинству нетривиальных приложений для удовлетворения своих потребностей требуется объединить разные технологии: например, задействовать базу данных OLTP для пользовательских запросов, кэш для ускоренной обработки общих запросов, полнотекстовый индекс для поисковых запросов и склад данных для аналитики. Каждая из этих систем имеет свою копию данных, хранящихся в собственном представлении, оптимизированном для ее целей.

Поскольку одни и те же или связанные данные присутствуют в нескольких местах, их необходимо синхронизировать: если какой-либо элемент обновляется в БД, то должен обновиться также в кэше, индексах поиска и складе данных. С помощью последних эта синхронизация данных обычно выполняется благодаря процессам ETL (см. подраздел «Складирование данных» раздела 3.2), часто путем создания полной копии БД, ее преобразования и массовой загрузки в склад — другими словами, пакетного процесса. Аналогичным образом, как было показано в подразделе «Выходные данные пакетных потоков» раздела 10.2, задействуя пакетные процессы, можно создать поисковые индексы, системы выдачи рекомендаций и другие производные системы обработки информации.

Если периодически делать полный дамп базы слишком долго, иногда используется альтернативный вариант — *двойная запись*, при которой в случае изменения данных приложение явно записывает информацию в каждую из систем: например, сначала заносит данные в базу, затем обновляет поисковый индекс, а потом аннулирует записи кэша (или даже выполняет все эти операции одновременно).

Однако у двойной записи есть ряд серьезных проблем, одна из которых — состояние гонки, показанное на рис. 11.4. В этом примере два клиента конкурентно хотят обновить элемент X: клиент 1 хочет присвоить ему значение A, а клиент 2 — значение B. Оба клиента сначала записывают новое значение в БД, а затем — в индекс поиска. Из-за неудачного хронометража запросы чередуются: база сначала принимает запись от клиента 1 и устанавливает значение A, а затем — запись от клиента 2 и устанавливает значение B, вследствие чего окончательное значение элемента в базе равно B. Поисковый индекс, наоборот, сначала видит запись от клиента 2, а затем — от клиента 1, так что конечным значением в индексе поиска является A. Теперь эти две системы несовместимы, даже несмотря на отсутствие ошибки.

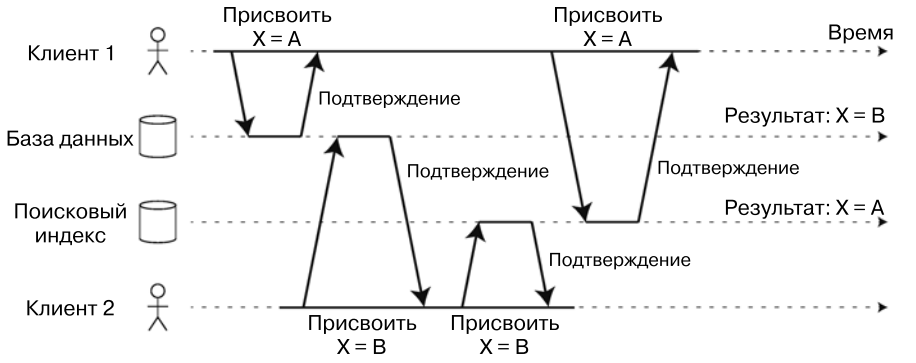


Рис. 11.4. В базе данных элементу X сначала присваивается значение A, а потом — B, тогда как в поисковом индексе запись происходит в обратном порядке

Если у вас нет специального механизма обнаружения конкурентных процессов, такого как векторы версий, обсуждавшиеся в подразделе «Обнаружение конкурентных операций записи» раздела 5.4, то вы даже не заметите, что такая операция имела место — просто одно значение будет тихо заменено другим.

Другая проблема с двойной записью заключается в том, что одна из записей может оказаться неудачной, а вторая — успешной. Это уже проблема не конкуретного доступа, а отказоустойчивости, но она также приводит к несовместимости двух систем. Гарантировать то, что обе записи были либо успешными, либо нет, — задача атомарной фиксации, и ее решение стоит дорого (см. подраздел «Атомарная и двухфазная фиксация (2PC)» раздела 9.4).

При наличии только одной реплицированной базы данных с одним ведущим узлом последний определяет порядок записи, поэтому здесь будет действенной репликация конечного автомата в сочетании с репликами базы. Однако на рис. 11.4 ведущих узлов нет: у БД может быть свой такой узел, а у поискового индекса — свой, но ни один из них не подчиняется другому, что может вызвать конфликты (см. раздел 5.3).

Если бы ведущий узел был только один — например, база данных — и если сделать поисковый индекс ведомым, то это улучшило бы ситуацию. Но возможно ли такое сочетание на практике?

Перехват изменений данных

Проблема большинства журналов репликации БД заключается в том, что они долгое время считались внутренней деталью реализации базы, а не открытым API. Предполагается, что клиенты направляют запрос в базу, используя ее модель данных и язык запросов, а не анализируют журналы репликации, чтобы извлечь оттуда данные.

Десятки лет в документации многих БД просто не было описания того, как прочитать журнал изменений. По этой причине было сложно получить все изменения из базы и скопировать их в другую систему хранения — поисковый индекс, кэш или склад данных.

В последнее время растет интерес к *перехвату изменений данных* (change data capture, CDC) — процессу наблюдения за всеми изменениями данных, записанными в базу, и их извлечения в той форме, в какой они могут быть реплицированы в другие системы. CDC особенно интересен, если изменения доступны в виде потока сразу же после их записи.

Например, можно перехватывать изменения базы данных и постоянно применять их к поисковому индексу. Если журнал изменений применяется в том же порядке,

то можно ожидать, что данные в поисковом индексе будут соответствовать данным в базе. Поисковый индекс и другие производные системы обработки информации являются обычными потребителями потока изменений (рис. 11.5).

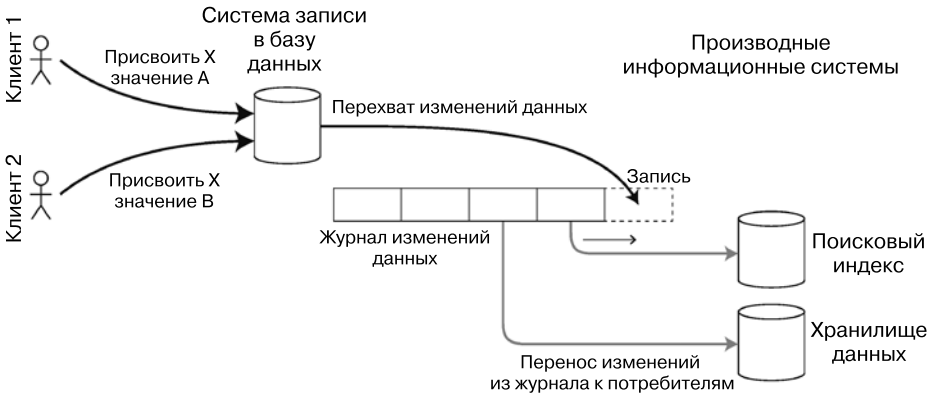


Рис. 11.5. Данные принимаются в том порядке, в каком они были записаны в базу, а затем изменения в том же порядке переносятся на другие системы

Реализация перехвата изменений данных

Как говорилось во введении к данной части, потребителей журнала можно назвать *производными информационными системами*: данные, хранящиеся в поисковом индексе и на складе, — просто другое представление данных из системы записи. Перехват изменений данных представляет собой механизм, гарантирующий, что все изменения, внесенные в систему записи, распространятся и на производные информационные системы, в которых будут созданы точные копии этих данных.

По сути, при перехвате изменений одна база данных становится ведущим узлом (источником, из которого перехватываются изменения), а остальные — ведомыми. Для переноса изменений событий из исходной базы хорошо подходит брокер сообщений на основе журналирования, поскольку сохраняет последовательность сообщений (и исключает проблему переупорядочения, показанную на рис. 11.2).

Для реализации перехвата изменений можно использовать триггеры базы данных (см. пункт «Триггерная репликация» подраздела «Реализация журналов репликации» раздела 5.1): создать триггеры, которые отслеживали бы изменения таблиц данных и добавляли в таблицу изменений соответствующие записи. Однако триггеры имеют тенденцию давать сбои и требуют значительных накладных расходов. Синтаксический анализ журнала репликации — более надежный подход, хотя и он сопряжен с проблемами, такими как отслеживание изменений схемы.

Эта идея, будучи масштабированной, используется в LinkedIn Databus [25], Facebook Wormhole [26] и Yahoo! Sherpa [27]. В Bottled Water реализован CDC

для PostgreSQL, где специальный API декодирует журнал с упреждающей записью [28]. В Maxwell и Debezium реализовано нечто похожее для MySQL, с синтаксическим анализом двоичного журнала [29–31]. Mongoriver читает операционный журнал MongoDB [32, 33], а GoldenGate предоставляет аналогичные возможности для Oracle [34, 35].

Как и брокеры сообщений, перехват данных обычно является асинхронным: система записи данных в базу не ждет, пока изменения будут применены у потребителей, прежде чем завершить операцию. Эксплуатационные преимущества такой архитектуры заключаются в том, что появление медленного потребителя не очень сильно влияет на систему записи, но имеет такой недостаток: ей свойственны все проблемы, связанные с задержкой репликации (см. раздел 5.2).

Исходный снимок

Если есть журнал всех изменений, которые когда-либо делались в базе данных, то, повторив его, можно полностью восстановить ее состояние. Однако постоянное хранение всех изменений зачастую требует чрезмерно большого объема дискового пространства, а воспроизведение занимает слишком много времени, поэтому журнал должен быть сокращен.

Например, для построения нового полнотекстового индекса требуется полная копия всей базы данных — журнала последних изменений недостаточно, поскольку в нем отсутствуют давно не обновлявшиеся элементы. Поэтому при отсутствии полного журнала истории нужно начать с надежного снимка, как обсуждалось в подразделе «Создание новых ведомых узлов» раздела 5.1.

Снимок базы данных должен соответствовать известной позиции или смещению в журнале изменений, чтобы знать, в какой момент начать применять изменения после обработки снимка. В некоторых инструментах CDC интегрированы средства создания такого снимка, в других предполагается ручное выполнение этой операции.

Уплотнение журнала

Если есть возможность хранить журнал только ограниченного объема, то нужно создавать снимок базы перед каждым добавлением новой производной информационной системы. Хорошей альтернативой этому является *уплотнение журнала*.

Мы уже обсуждали такое уплотнение в подразделе «Хеш-индексы» раздела 3.1, в контексте журнализированных подсистем хранения (см. пример на рис. 3.2). Его принцип прост: подсистема периодически просматривает журнал и при нахождении записи с одинаковыми ключами отбрасывает дубликаты, оставляя только последнее обновление для каждого ключа. Этот процесс уплотнения и слияния выполняется в фоновом режиме.

Если в журналированной подсистеме хранения запись помечена специальным нулевым значением *(отметка об удалении)*, это значит, что ее ключ был удален, как и сама запись при первом же уплотнении журнала. Но до тех пор, пока ключ не перезаписан и не удален, запись сохраняется в журнале. Объем дискового пространства, необходимый для хранения уплотненного журнала, зависит только от текущего содержимого базы данных, а не от количества операций записей, которые когда-либо выполнялись в ней. При частой перезаписи одного и того же ключа его предыдущие значения будут удалены и сохранится только последнее из них.

Такая же идея работает в контексте брокеров сообщений на основе журналирования и перехвата изменений данных. Если система CDC настроена так, что у каждого изменения есть первичный ключ и каждое обновление ключа заменяет его предыдущее значение, то достаточно хранить только самую последнюю запись для каждого ключа.

Теперь всякий раз, когда требуется восстановить производную информационную систему, такую как поисковый индекс, можно создать в теме уплотненного журнала нового потребителя со смещением 0 и последовательно сканировать все сообщения из журнала. В нем гарантированно содержатся самые последние значения для всех ключей БД (и, вероятно, некоторые старые значения) — другими словами, журнал позволяет получить полную копию содержимого БД без необходимости делать еще один снимок исходной базы данных CDC.

Уплотнение журнала поддерживается в Apache Kafka. Как мы увидим далее в этой главе, оно позволяет использовать брокер сообщений для долговременного хранения сообщений, а не только для передачи при их обмене.

API для потоков изменений

Все чаще встречаются базы с поддержкой потоков изменений как интерфейса первого уровня вместо модифицированных и реконструированных CDC. Например, в RethinkDB можно подписаться на уведомления, когда результат запроса меняется [36]; Firebase [37] и CouchDB [38] обеспечивают синхронизацию данных на основе ленты изменений, которая также доступна для приложений, а Meteor задействует журнал операций MongoDB для подписки на изменения данных и обновление пользовательского интерфейса [39].

В VoltDB транзакции могут непрерывно экспортировать данные из базы в виде потока [40]. База данных представляет выходной поток в реляционной модели как таблицу, в которую транзакции могут вставлять кортежи, но к которой запрещены запросы. Поток состоит из журнала кортежей, совершенных транзакциями и записанных в эту специальную таблицу в порядке их выполнения. Внешние потребители могут асинхронно читать такой журнал и использовать его для обновления производных информационных систем.

Kafka Connect [41] — это попытка объединить Kafka с инструментами сбора информации об изменениях данных для широкого спектра баз. Поскольку поток событий изменений находится в Kafka, его можно применять для обновления производных информационных систем, таких как поисковые индексы, а также для передачи в потоковые системы обработки, как будет описано далее в настоящей главе.

Источники событий

Есть несколько параллелей между рассмотренными здесь идеями и *источниками событий* — технологией, разработанной сообществом предметно-ориентированного проектирования (domain-driven design, DDD) [42–44]. Мы кратко обсудим источники событий, поскольку на них основаны некоторые полезные и актуальные идеи для потоковых систем.

Подобно перехвату изменений данных, источники событий подразумевают сохранение всех изменений состояния приложения в виде журнала изменений. Главное отличие заключается в том, что в технологии источников событий эта идея используется на другом уровне абстракции.

- При перехвате изменений данных приложение использует базу разными способами, обновляя и удаляя записи по мере необходимости. Журнал изменений извлекается из БД на низком уровне (например, путем синтаксического анализа журнала репликации). Это гарантирует извлечение записей из базы в том же порядке, в котором они были туда внесены, и условия гонки, продемонстрированные на рис. 11.4, не возникнут. Приложению, записывающему данные в базу, не нужно знать, что происходит CDC.
- Применительно к источникам событий логика приложения явно построена на основе неизменяемых событий, которые записываются в журнал. В этом случае события дописываются в хранилище, обновления или удаления не рекомендованы или запрещены. События отражают то, что происходит на уровне приложений, а не изменения низкоуровневых состояний.

Источники событий — эффективная технология моделирования данных: с точки зрения приложения более важно записывать действия пользователя как неизменные события, а не результат этих действий в изменяемой базе. Источники облегчают развитие приложений, помогают при отладке, упрощая понимание того, что и почему происходит, и защищают от ошибок приложений (см. пункт «Преимущества неизменяемых событий» подраздела «Состояние, потоки и неизменяемость» текущего раздела).

Например, при сохранении события «студент отказался от зачисления на курс» ясно показана цель действия, и это выражено в нейтральной форме, тогда как по его результату «запись была удалена из таблицы зачислений, причина отмены добавлена в таблицу обратной связи со студентом» можно сделать различные предположения

о том, как будут использоваться данные позднее. Если вводится новая функция приложения — например, «место предложено следующему претенденту из списка ожидания», — то источники событий позволяют легко связать новый результат с существующим событием.

Источники событий похожи на хронологическую модель хранения данных [45], а также на журнал событий и таблицу фактов, используемые в схеме типа «звезда» (см. подраздел «“Звезды” и “снежинки”: схемы для аналитики» раздела 3.2).

Специализированные базы данных, такие как Event Store [46], были разработаны для поддержки приложений с помощью источников событий, но технология в целом не зависит от какого-либо конкретного инструмента. Для создания приложений в этом стиле можно использовать также обычную БД или брокер сообщений на основе журналирования.

Получение текущего состояния из журнала событий

Сам по себе журнал событий не очень полезен: обычно пользователи ожидают увидеть текущее состояние системы, а не историю его изменений. Например, в интернет-магазине покупатель ожидает увидеть текущее содержимое своей корзины, а не список всех изменений, которые он когда-либо в ней делал.

Таким образом, приложения, применяющие источники событий, должны брать журнал событий (представляющий данные, *записанные* в систему) и преобразовывать их в состояние, подходящее для представления пользователю (способ *чтения* данных из системы [47]). Это преобразование может задействовать произвольную логику, но оно должно быть однозначным, чтобы при многократном выполнении генерировать по данным из журнала событий одинаковое состояние приложения.

Как и при перехвате изменений данных, воспроизведение журнала событий позволяет восстановить текущее состояние системы. Однако уплотнение журнала необходимо выполнять по-разному.

- Событие CDC для обновления записи, как правило, содержит всю новую версию записи, вследствие чего текущее значение для первичного ключа полностью определяется самым последним событием для этого ключа. При уплотнении журнала предыдущие события для данного ключа могут быть отброшены.
- Применительно к источникам событий, наоборот, события моделируются на более высоком уровне: обычно событие отражает действия пользователя, а не детали обновления состояния, которое произошло в результате этого действия. В таком случае последующие события обычно не перезаписывают предшествующие, так что для восстановления конечного состояния нужна полная история событий. Уплотнение журнала в данном случае невозможно.

Приложения, использующие источники событий, как правило, имеют некий механизм для хранения моментальных снимков текущего состояния системы, которое получают из журнала событий, поэтому им не нужно многократно обрабатывать весь журнал. Однако данная оптимизация касается только производительности — она позволяет ускорить чтение и восстановление после сбоев; система должна постоянно хранить все необработанные события и при необходимости обрабатывать журнал целиком. Это предположение будет рассмотрено в пункте «Ограничения неизменяемости» подраздела «Состояние, потоки и неизменяемость» текущего раздела.

Команды и события

Философия источников событий четко различает *события* и *команды* [48]. Когда поступает запрос от пользователя, вначале он представляет собой команду. На данном этапе тоже вероятен сбой — например, при нарушении некоего условия целостности. Сначала приложение должно проверить, может ли выполнить команду. Если проверка прошла успешно и команда принята, то она становится событием, которое является долговечным и неизменным.

Например, если пользователь пытается зарегистрироваться под каким-либо именем, или забронировать место в самолете, или купить билет в театр, то приложение должно убедиться, что это имя или место еще не занято. (Такой пример был рассмотрен в подразделе «Отказоустойчивый консенсус» раздела 9.4.) При успешном прохождении проверки приложение может сгенерировать событие для демонстрации того, что данное имя пользователя зарегистрировано и ему присвоен определенный ID или что место зарезервировано для данного клиента.

В тот момент, когда событие сгенерировано, оно становится *фактом*. Даже если клиент впоследствии изменит свое решение или отменит бронирование, факт остается фактом: ранее он сделал резервирование определенного места, а изменение или аннулирование — это уже другое событие, которое добавится позже.

Потребителю потока событий не разрешается отклонять событие: к тому времени, когда потребитель увидит событие, оно уже является неизменной частью журнала и, возможно, его уже видели другие потребители. Таким образом, любая проверка команды должна выполняться синхронно, прежде чем она станет событием, — например, с помощью сериализируемой транзакции, которая проверяет команду как неделимую единицу и затем публикует событие.

Вероятен другой вариант: запрос пользователя на бронирование места может быть разделен на два события — сначала предварительное резервирование, а затем отдельное событие подтверждения после проверки бронирования (как было описано в пункте «Реализация линеаризуемого хранилища с помощью рассылки общей последовательности» подраздела «Рассылка общей последовательности» раздела 9.3). Такое разделение позволяет выполнять валидацию в асинхронном процессе.

Состояние, потоки и неизменяемость

Как было показано в главе 10, преимуществом пакетной обработки является неизменяемость входных файлов. Благодаря этому можно запускать экспериментальные задачи обработки на рабочих входных файлах, не опасаясь их повредить. Именно принцип неизменяемости также делает эффективными инструментами источники событий и перехват изменений данных.

Мы привыкли хранить текущее состояние приложения в базе данных — такое представление оптимизировано для операций чтения и обычно наиболее удобно для обслуживания запросов. Состояние приложений постоянно меняется, вследствие чего базы поддерживают обновление, удаление и вставку данных. Как данное обстоятельство согласуется с неизменяемостью?

Всякий раз, когда состояние изменяется, это является результатом изменившихся событий. Например, список свободных мест есть результат операций бронирования, ранее обработанных системой, текущий баланс на счету — результат кредитных и дебетных операций со счетом, а график времени отклика для веб-сервера — результат сбора показаний времени отклика для всех веб-запросов, которые произошли за определенное время.

Независимо от того, как именно изменяется состояние, всегда существует последовательность событий, вызвавших эти изменения. Даже если операция была сделана, а потом отменена, факт остается фактом: события произошли. Основная идея заключается в том, что изменяемое состояние и список событий, открытый только для дополнения, не противоречат друг другу: это две стороны одной и той же медали. *Журнал изменений* отражает эволюцию состояния системы с течением времени.

В математическом представлении состояние приложения — интеграл потока событий по времени, а поток изменений — производная состояния по времени (рис. 11.6) [49–51]. Аналогия имеет ограничения (например, вторая производная от состояния не имеет смысла), но является полезной отправной точкой, когда речь идет о данных.

$$state(now) = \int_{t=0}^{now} stream(t)\,\mathrm{d}t \qquad stream(t) = \frac{\mathrm{d}\,state(t)}{\mathrm{d}t}$$

Рис. 11.6. Связь между текущим состоянием приложения и потоком событий

Долговременное хранение журнала изменений приводит к простому результату: состояние является воспроизводимым. Если журнал событий является системой записи и из него можно получить любое изменяемое состояние, то будет

легче рассуждать о потоке данных через систему. Как говорит Пэт Хелланд (Pat Helland) [52]:

> «В журнал транзакций заносятся все изменения базы данных. Единственный способ изменить журнал — высокоскоростные операции добавления. В этом смысле содержимое БД — кэш последних записей журнала. Истина является журналом. База представляет собой лишь кэш подмножества журнала. Такое кэшированное подмножество хранит последние значения всех записей и индекса из журнала».

Уплотнение журнала, как было показано в одноименном пункте подраздела «Перехват изменений данных» текущего раздела, является одним из способов преодоления различия между журналом и состоянием базы данных: в уплотненном журнале хранятся только последние версии всех записей, а перезаписанные отбрасываются.

Преимущества неизменяемых событий

Неизменяемость в базах данных — старая идея. Например, бухгалтеры веками использовали неизменяемость в финансовом учете. В момент возникновения транзакция фиксируется в *реестре*, доступном только для дописывания. Реестр — это журнал событий, в котором описываются деньги, товары или услуги, перешедшие из рук в руки. Информацию записей учета, таких как прибыль, убыток или баланс, получают из обработки данных о транзакциях, записанных в этой книге [53].

При ошибке бухгалтер не удаляет и не изменяет неверную транзакцию в реестре, а добавляет еще одну, которая компенсирует ошибку, — например, возвращает неправильный платеж. Неверная транзакция остается в реестре навсегда, поскольку данный факт может быть важен в целях аудита. Если неверные цифры, полученные из реестра, уже опубликованы, то цифры за следующий отчетный период будут отражать коррекцию. В бухгалтерском учете это совершенно нормальный процесс [54].

В первую очередь такой контроь важен в финансовых системах, но он также полезен и для многих других систем, в которых нет столь строгого регулирования. Как обсуждалось в пункте «Философия выходных данных пакетного процесса» подраздела «Выходные данные пакетных потоков» раздела 10.2, при случайном запуске ошибочного кода, который запишет в базу неправильные данные и при этом способен перезаписать старые данные, восстановление будет намного сложнее. Благодаря журналу неизменяемых событий, доступному только для дописывания, гораздо легче диагностировать, что произошло, и устранить проблему.

Неизменяемые события хранят больше информации, чем только текущее состояние. Например, в интернет-магазине клиент может добавить товар в корзину,

а затем удалить его оттуда. Хотя с точки зрения выполнения заказа второе событие отменяет первое, в целях аналитики может быть полезно знать, что клиент собирался купить данный товар, но затем отказался. Вероятно, он захочет купить его позже, или стоит предложить ему замену. Эта информация записывается в журнал событий, но будет потеряна в БД, так как в ней из корзины будут удалены соответствующие элементы при удалении товара [42].

Получение нескольких представлений из одного журнала событий

И это еще не все. Отделяя изменяемое состояние от журнала неизменяемых событий, из одного и того же журнала событий можно получить разные представления, рассчитанные на чтение. Здесь можно провести аналогию с несколькими потребителями одного потока (рис. 11.5): например, используя этот метод, аналитическая база Druid получает данные непосредственно из Kafka [55], Pistachio — распределенное хранилище данных типа «ключ — значение» — применяет Kafka как журнал фиксации транзакций [56], а приемники Kafka Connect экспортируют данные из Kafka в различные БД и индексы [41]. Для многих других систем хранения и индексирования, таких как поисковые серверы, было бы целесообразно аналогичным образом получать входные данные из распределенного журнала (см. подраздел «Синхронизация систем» текущего раздела).

Наличие явного этапа перевода данных из журнала событий в базу упрощает дальнейшее развитие приложения: чтобы ввести новую функцию, которая будет представлять данные каким-то новым способом, можно построить на базе журнала событий отдельное представление, оптимизированное для чтения, для новой функции и запустить его параллельно с существующими системами, не меняя их. Выполнить последнее часто бывает проще, чем сложную миграцию внутри существующей системы. По мере того как надобность в старой системе отпадает, ее можно просто выключить и освободить занимаемые ею ресурсы [47, 57].

Хранить данные обычно довольно просто, если только не нужно беспокоиться о запросах и предоставлении доступа; сложные архитектуры, системы индексирования и хранения часто появляются для того, чтобы поддерживать определенные шаблоны запросов и доступа (см. главу 3). Разделяя форму записи данных от формы чтения и обеспечивая несколько разных представлений для чтения, мы получаем большую гибкость. Эту идею иногда называют *разделением ответственности на команды и запросы* (command query responsibility segregation, CQRS) [42, 58, 59].

Традиционный подход к построению БД и схемы основан на ошибочном представлении о том, что данные должны записываться и выдаваться по запросу в одной и той же форме. Споры о нормализации и денормализации (см. подраздел «Связи

“многие-к-одному” и “многие-ко-многим”» раздела 2.1) становятся во многом неактуальными, если есть возможность переводить данные из оптимизированного для записи журнала событий в оптимизированное для чтения состояние приложения. Это связано с тем, что вполне разумно денормализировать данные в оптимизированных для чтения представлениях, поскольку процесс преобразования предоставляет механизм, позволяющий обеспечивать его соответствие представлению с журналом событий.

В подразделе «Описание нагрузки» раздела 1.3 обсуждались ленты твитов Twitter, кэш недавно написанных твитов, которые отслеживает конкретный пользователь (например, почтовый ящик). Это еще один пример состояния, оптимизированного для чтения: ленты твитов сильно денормализованы, поскольку твиты дублируются в лентах разных людей, отслеживающих вас. Но сервис разветвления сохраняет это дублированное состояние со всеми новыми твитами и новыми подписками, что позволяет управлять дублированием.

Контроль конкурентных процессов

Самый большой недостаток источников данных и перехвата изменений данных — действия потребителей журнала событий обычно являются асинхронными, вследствие чего существует вероятность, что пользователь сделает запись в журнал, затем попробует ее прочитать из производного представления и обнаружит, что эта запись еще не отражена в представлении для чтения. Такая проблема и ее возможные решения уже обсуждались в подразделе «Читаем свои же записи» раздела 5.2.

Одним из решений данной проблемы является обновление представления для чтения одновременно с добавлением события в журнал. Для этого нужно, чтобы в транзакции искомые операции записи были объединены в неделимую единицу, независимо от того, нужно сохранить журнал событий и производить чтение в одной системе хранения или же требуется распределенная транзакция в разных системах. Вместо этого можно использовать технологию, описанную в пункте «Реализация линеаризуемого хранилища с помощью рассылки общей последовательности» подраздела «Рассылка общей последовательности» раздела 9.3.

Впрочем, получение текущего состояния из журнала событий упрощает некоторые аспекты контроля конкурентного доступа. По большей части потребность в транзакциях с несколькими объектами (см. подраздел «Однообъектные и многообъектные операции» раздела 7.1) связана с одним действием пользователя, требующим изменения данных в нескольких местах. С помощью источников событий можно построить событие таким образом, чтобы оно было самодостаточным, полностью описывая действие пользователя. Такое действие потребует всего одной записи в одном месте — а именно, добавления события в журнал, — и его легко сделать неделимым.

Если журнал событий и состояние приложения одинаково секционированы (например, для обработки события для клиента в секции 3 в состоянии приложения требуется только обновление данной секции), то простой однопоточный потребитель журнала не нуждается в контроле конкурентного доступа для записи — по своей природе он способен обрабатывать события только последовательно (см. также подраздел «По-настоящему последовательное выполнение» раздела 7.3). Чтобы избежать неопределенности при конкурентном доступе, в журнале всем событиям секции присваиваются порядковые номера [24]. Если же событие касается нескольких секций, то требуется немного больше работы — об этом мы поговорим в главе 12.

Ограничения неизменяемости

Многие системы, в которых не используется модель, основанная на событиях, тем не менее полагаются на неизменность: внутри баз существуют неизменяемые структуры данных, или же хранится несколько версий данных для поддержки моментальных снимков (см. пункт «Индексы и изоляция снимков состояния» подраздела «Изоляция снимков состояния и воспроизводимое чтение» раздела 7.2). Системы контроля версий, такие как Git, Mercurial и Fossil, также полагаются на неизменяемые данные при хранении истории версий файлов.

Насколько реально хранить неизменяемую историю всех изменений постоянно? Ответ зависит от количества перезаписей в наборе данных. Одни приложения главным образом добавляют данные и редко обновляют или удаляют их; такие данные легко сделать неизменными. Другие приложения имеют высокую долю обновлений и удалений на сравнительно небольшом наборе; в этих случаях неизменяемая история способна быстро разрастаться, фрагментация может стать проблемой, а скорость сжатия и сборки мусора становится решающей для оперативной устойчивости [60, 61].

Кроме производительности, бывают и иные обстоятельства, при которых необходимо удалить данные из соображений администрирования, несмотря на принцип неизменяемости. Например, правила конфиденциальности могут потребовать удалить личную информацию пользователя после закрытия его учетной записи; закон о защите данных может требовать удаления ошибочной информации, или же понадобится справиться со случайной утечкой конфиденциальной информации.

В этих обстоятельствах недостаточно просто дописать в журнал новое событие, указывающее на то, что предыдущие данные считаются удаленными, — надо действительно переписать историю и притвориться, будто эти данные никогда не существовали. Datomic называет такую функцию *вырезанием* [62], а в системе контроля версий Fossil аналогичная концепция именуется *отторжением* [63].

По-настоящему удалить данные на удивление сложно [64], поскольку их копии могут продолжать жить во многих местах: например, в системах хранения данных, файловых системах и SSD удаленные данные часто не перезаписываются, а просто переносятся в новое место [52]. Резервные копии часто преднамеренно делаются неизменяемыми, во избежание случайного удаления или повреждения. «Удалить данные» скорее означает «затруднить получение», чем действительно «сделать получение невозможным». Тем не менее иногда приходится пытаться это сделать, как будет показано в пункте «Законодательство и саморегулирование» подраздела «Конфиденциальность и отслеживание» раздела 12.4.

11.3. Обработка потоков

До сих пор мы говорили в этой главе о том, откуда поступают потоки (события, сообщающие о действиях пользователей, показаниях датчиков, операциях записи в базу данных), и о том, как они переносятся (с помощью прямого обмена сообщениями, через брокеров сообщений, журналы событий).

Осталось обсудить, что можно сделать с имеющимся потоком, а именно как его обработать. В широком смысле есть три варианта.

1. Извлекать данные из событий и записывать их в БД, кэш, поисковый индекс или другую подобную систему хранения, откуда они могут быть запрошены другими клиентами. Как показано на рис. 11.5, это хороший способ синхронизировать базу с изменениями, происходящими в других частях системы, особенно если потребитель потока является единственным клиентом, имеющим право записи в БД. Запись в систему хранения является потоковым эквивалентом того, что обсуждалось в подразделе «Выходные данные пакетных потоков» раздела 10.2.
2. Передавать события пользователям. Например, отправив оповещения по электронной почте, или выдавая предупреждения, или транслируя поток событий на панель мониторинга в реальном времени, где они визуализируются. В этом случае конечным потребителем потока является человек.
3. Обрабатывать один или несколько входных потоков и создавать один или несколько выходных. Потоки могут проходить через конвейер, состоящий из нескольких этапов обработки, прежде чем окажутся на выходе (вариант 1 или 2).

В оставшейся части этой главы мы рассмотрим вариант 3: обработку потоков для создания других, производных потоков. Код, обрабатывающий такие потоки, называется *оператором* или *задачей*. Он тесно связан с процессами Unix и задачами MapReduce, которые обсуждались в главе 10. Структура потока данных также аналогична: процессор потока потребляет входные потоки в режиме чтения и записывает выходные данные в другое место в режиме дописывания.

Паттерны для секционирования и распараллеливания в потоковых процессорах также очень похожи на паттерны в MapReduce и подсистемы потока данных, описанные в главе 10, в связи с чем мы не станем сейчас возвращаться к этим темам. Основные операции отображения, такие как преобразование и фильтрация записей, здесь работают так же.

Существенное отличие от пакетных задач состоит в том, что поток никогда не заканчивается. Это различие имеет много последствий: как обсуждалось в начале главы, в случае неограниченного набора данных сортировка не имеет смысла, так что объединения с сортировкой слияния (см. подраздел «Объединение и группировка на этапе сжатия» раздела 10.2) здесь неприменимы. Механизмы отказоустойчивости также должны поменяться: в случае пакетной задачи, которая выполняется в течение нескольких минут, неудачную задачу можно просто запустить заново. Но потоковые задачи выполняются в течение нескольких лет, и их перезапуск с начала в случае сбоя невозможен.

Применение обработки потоков

Обработка потоков уже давно служит для мониторинга, когда организации требуется получать предупреждения, если происходят определенные события. Например:

- системы обнаружения мошенничества распознают ситуацию неожиданного изменения схемы использования кредитной карты и блокируют карту, поскольку она, вероятно, была украдена;
- торговые системы изучают изменения цен на финансовом рынке и совершают сделки в соответствии с заданными правилами;
- системы производства контролируют состояние машин на заводе и в случае неисправности быстро распознают проблему;
- военные и разведывательные системы следят за действиями потенциального агрессора и поднимают тревогу, если есть признаки нападения.

Такие приложения требуют довольно сложных шаблонов и системы соответствий. Однако со временем появились и другие способы обработки потоков. В данном разделе мы кратко рассмотрим и сравним некоторые из этих приложений.

Обработка сложных событий

Обработка сложных событий (complex event processing, CEP) — это технология, разработанная в 1990-х годах для анализа потоков событий, особенно для тех приложений, которые требуют поиска событий по определенным шаблонам [65, 66]. Подобно тому как регулярные выражения позволяют находить в строке фрагменты по определенным символьным шаблонам, CEP дает возможность создавать правила для поиска в потоке событий, соответствующих заданным шаблонам.

Для описания шаблонов событий, подлежащих обнаружению, в системах CEP часто применяется декларативный язык запросов высокого уровня, такой как SQL, или графический пользовательский интерфейс. Эти запросы затем передаются в систему обработки, которая потребляет входные потоки, а внутри представляет собой конечный автомат, разыскивающий заданное соответствие. Когда оно найдено, система генерирует *сложное событие* (отсюда и название технологии), содержащее подробное описание обнаруженного шаблона [67].

В этих системах взаимосвязь между запросами и данными противоположна той, что присуща обычным базам, где данные хранятся постоянно, а запросы считаются временными событиями. Когда в базу поступает запрос, она ищет данные, соответствующие запросу, а потом забывает о нем. В системах CEP эти роли меняются: запросы хранятся в течение длительного времени, а события из входных потоков непрерывно протекают мимо них в поисках запроса, соответствующего шаблону события [68].

CEP реализована в таких продуктах, как Esper [69], IBM InfoSphere Streams [70], Apama, TIBCO StreamBase и SQLstream. Распределенные потоковые процессоры, такие как Samza, также поддерживают SQL для декларативных запросов в потоках [71].

Аналитика потоков

Другая область, в которой используется потоковая обработка, — *аналитика* потоков. Граница между CEP и аналитикой потоков размыта, но, как правило, последняя заинтересована не столько в поиске определенных последовательностей событий, сколько в сборе большого количества событий и генерации статистических показателей на его основе. Например:

- измерение частоты событий определенного типа (как часто это происходит в течение заданного времени);
- вычисление скользящего среднего значения за некоторый период времени;
- сравнение текущих статистических показателей с данными за предшествующие временные интервалы (например, для выявления тенденций или предупреждения о необычно высоких или низких показателях по сравнению с тем же временем на прошлой неделе).

Такая статистика обычно вычисляется за фиксированные временные интервалы — например, можно узнать среднее количество запросов сервиса в секунду за последние 5 минут и их 99-процентное время отклика в течение этого периода. Усреднение в течение нескольких минут сглаживает нерелевантные колебания, сохраняя регулярную картину изменений в структуре трафика. Временной интервал, в течение которого выполняется сбор данных, называется *окном,* и его мы подробно рассмотрим в подразделе «Рассуждения о времени» текущего раздела.

В системах потоковой аналитики иногда используются вероятностные алгоритмы, такие как фильтры Bloom (упоминавшиеся в пункте «Оптимизация производительности» подраздела «SS-таблицы и LSM-деревья» раздела 3.1) для определения принадлежности множеству, HyperLogLog [72] для оценки мощности и различные алгоритмы оценки процентилей (см. врезку «Процентили на практике» в подразделе «Описание производительности» раздела 1.3). Вероятностные алгоритмы дают приблизительные результаты, но зато требуют значительно меньше памяти в потоковом процессоре, чем точные алгоритмы. Такое применение аппроксимирующих алгоритмов иногда заставляет людей полагать, что системы потоковой обработки неточны и всегда связаны с потерями данных, но в действительности это не так: в природе потоковой обработки нет ничего приближенного, а вероятностные алгоритмы являются просто оптимизацией [73].

Многие распределенные системы обработки потоков с открытым исходным кодом разработаны с расчетом на аналитику: например, Apache Storm, Spark Streaming, Flink, Concord, Samza, Kafka Streams [74]; есть также хостинговые сервисы, такие как Google Cloud Dataflow и Azure Stream Analytics.

Поддержка материализованных представлений

В разделе 11.2 было показано: поток изменений в базе данных может использоваться для того, чтобы информация в производных информационных системах, таких как кэши, поисковые индексы и склады данных, соответствовала тому, что хранится в исходной базе. Эти примеры можно рассматривать как частные случаи поддержки *материализованных представлений* (см. подраздел «Агрегирование: кубы данных и материализованные представления» раздела 3.3): получение альтернативного представления для некоторого набора данных в целях более удобной обработки запросов и обновления этого представления каждый раз при изменении базовых данных [50].

Аналогичным образом в случае источников событий состояние приложения поддерживается с помощью журнала событий; здесь состояние приложения также является своего рода материализованным представлением. В отличие от сценариев потоковой аналитики здесь, как правило, недостаточно рассматривать только события, произошедшие в рамках некоего временного окна. Для построения материализованного представления потенциально потребуются *все* события за произвольный период времени, за исключением устаревших, которые могут быть отброшены путем уплотнения журнала (см. пункт «Уплотнение журнала» подраздела «Перехват изменений данных» раздела 11.2). По сути, нам нужно окно, простирающееся до самого начала регистрации событий.

В принципе, любой потоковый процессор подходит для поддержки материализованных представлений, хотя необходимость постоянно поддерживать события

противоречит назначению некоторых систем, ориентированных на аналитику, — они в основном работают с окнами ограниченной продолжительности. Samza и Kafka Streams поддерживают этот вид применения, опираясь на реализованные в Kafka возможности сжатия журнала [75].

Поиск в потоках

Системы CEP позволяют находить события, удовлетворяющие заданному шаблону, но иногда требуется и поиск событий по сложным критериям, таким как полнотекстовые поисковые запросы.

Например, сервисы мониторинга СМИ подписываются на каналы новостей и трансляции средств массовой информации и ищут новости, в которых упоминаются компании, продукты или темы, представляющие интерес. Для этого вначале формулируется поисковый запрос, а затем ведется постоянный поиск соответствий элементов из потока новостей по этому запросу. Подобные функции существуют на отдельных сайтах: например, пользователи сайтов, посвященных недвижимости, могут получать уведомления, когда на рынке появляется новый объект, соответствующий их критериям поиска. Примером реализации такого рода поиска в потоке является фильтр в Elasticsearch [76].

Обычные поисковые системы вначале индексируют документы, а затем выполняют запросы по индексу. При поиске в потоке все делается наоборот: запросы хранятся, а документы проходят через фильтр, как, например, в CEP. В простейшем случае можно проверять каждый документ на соответствие каждому запросу, хотя при большом количестве запросов это может занять много времени. Чтобы оптимизировать процесс, можно индексировать запросы и документы и тем самым сузить набор запросов, которые могут соответствовать определенному документу [77].

Передача сообщений и RPC

В подразделе «Поток данных передачи сообщений» раздела 4.2 обсуждались системы передачи сообщений как альтернатива RPC — другими словами, механизмы обслуживания коммуникаций, используемые, например, в акторной модели. Хотя эти системы также основаны на сообщениях и событиях, их не принято считать потоковыми процессорами.

- Акторные фреймворки, прежде всего, механизмы управления конкурентным доступом и распределенным выполнением коммуникационных модулей, тогда как потоковая обработка — это, прежде всего, технология управления данными.
- Коммуникация между акторами часто бывает кратковременной и «один-к-одному», тогда как журналы событий являются долговечными и многопользовательскими.

- Акторы могут обмениваться информацией произвольным образом (включая паттерны циклических запросов/ответов), но потоковые процессоры обычно объединяются в ациклические конвейеры, где каждый поток представляет собой выходные данные конкретной задачи и получен на основе четко определенного набора входных потоков.

Таким образом, существует некая область пересечения между RPC-подобными системами и потоковой обработкой. Например, в Apache Storm есть функция, называемая *распределенным RPC* и позволяющая передавать пользовательские запросы набору узлов, который также обрабатывают потоки событий. Эти запросы затем объединяются с событиями из входных потоков, результаты могут быть агрегированы и отправлены обратно пользователю [78]. (См. вдобавок пункт «Обработка данных, хранящихся в нескольких разделах» подраздела «Наблюдение за производными состояниями» раздела 12.2.)

Кроме того, можно обрабатывать потоки с помощью акторных фреймворков. Тем не менее многие такие фреймворки не гарантируют доставку сообщений в случае сбоев, поэтому обработка не является отказоустойчивой, если только не реализовать дополнительную логику повтора операций.

Рассуждения о времени

Потоковые процессоры часто имеют дело с временем. Особенно когда они используются для аналитики с ее временными окнами вроде «среднего значения за последние 5 минут». Может показаться, что смысл фразы «последние 5 минут» однозначен и ясен, но, к сожалению, это понятие на удивление сложное.

В пакетном процессе задачи обработки быстро проходят по большому набору исторических событий. Если требуется какая-то разбивка по времени, то пакетный процесс ориентируется по временным меткам, встроенным в каждое событие. Нет смысла смотреть на системные часы компьютера, выполняющего пакетный процесс: время выполнения процесса не имеет отношения ко времени возникновения события.

Пакетный процесс может за считаные минуты прочитать исторические события за год; в большинстве случаев временной интервал представляет собой год, а не несколько минут обработки. Более того, использование временных меток в событиях позволяет детерминировать обработку: повторение одного и того же процесса на одном и том же наборе входных данных дает тот же результат (см. пункт «Отказоустойчивость» подраздела «Материализация промежуточного состояния» раздела 10.3).

Во многих потоковых системах обработки, напротив, для определения временного окна используются локальные системные часы вычислительной машины *(время обработки)* [79]. Преимущество такого подхода заключается в его простоте.

Это разумно в случае пренебрежительно малой задержки между созданием и обработкой событий. Однако при сколько-нибудь значительном отставании обработки, то есть если обработка может произойти заметно позже времени возникновения события, все нарушается.

Время наступления и время обработки события

Есть много причин, по которым обработка может задерживаться: очередь, сетевые сбои (см. раздел 8.2), низкая производительность, приводящая к гонке в брокере сообщений или процессоре, перезапуск потребителя потока или повторная обработка прошлых событий (см. пункт «Повторение старых сообщений» подраздела «Секционирование журналов» раздела 11.1) при восстановлении после сбоя или после исправления ошибок в коде.

Задержки сообщений также могут привести к непредсказуемому изменению последовательности сообщений. Например, предположим, что пользователь сначала делает один веб-запрос (обрабатываемый веб-сервером A), а затем второй (обрабатываемый сервером B). Серверы A и B генерируют события, описывающие обрабатываемые ими запросы, но событие B достигает брокера сообщений прежде события A. Теперь потоковые процессоры будут видеть сначала событие B, а затем A, даже если в действительности они произошли в обратном порядке.

В каком-то смысле это похоже на фильмы «Звездные войны»: «Эпизод IV» был выпущен в 1977 году, «Эпизод V» — в 1980-м, «Эпизод VI» — в 1983-м, а затем появились эпизоды I, II и III — в 1999, 2002 и 2005 годах соответственно, а «Эпизод VII» — в 2015 году [80][1]. Если вы смотрели фильмы в том порядке, в котором они были сняты, то порядок вашей обработки этих фильмов не соответствует последовательности повествования. (Номер эпизода похож на временную метку события, а дата просмотра фильм — время обработки.) Будучи людьми, мы способны справиться с такими нарушениями последовательности, но в алгоритмах обработки потока необходимо специально учитывать такие возможности нарушения хронометража и упорядоченности.

Несоответствие времени события и времени обработки приводит к ошибочным данным. Например, предположим, что у нас есть потоковый процессор, который измеряет скорость запросов (подсчитывает количество запросов в секунду). При перезагрузке процессора потока он может на минутку отключиться и затем после включения обрабатывать события с отставанием. Если измерять частоту событий по времени обработки, то результаты будут выглядеть так, будто в момент отставания процессора произошел внезапный аномальный всплеск запросов, в то время как на самом деле их частота не изменялась (рис. 11.7).

---

[1] Авторы благодарят Костаса Клаудаса (Kostas Kloudas) из сообщества Flink за эту аналогию.

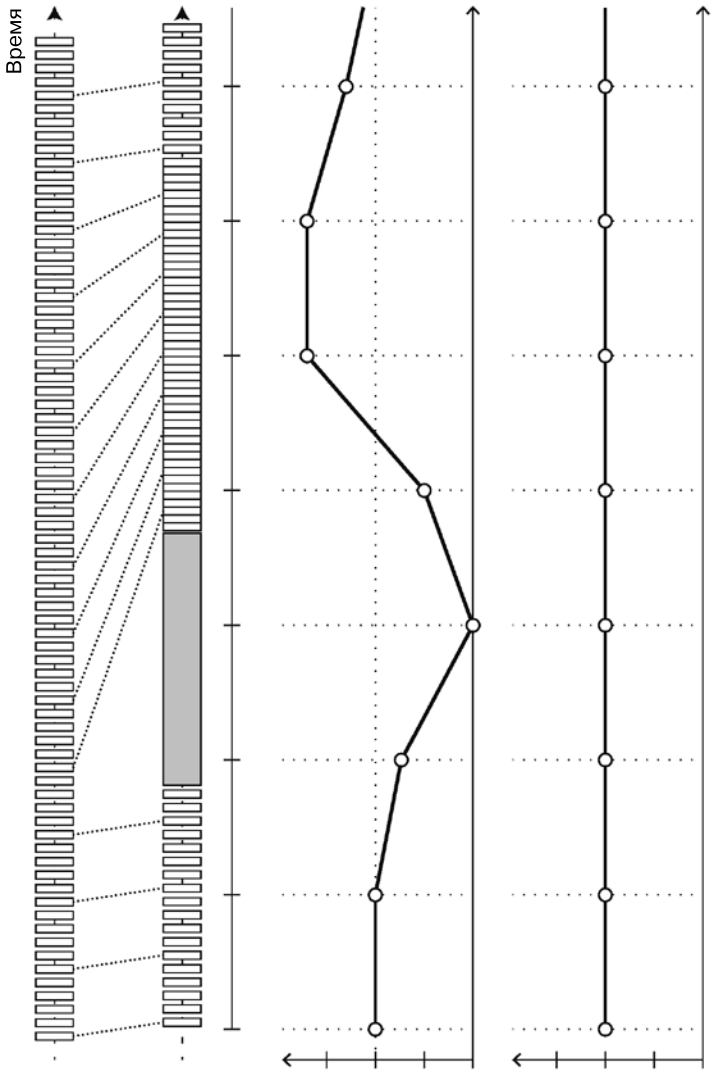


Рис. 11.7. Окно времени, устанавливаемое по времени обработки, вносит ошибки из-за изменений в скорости обработки

Как узнать о готовности

Сложность с определением временных окон с точки зрения времени наступления события заключается в том, что вы никогда не можете быть уверены в получении всех событий для определенного временного окна.

Например, предположим, что мы группируем события по одноминутным интервалам с целью получить количество запросов в минуту. Мы подсчитали некоторое количество событий с отметками времени, соответствующими 37-й минуте часа. Но время идет дальше, и теперь большинство входящих событий приходится на 38-ю и 39-ю минуты часа. Когда мы сможем объявить об окончании обработки окна 37-й минуты и вывести его значение счетчика?

Можно посчитать, что время истекло, и объявить временной интервал законченным после того, как в течение какого-то времени не поступало новых событий. Но может случиться так, что некоторые события задержались из-за сбоя сети и попали в буфер на другой машине. Нужно иметь возможность обрабатывать такие *отставшие* события, появившиеся после объявления временного интервала завершенным. В целом у нас есть два варианта [1].

1. Игнорировать отставшие события, поскольку их доля, скорее всего, в обычных условиях невелика. Можно отслеживать количество отброшенных событий как показатель и выдавать предупреждения, если система начнет отбрасывать значительный объем данных.
2. Опубликовать *поправку*, обновленное значение для данного временного окна с учетом отставших событий. В этом случае также может потребоваться отменить предыдущий вывод.

Иногда можно использовать специальные сообщения вида: «С этого момента сообщения с временной меткой раньше чем t не будут поступать». Потребители могут применять такие сообщения для запуска временных окон [81]. Однако если события генерируют несколько инициаторов на разных машинах, каждый из которых имеет собственные минимальные пороговые значения времени, то потребители должны отслеживать каждого производителя отдельно. Это вызывает сложности при добавлении и удалении инициаторов.

Чьи часы вы используете

Когда события могут буферизоваться в разных точках системы, присвоение им временных меток усложняется. Например, рассмотрим мобильное приложение, которое передает на сервер некие показатели своей работы. Приложение может работать, пока мобильное устройство отключено от Сети. В этом случае оно будет буферизовать события локально и отправит их на сервер, когда появится подключение

к Интернету (вероятно, через несколько часов или даже дней). Для всех потребителей данного потока события будут появляться с большим отставанием.

В этом контексте временная метка событий должна быть действительным временем, в которое произошло данное действие пользователя, по локальным часам мобильного устройства. Однако показания часов на устройстве, управляемом пользователем, часто не считаются достоверными, так как на них случайно или намеренно может быть установлено неправильное время (см. подраздел «Синхронизация часов и их точность» раздела 8.3). Время поступления события на сервер (по часам сервера), скорее всего, будет точным, поскольку он находится под вашим контролем, но менее значимым с точки зрения описания действий пользователя.

Чтобы задействовать недостоверные показания часов устройства, применяется следующий подход с регистрацией трех временных меток [82]:

- время, когда произошло событие, по часам устройства;
- время, когда событие было отправлено на сервер, по часам устройства;
- время, когда событие было получено сервером, по часам сервера.

Вычитая вторую временную метку из третьей, можно оценить смещение между часами устройства и сервера (при условии, что сетевая задержка незначительна по сравнению с требуемой точностью временной метки). Затем можно применить это смещение к временной метке события и таким образом оценить истинное время, в которое произошло событие (если между временем события и временем его отправки на сервер смещение часов устройства не изменилось).

Данная проблема касается не только потоковой обработки — пакетная обработка страдает от тех же проблем, связанных с временем. Однако в контексте потока, где мы более осведомлены о течении времени, это более заметно.

Типы временных окон

Итак, мы знаем, как выяснить временную метку события. Следующим шагом будет решение о том, как определить окно для периода времени. Затем такое окно можно применить для сбора данных, например для подсчета событий или вычисления среднего значения в заданном временном окне. В большинстве случаев используются несколько типов окон [79, 83].

- *«Падающее» окно*. Имеет постоянную длину, и каждое событие принадлежит только одному окну. Например, для одноминутных окон все события с отметками времени между 10:03:00 и 10:03:59 попадают в одно окно, события между 10:04:00 и 10:04:59 — в следующее и т. д. Чтобы реализовать одноминутное «падающее» окно, можно взять временную метку каждого события и, округляя ее до ближайшей минуты, определить, какому окну принадлежит данное событие.

- *«Прыгающее» окно*. Тоже имеет фиксированную длину. Однако такие окна могут перекрываться для обеспечения некоторого сглаживания. Например, 5-минутное окно с размером «прыжка» в 1 минуту будет содержать события между 10:03:00 и 10:07:59, в следующем окне будут отображаться события между 10:04:00 и 10:08:59 и т. д. Чтобы реализовать «прыгающее» окно, нужно сначала вычислить 1-минутные «падающие» окна, а затем объединить несколько соседних окон.
- *Скользящее окно*. Содержит все события, происходящие с некоторыми интервалами между собой. Например, 5-минутное скользящее окно будет охватывать события с 10:03:39 по 10:08:12, поскольку между ними прошло менее 5 минут (обратите внимание, что в случае «падающих» и «прыгающих» 5-минутных окон с их фиксированными границами эти два события не попали бы в одно окно). Скользящее окно может быть реализовано путем хранения буфера событий, отсортированных по времени, и удаления из окна старых событий, когда их время истекает.
- *Окно сессии*. В отличие от других типов у окна сессии нет фиксированной продолжительности. Вместо этого оно определяется как множество всех событий для одного и того же пользователя, которые происходят близко друг к другу во времени. Окно заканчивается, когда пользователь какое-то время неактивен (например, если событий не поступало в течение 30 минут). Разделение на сессии является обычным требованием при анализе сайта (см. пункт «GROUP BY» подраздела «Объединение и группировка на этапе сжатия» раздела 10.2).

Объединения потоков

В главе 10 было показано, как в пакетных задачах можно использовать объединения наборов данных по ключу и что такие объединения являются важной частью конвейеров обработки информации. Поскольку потоковая обработка — это обобщение информационных конвейеров для инкрементной обработки неограниченных наборов данных, существует такая же потребность в объединениях потоков.

Однако тот факт, что новые события могут появляться в потоке в любое время, делает объединение потоков более сложным, чем в пакетных задачах. Для лучшего понимания ситуации мы выделим три типа объединений: *поток — поток*, *поток — таблица* и *таблица — таблица* [84]. Ниже мы рассмотрим каждое из них на примере.

Объединение «поток — поток» (объединение окон)

Предположим, у нас есть функция поиска на сайте и мы хотим отслеживать последние тенденции в поиске URL. Каждый раз, когда кто-то вводит поисковый запрос, мы регистрируем событие, содержащее запрос и его результаты. Всякий раз при чьем-то нажатии одного из результатов поиска мы регистрируем другое событие,

записывая переход по ссылке. Чтобы рассчитать процент переходов по ссылкам для каждого URL в результатах поиска, необходимо объединить события поиска и перехода по ссылке, связанные одинаковым ID сессии. Подобный анализ часто используется в рекламных системах [85].

В случае отказа пользователя от результатов поиска нажатия ссылки может и не быть. И даже если оно произойдет, время между поисковым запросом и нажатием может быть очень разным: как правило, это несколько секунд, а иногда и несколько дней или недель (например, пользователь запускает поиск, забывает об этой вкладке браузера, а затем возвращается к ней и переходит по ссылке). Из-за различных задержек сети событие нажатия ссылки может поступить на обработку даже раньше, чем событие поиска. Можно выбрать для объединения подходящее окно, например, объединить нажатие с поиском, если между ними прошло не более часа.

Обратите внимание: встраивание сведений о результатах поиска в событие нажатия — не то же самое, что объединение событий, так вы узнаете только о тех случаях, когда пользователь нажал результат поиска, но не о запросах, в которых он не нажал ни один из результатов. Чтобы измерить качество поиска, нужна точная частота нажатий, а для этого требуются события как поиска, так и нажатий.

Чтобы реализовать этот тип объединения, поточный процессор должен поддерживать *состояния*, например все события, произошедшие за последний час и индексированные по ID сессии. Всякий раз, когда происходит событие поиска или нажатия, оно добавляется к соответствующему индексу. Потоковый процессор каждый раз проверяет оба индекса с целью узнать, не получено ли уже другое событие для того же ID сессии. При поступлении соответствующего события процессор генерирует событие, в котором указывается, какой результат поиска был выбран. Если событие поиска истекло, а соответствующее событие нажатия не было получено, то генерируется событие, в котором сообщается, что пользователь не выбрал ни один из результатов поиска.

Объединения «поток — таблица» (обогащение потока)

В пункте «Пример: анализ активности пользователя» подраздела «Объединение и группировка на этапе сжатия» раздела 10.2 был показан пример пакетной задачи, объединяющей два набора данных: набор событий активности пользователя и БД пользовательских профилей (см. рис. 10.2). Естественно представлять события активности как поток и постоянно выполнять в потоковом процессоре одно и то же объединение: на входе поток событий активности, содержащих ID пользователя, а на выходе — поток событий активности, в которых идентификатор дополнен информацией о его профиле. Этот процесс иногда называют *обогащением* событий активности информацией из базы данных.

Чтобы выполнить это объединение, потоковый процесс должен просматривать события активности по одному, находить ID пользователя в базе данных и добавлять в событие активности информацию о профиле этого человека. Поиск в БД может быть реализован путем запроса удаленной базы; однако, как описано в пункте «Пример: анализ активности пользователя» подраздела «Объединение и группировка на этапе сжатия» раздела 10.2, такие удаленные запросы, скорее всего, будут медленными и могут вызвать перегрузку базы данных [75].

Другой подход заключается в том, чтобы загрузить в потоковый процессор копию базы и запрашивать ее локально, без передачи данных по сети. Это очень похоже на хеш-объединения, описанные в подразделе «Объединения на этапе сопоставления» раздела 10.2: локальная копия БД может храниться в памяти в виде хеш-таблицы, если она достаточно мала, или же в виде индекса на локальном диске.

В отличие от пакетных задач, где используется моментальный снимок базы данных на момент ввода исходных данных, потоковый процессор работает долгое время, и содержимое базы, очевидно, со временем изменится. В то время локальная копия БД потокового процессора должна обновиться. Эту проблему можно решить путем сбора данных об изменениях: потоковый процессор может подписаться на журнал изменений пользовательских профилей в базе данных, а также на поток событий активности. При создании или изменении профиля потоковый процессор обновит свою локальную копию базы данных. Таким образом, мы получаем объединение двух потоков: событий активности и обновлений профиля.

На практике объединение «поток — таблица» очень похоже на объединение «поток — поток»; главное различие заключается в том, что для потока изменений таблицы объединение использует окно, начало которого совпадает с «началом времени» (концептуально бесконечное окно), причем новые версии записей перезаписывают старые. Для входных данных потока объединение может вообще не поддерживать окно.

Объединения «таблица — таблица» (материализованная поддержка представлений)

Рассмотрим пример ленты сообщений Twitter, представленный в подразделе «Описание нагрузки» раздела 1.3. Там отмечалось, что, когда пользователь хочет просмотреть свою ленту сообщений, слишком накладно перебирать всех людей, которых он отслеживает, находить их последние твиты и объединять их.

Лучше применять кэш ленты сообщений: вариант папки Входящие для каждого пользователя, в которую записываются твиты по мере их отправки, вследствие чего чтение ленты сообщений выполняется за один просмотр. Для реализации и поддержки такого кэша нужна следующая обработка событий:

- если пользователь *u* отправляет новый твит, то он добавляется в ленту сообщений каждого подписчика *u*;

- если пользователь удаляет твит, то он удаляется из лент сообщений всех пользователей;
- если пользователь *u*1 начинает отслеживать сообщения пользователя *u*2, последние твиты *u*2 добавляются в ленту сообщений *u*1;
- если пользователь *u*1 отписывается от обновлений пользователя *u2*, то твиты последнего удаляются из ленты сообщений *u*1.

Для реализации такого обслуживания кэша в потоковом процессоре нужны потоки событий для твитов (отправка и удаление) и отношения подписки (отслеживание и устранение). Процесс потока должен поддерживать базу данных, содержащую набор подписчиков для каждого пользователя, чтобы знать, какие ленты сообщений необходимо обновлять при поступлении нового твита [86].

При взгляде на этот процесс потока с другой стороны видно, что он поддерживает материализованное представление запроса, объединяющего две таблицы (твиты и подписки), примерно так:

```
SELECT follows.follower_id AS timeline_id,
  array_agg(tweets.* ORDER BY tweets.timestamp DESC)
FROM tweets
JOIN follows ON follows.followee_id = tweets.sender_id
GROUP BY follows.follower_id
```

Объединение потоков прямо соответствует объединению таблиц в данном запросе. Ленты сообщений — фактически кэш результата этого запроса, который обновляется всякий раз при изменении соответствующих таблиц[1].

Зависимость объединений от времени

Описанные здесь три типа объединений («поток — поток», «поток — таблица» и «таблица — таблица») имеют много общего: все они требуют, чтобы процессор потока поддерживал то или иное состояние (события поиска и нажатия, профили пользователей или лента сообщений), основанное на одном наборе входных данных объединения, и запрос, который указывает на сообщения из другого набора входных данных объединения.

Важна также последовательность событий, поддерживающих состояние (имеет значение, что произошло раньше: вы подписались на ленту сообщений, а затем отменили подписку или наоборот). В секционированном журнале сохраняется

[1] Если рассматривать поток как производную от таблицы (см. рис. 11.6), а объединение — как произведение двух таблиц $u \cdot v$, то получается интересная вещь: для потока изменений материализованного объединения выполняется правило произведения — $(u \cdot v)' = u'v + uv'$. Другими словами, любое изменение твитов объединяется с текущим множеством подписчиков и любое изменение подписчиков объединяется с текущими твитами [49, 50].

порядок событий внутри одной секции, но, как правило, нет гарантии сохранения этого порядка в разных потоках или секциях.

Возникает вопрос: если события в разных потоках происходят примерно в одно и то же время, то в каком порядке они обрабатываются? Возьмем пример объединения «поток — таблица». Пользователь обновит свой профиль; какие события активности объединятся со старым профилем (будут обработаны до обновления профиля), а какие — с новым (будут обработаны после обновления профиля)? Другими словами: если мы делаем объединение с каким-то состоянием и со временем это состояние изменится, то какой момент времени надо задействовать для объединения [45]?

Такая временная зависимость возникает во многих случаях. Например, при продаже товаров необходимо записать в счет правильный налог, зависящий от страны или штата, типа продукта и даты продажи (поскольку время от времени размер налогов меняется). При объединении продажи с таблицей налоговых ставок, очевидно, желательно сделать объединение со значением налога на момент продажи, который может отличаться от текущего значения налога при повторной обработке исторических данных.

Если порядок событий в потоке не определен, то объединение становится недетерминированным [87]. Это значит, что невозможно повторно запустить задачу на том же наборе входных данных и получить тот же результат: при повторном запуске задачи события во входных потоках могут следовать в другом порядке.

В складах данных эта проблема известна как *медленно меняющееся измерение* (slowly changing dimension, SCD). Ее часто решают с помощью уникального идентификатора для каждой версии в объединенной записи: например, всякий раз при изменении размера налога ей предоставляется новый ID, а в счет включается идентификатор налога на момент продажи [88, 89]. Такое дополнение делает объединение детерминированным, но приводит к невозможности уплотнения журнала, поскольку в таблице необходимо сохранять все версии записей.

Отказоустойчивость

В заключительном подразделе мы рассмотрим, как потоковые процессоры справляются с ошибками. В главе 10 мы увидели, что системы пакетной обработки довольно легко исправляют сбои: если операция в задаче MapReduce завершается неудачно, то ее можно просто перезапустить на другой машине, а выходные данные после неудачного завершения будут отброшены. Такое прозрачное повторение возможно, поскольку входные файлы неизменяемы, каждая задача записывает свои выходные данные в отдельный файл на HDFS, и они становятся видимыми только при успешном завершении задачи.

В частности, пакетный подход к отказоустойчивости гарантирует: результат выполнения пакетного задания будет таким же, как при отсутствии ошибки, даже

если на самом деле отдельные задачи завершились неуспешно. Кажется, что каждая входная запись обрабатывалась ровно один раз — ни одна запись не пропускается и не обрабатывается дважды. Хотя перезапуск задач говорит о возможности повторной обработки некоторых записей, видимый результат на выходе выглядит так, как если бы все они обрабатывались только один раз. Этот принцип известен под именем *«семантика "выполнение один раз"»*, хотя правильнее было бы назвать его *семантикой «кажется, что только один раз»* [90].

Такая же проблема отказоустойчивости возникает и при потоковой обработке, но здесь с ней сложнее справиться: нельзя дождаться, пока задача будет закончена, и только потом сделать свой вывод видимым. Поток бесконечен, и его обработка никогда не закончится.

Микропакеты и контрольные точки

Одно из решений этой проблемы — разбить поток на небольшие блоки и рассматривать каждый из них как миниатюрный пакетный процесс. Такой подход называется *микропакетами* и используется в Spark Streaming [91]. Размер пакета обычно составляет примерно 1 секунду, это компромисс производительности: пакеты меньшего размера требуют больше затрат на планирование и координацию, в то время как увеличение размера пакета означает более длительную задержку, прежде чем можно будет вывести результаты обработки потока.

В технологии микропакетов неявно реализовано «падающее» окно, равное размеру пакета (определяемое не только временными метками событий, но и временем обработки); любые задания, требующие окон большего размера, должны явно переносить состояние из одного микропакета в другой.

Вариант этой технологии, реализованный в Apache Flink, периодически генерирует контрольные точки состояния и записывает их в долговременное хранилище [92, 93]. Если оператор потока выходит из строя, то может перезапуститься с последней контрольной точки, отбросив все результаты, сгенерированные между последней контрольной точкой и моментом сбоя. Контрольные точки активизируются барьерами в потоке сообщений аналогично границам между микропакетами, но без требования определенного размера окон.

В системах потоковой обработки технологии микропакетов и контрольных точек обеспечивают одну и ту же семантику «выполнение один раз», как и в случае пакетной обработки. Однако как только данные выходят из процессора потока (например, записываются в БД, отправляются в виде сообщения внешнему брокеру сообщений или в виде письма по электронной почте), система больше не может отбросить выходные данные неудачного пакета. В таком случае перезапуск неудавшейся задачи приводит к тому, что внешний побочный эффект случается дважды. Для предотвращения этой проблемы одних только микропакетов и контрольных точек недостаточно.

Подтверждение малых изменений

Чтобы в случае сбоя повторно обрабатывать данные только один раз, необходимо гарантировать следующее: все выходные данные и побочные эффекты обработки события вступают в силу *тогда и только тогда*, когда обработка прошла успешно. Побочные эффекты — это любые сообщения, отправленные следующим операторам или внешним системам обмена сообщениями (включая уведомления по электронной почте или уведомления без запроса), записи в базу, изменения состояния оператора и подтверждения ввода сообщений (в том числе увеличение смещения потребителя в брокере сообщений на основе журналирования).

Все указанные действия либо должны входить в состав атомарной операции, либо ни одно из них не должно произойти, но им не следует рассинхронизироваться. Если вам данный подход кажется знакомым, то потому, что мы обсуждали его в пункте «Однократная обработка сообщений» подраздела «Распределенные транзакции на практике» раздела 9.4 в контексте распределенных транзакций и двухфазной фиксации.

В главе 9 обсуждались проблемы традиционных реализаций распределенных транзакций, таких как XA. Однако в более ограниченных системах механизм атомарной фиксации может оказаться эффективным. Именно такой подход реализован в Google Cloud Dataflow [81, 92] и VoltDB [94], и есть планы добавить аналогичные функции в Apache Kafka [95, 96]. В отличие от XA, в этих реализациях не предпринимается попыток проводить транзакции через гетерогенные технологии, вместо этого они остаются внутренними и управляют как изменениями состояния, так и обменом сообщениями в рамках системы потоковой обработки. Вычислительные затраты на реализацию протокола транзакции могут быть амортизированы путем обработки нескольких входных сообщений в рамках одной транзакции.

Идемпотентность

Наша цель — отказаться от частичного вывода неудачных задач, чтобы их можно было безопасно повторить, не получая дважды одни и те же удачные данные. Распределенные транзакции — один из способов достижения этой цели, но есть и другой — положиться на *идемпотентность* [97].

Идемпотентная операция — это такая операция, которую можно выполнять несколько раз, и результат будет таким же, как если бы ее выполнили только один раз. Например, присвоение значения ключу в хранилище типа «ключ — значение» для того или иного фиксированного значения является идемпотентным (при повторной записи значения оно просто перезаписывается и остается неизменным), в отличие от приращения счетчика (повторное приращение означает, что значение будет увеличено дважды).

Даже если операция по своей природе не является идемпотентной, ее часто можно сделать таковой, задействовав небольшое количество дополнительных метаданных. Например, в Kafka каждое сообщение имеет постоянное монотонно увеличивающееся смещение. При записи значений во внешнюю базу данных можно заносить туда смещение сообщения, которое вызвало последнюю операцию записи. Таким образом можно будет узнать, было ли обновление уже применено, и не повторять его дважды.

Аналогичная идея лежит в основе управления состоянием в Trident Storm [78]. Исходя из идемпотентности, делается несколько предположений: перезапуск неудачно завершенной задачи должен воспроизводить одни и те же сообщения в одном и том же порядке (здесь действует брокер на основе журналирования), обработка должна быть детерминированной, и ни один узел не может в то же время обновлять это же значение [98, 99].

Если сбой распространяется от одного узла к другому, то может потребоваться ограждение (см. пункт «Ведущий узел и блокировки» подраздела «Истина определяется большинством» раздела 8.4), чтобы предотвратить вмешательство узла, который считается отключенным, но на самом деле действует. Все же, несмотря на эти оговорки, идемпотентные операции могут быть эффективным способом достижения семантики «выполнение один раз» с небольшими вычислительными затратами.

Восстановление состояния после сбоя

Любой потоковый процесс, требующий состояния — например, оконные агрегирования (такие как счетчики, вычисления средних значений, гистограммы), таблицы и индексы, используемые для объединений, — должен гарантировать, что это состояние может быть восстановлено после сбоя.

Один из вариантов заключается в том, чтобы сохранить состояние в удаленном хранилище данных и реплицировать его. Однако обращение к удаленной базе данных для каждого сообщения способно замедлить систему (см. пункт «Объединения “поток — таблица” (обогащение потока)» подраздела «Объединения потоков» текущего раздела). Вместо этого можно сохранять состояние локально в потоковом процессоре и периодически реплицировать его. Тогда в случае сбоя потокового процессора при восстановлении новая задача прочитает реплицированное состояние и возобновит обработку без потери данных.

Например, Flink периодически делает снимки состояния оператора и записывает их в надежное хранилище, такое как HDFS [92, 93]; Samza и Kafka Streams реплицируют изменения состояния, отправляя их в специальную тему Kafka с уплотнением журнала, аналогичную снимку изменений данных [84, 100]. VoltDB реплицирует состояние путем избыточной обработки каждого входного сообщения на нескольких узлах (см. подраздел «По-настоящему последовательное выполнение» раздела 7.3).

В некоторых случаях в репликации состояния даже нет нужды, поскольку оно может быть восстановлено из входных потоков. Например, при условии, что состояние состоит из операций агрегации в достаточно коротком окне, будет быстрее просто воспроизвести входные события, соответствующие этому окну. Если состояние является локальной репликой БД, поддерживаемой методом захвата изменений данных, то база также может быть восстановлена из потока с уплотненным журналом изменений (см. пункт «Уплотнение журнала» подраздела «Перехват изменений данных» раздела 11.2).

Однако все эти компромиссы зависят от производительности базовой инфраструктуры: в некоторых системах сетевая задержка может быть ниже, чем задержка доступа к диску, а пропускная способность сети может быть сопоставима с пропускной способностью диска. Не существует идеального, универсального компромисса для всех ситуаций. Кроме того, преимущества местного и удаленного хранения состояния могут меняться по мере развития способов хранения информации и сетевых технологий.

11.4. Резюме

В этой главе были рассмотрены потоки событий — для каких целей они применяются и как их обрабатывать. В некоторых отношениях потоковая обработка очень похожа на пакетную, представленную в главе 10, с той разницей, что потоковая выполняется непрерывно и не на наборе входных данных фиксированного размера, а для неограниченных (бесконечных) потоков. С этой точки зрения брокеры сообщений и журналы событий служат потоковым эквивалентом файловой системы.

Мы уделили некоторое время сравнению следующих двух типов брокерских сообщений.

- *Брокеры сообщений типа AMKP/JMS*. Брокер отправляет сообщения потребителям, а те высылают подтверждение каждый раз, когда сообщение успешно обработано. После получения подтверждения удаляются из брокера сообщения. Такая технология подходит в качестве асинхронной формы RPC (см. также подраздел «Поток данных передачи сообщений» раздела 4.2) — например, в очереди задач, где точный порядок обработки сообщений неважен и нет необходимости возвращаться и заново читать старые сообщения после их обработки.
- *Брокер сообщений на основе журналирования*. Брокер отправляет все сообщения из данного раздела одному узлу-потребителю и всегда в одном и том же порядке. Параллелизм достигается путем секционирования, а потребители отслеживают прогресс, проверяя смещение последнего обработанного сообщения. Брокер сохраняет сообщения на диске, поэтому при необходимости можно вернуться и прочитать старые сообщения заново.

Применение журналов в потоковой обработке похоже на использование журналов репликации в базах данных (см. главу 5) и системы хранения информации на основе журналирования (см. главу 3). Как видим, эта технология особенно подходит для систем потоковой обработки, которые потребляют входные потоки и генерируют производные состояния или выходные потоки.

В отношении потоков мы обсудили несколько возможностей: так, события активности пользователей, периодически считываемые показания датчиков и каналы данных (например, информация о рынках в финансовой сфере) естественно представить в виде потоков. Мы также увидели, что полезно осуществлять запись в базу данных в виде потока: можно фиксировать историю всех изменений, сделанных в базе, — либо неявно, путем захвата данных об изменениях, либо явно, через источники событий. Уплотнение журнала позволяет потоку сохранять полную копию содержимого БД.

Представление баз данных в виде потоков открывает широкие возможности для интеграции систем. Можно постоянно поддерживать актуальность производных информационных систем, таких как индексы поиска, кэши и аналитические системы, потребляя журнал изменений и применяя их к производной системе. Можно даже создавать новые представления для уже существующих данных, потребляя журнал изменений с самого начала вплоть до настоящего времени.

Средства для поддержания состояния в виде потоков и повторения сообщений легли в основу технологий, которые позволяют объединять потоки и обеспечивают отказоустойчивость в различных системах потоковой обработки. Мы рассмотрели несколько применений такой обработки, в том числе поиск событий по шаблону (сложная обработка событий), вычисление оконных агрегаций (анализ потоков) и поддержание производных информационных систем в актуальном состоянии (материализованные представления).

Затем мы обсудили трудности, возникающие при определении временных интервалов в потоковых процессорах, в том числе рассмотрели различия между временем обработки и временными метками событий. Кроме того, изучили проблему взаимодействия с событиями, которые появляются после того, как временное окно было закрыто.

Мы выделили три типа объединений, встречающихся в потоковых процессах.

- *Объединения «поток — поток»*. Оба входных потока состоят из событий активности. Оператор объединения ищет связанные события, произошедшие в течение некоего времени. Например, это могут быть два действия одного и того же пользователя, интервал между которыми не превышает 30 минут. Если нужно найти связанные события в одном и том же потоке, то оба набора входных данных объединения могут принадлежать одному потоку *(самообъединение)*.

- *Объединение «поток — таблица»*. Один входной поток состоит из событий активности, а второй представляет собой журнал изменений базы данных. В журнале хранится локальная копия БД. Для каждого события активности оператор объединения делает запрос в базу и выводит событие активности, обогащенное информацией из БД.
- *Объединение «таблица — таблица»*. Оба входных потока являются изменениями базы данных. В этом случае каждое изменение из одной таблицы объединяется с последним состоянием из другой. Результатом является поток изменений, который передается в материализованное представление объединения двух таблиц.

Наконец, мы обсудили методы обеспечения отказоустойчивости и семантики «выполнение один раз» в потоковом процессоре. Как и при пакетной обработке, здесь нужно отбрасывать частичный вывод неудачно завершенных задач. Однако, поскольку выполнение потока длительное и результат выводится непрерывно, нельзя просто отказаться от всех выходных данных. Вместо этого применяется более дробный механизм восстановления, основанный на микропакетах, контрольных точках, транзакциях и идемпотентных операциях записи.

11.5. Библиография

1. *Akidau T., Bradshaw R., Chambers C., et al.* The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing // Proceedings of the VLDB Endowment, volume 8, number 12, pages 1792–1803, August 2015 [Электронный ресурс]. — Режим доступа: http://www.vldb.org/pvldb/vol8/p1792-Akidau.pdf.
2. *Abelson H., Sussman G. J., Sussman J.* Structure and Interpretation of Computer Programs, 2nd edition. — MIT Press, 1996. https://mitpress.mit.edu/sicp/.
3. *Eugster P. Th., Felber P. A., Guerraoui R., Kermarrec A.-M.* The Many Faces of Publish/Subscribe // ACM Computing Surveys, volume 35, number 2, pages 114–131, June 2003 [Электронный ресурс]. — Режим доступа: http://www.cs.ru.nl/~pieter/oss/manyfaces.pdf.
4. *Hellerstein J. M., Stonebraker M.* Readings in Database Systems, 4th edition. — MIT Press, 2005. http://redbook.cs.berkeley.edu/.
5. *Carney D., Çetintemel U., Cherniack M., et al.* Monitoring Streams — A New Class of Data Management Applications // 28th International Conference on Very Large Data Bases (VLDB), August 2002 [Электронный ресурс]. — Режим доступа: http://www.vldb.org/conf/2002/S07P02.pdf.
6. *Sackman M.* Pushing Back. May 5, 2016 [Электронный ресурс]. — Режим доступа: http://tech.labs.oliverwyman.com/blog/2016/05/05/pushing-back/.

7. *Martí V.* Brubeck, a statsd-Compatible Metrics Aggregator. June 15, 2015 [Электронный ресурс]. — Режим доступа: https://githubengineering.com/brubeck/.
8. *Lowenberger S.* MoldUDP64 Protocol Specification V 1.00. July 2009 [Электронный ресурс]. — Режим доступа: http://www.nasdaqtrader.com/content/technicalsupport/specifications/dataproducts/moldudp64.pdf.
9. *Hintjens P.* ZeroMQ — The Guide. — O'Reilly Media, 2013. http://zguide.zeromq.org/page:all.
10. *Malpass I.* Measure Anything, Measure Everything. February 15, 2011 [Электронный ресурс]. — Режим доступа: https://codeascraft.com/2011/02/15/measure-anything-measure-everything/.
11. *Plaetinck D.* 25 Graphite, Grafana and statsd Gotchas. March 3, 2016 [Электронный ресурс]. — Режим доступа: https://grafana.com/blog/2016/03/03/25-graphite-grafana-and-statsd-gotchas/.
12. *Lindsay J.* Web Hooks to Revolutionize the Web. May 3, 2007 [Электронный ресурс]. — Режим доступа: http://progrium.com/blog/2007/05/03/web-hooks-to-revolutionize-the-web/.
13. *Gray J. N.* Queues Are Databases // Microsoft Research Technical Report MSR-TR-95-56, December 1995 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/queues-are-databases/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F69641%2Ftr-95-56.pdf.
14. *Hapner M., Burridge R., Sharma R., et al.* JSR-343 Java Message Service (JMS) 2.0 Specification. March 2013 [Электронный ресурс]. — Режим доступа: https://jcp.org/en/jsr/detail?id=343.
15. *Aiyagari S., Arrott M., Atwell M., et al.* AMQP: Advanced Message Queuing Protocol Specification // Version 0-9-1, November 2008 [Электронный ресурс]. — Режим доступа: http://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf.
16. Google Cloud Pub/Sub: A Google-Scale Messaging Service. 2016 [Электронный ресурс]. — Режим доступа: https://cloud.google.com/pubsub/architecture.
17. Apache Kafka 0.9 Documentation. November 2015 [Электронный ресурс]. — Режим доступа: http://kafka.apache.org/documentation.html.
18. *Kreps J., Narkhede N., Rao J.* Kafka: A Distributed Messaging System for Log Processing // 6th International Workshop on Networking Meets Databases (NetDB), June 2011 [Электронный ресурс]. — Режим доступа: http://www.longyu23.com/doc/Kafka.pdf.
19. Amazon Kinesis Streams Developer Guide. April 2016 [Электронный ресурс]. — Режим доступа: http://docs.aws.amazon.com/streams/latest/dev/introduction.html.
20. *Stewart L., Guo S.* Building DistributedLog: Twitter's High Performance Replicated Log Service. September 16, 2015 [Электронный ресурс]. — Режим доступа: https://blog.twitter.com/engineering/en_us/topics/infrastructure/2015/building-distributedlog-twitter-s-high-performance-replicated-log-servic.html.

21. DistributedLog Documentation // Twitter, Inc., May 2016 [Электронный ресурс]. — Режим доступа: https://bookkeeper.apache.org/distributedlog/.
22. *Kreps J.* Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). April 27, 2014 [Электронный ресурс]. — Режим доступа: https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines.
23. *Paramasivam K.* How We're Improving and Advancing Kafka at LinkedIn. September 2, 2015 [Электронный ресурс]. — Режим доступа: https://engineering.linkedin.com/apache-kafka/how-we_re-improving-and-advancing-kafka-linkedin.
24. *Kreps J.* The Log: What Every Software Engineer Should Know About Real Time Data's Unifying Abstraction. December 16, 2013 [Электронный ресурс]. — Режим доступа: https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying.
25. *Das S., Botev C., Surlaker K., et al.* All Aboard the Databus! // 3rd ACM Symposium on Cloud Computing (SoCC), October 2012 [Электронный ресурс]. — Режим доступа: https://915bbc94-a-62cb3a1a-s-sites.googlegroups.com/site/acm2012socc/s18-das.pdf?attachauth=ANoY7crfzxKAsTdGRJPaTjXa5eBBqFLDux6gZV4tPY4VMD4E_5_8Gly_7vWryTNLY1fP9zRwcj_1ufTNgGjm6ZD2sRYbgXSuNCACRIN_D37xI-chEjExG42iO6zX29IF1anyiSfDyaq3oWUfJ4Sc6xp4hbbGWG6E8cCnFgGC9DN_KZH95S6fdqeBCu6hY3knAQSI5t4kt-iTeKp7JwcK6Bf9zUN4518lkQ%3D%3D&attredirects=0.
26. *Sharma Y., Ajoux P., Ang P., et al.* Wormhole: Reliable Pub-Sub to Support Geo-Replicated Internet Services // 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI), May 2015 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-sharma.pdf.
27. *Narayan P. P. S.* Sherpa Update. June 8, 2010 [Электронный ресурс]. — Режим доступа: http://web.archive.org/web/20160801221400/https://developer.yahoo.com/blogs/ydn/sherpa-7992.html.
28. *Kleppmann M.* Bottled Water: Real-Time Integration of PostgreSQL and Kafka. April 23, 2015 [Электронный ресурс]. — Режим доступа: http://martin.kleppmann.com/2015/04/23/bottled-water-real-time-postgresql-kafka.html.
29. *Osheroff B.* Introducing Maxwell, a mysql-to-kafka Binlog Processor. August 20, 2015 [Электронный ресурс]. — Режим доступа: https://developerblog.zendesk.com/.
30. *Hauch R.* Debezium 0.2.1 Released. June 10, 2016 [Электронный ресурс]. — Режим доступа: http://debezium.io/blog/2016/06/10/Debezium-0/.
31. *Shankar P. S. U.* Streaming MySQL Tables in Real-Time to Kafka. August 1, 2016 [Электронный ресурс]. — Режим доступа: https://engineeringblog.yelp.com/2016/08/streaming-mysql-tables-in-real-time-to-kafka.html.
32. Mongoriver // Stripe, Inc., September 2014 [Электронный ресурс]. — Режим доступа: https://github.com/stripe/mongoriver.

33. *Harvey D.* Change Data Capture with Mongo + Kafka // Hadoop Users Group UK, August 2015 [Электронный ресурс]. — Режим доступа: https://www.slideshare.net/danharvey/change-data-capture-with-mongodb-and-kafka.
34. Oracle GoldenGate 12c: Real-Time Access to Real-Time Information // Oracle White Paper, March 2015 [Электронный ресурс]. — Режим доступа: http://www.oracle.com/us/products/middleware/data-integration/oracle-goldengate-realtime-access-2031152.pdf.
35. Oracle GoldenGate Fundamentals: How Oracle GoldenGate Works // Oracle Corporation, November 2012 [Электронный ресурс]. — Режим доступа: https://www.youtube.com/watch?v=6H9NibIiPQE.
36. *Akhmechet S.* Advancing the Realtime Web. January 27, 2015 [Электронный ресурс]. — Режим доступа: https://rethinkdb.com/blog/realtime-web/.
37. Firebase Realtime Database Documentation // Google, Inc., May 2016 [Электронный ресурс]. — Режим доступа: https://firebase.google.com/docs/database/.
38. Apache CouchDB 1.6 Documentation. 2014 [Электронный ресурс]. — Режим доступа: http://docs.couchdb.org/en/latest/.
39. *DeBergalis M.* Meteor 0.7.0: Scalable Database Queries Using MongoDB Oplog Instead of Poll-and-Diff. December 17, 2013 [Электронный ресурс]. — Режим доступа: https://blog.meteor.com/meteor-0-7-0-scalable-database-queries-using-mongodb-oplog-instead-of-poll-and-diff-7c4166441ab8.
40. Chapter 15. Importing and Exporting Live Data // VoltDB 6.4 User Manual, June 2016 [Электронный ресурс]. — Режим доступа: https://docs.voltdb.com/UsingVoltDB/ChapExport.php.
41. *Narkhede N.* Announcing Kafka Connect: Building Large-Scale Low Latency Data Pipelines. February 18, 2016 [Электронный ресурс]. — Режим доступа: https://www.confluent.io/blog/announcing-kafka-connect-building-large-scale-low-latency-data-pipelines/.
42. *Young G.* CQRS and Event Sourcing // Code on the Beach, August 2014 [Электронный ресурс]. — Режим доступа: https://www.youtube.com/watch?v=JHGkaShoyNs.
43. *Fowler M.* Event Sourcing. December 12, 2005 [Электронный ресурс]. — Режим доступа: https://martinfowler.com/eaaDev/EventSourcing.html.
44. *Vernon V.* Implementing Domain-Driven Design. — Addison-Wesley Professional, 2013. https://vaughnvernon.co/?page_id=168.
45. *Jagadish H. V., Mumick I. S., Silberschatz A.* View Maintenance Issues for the Chronicle Data Model // 14th ACM SIGACT-SIGMODSIGART Symposium on Principles of Database Systems (PODS), May 1995 [Электронный ресурс]. — Режим доступа: http://www.mathcs.emory.edu/~cheung/papers/StreamDB/Histogram/1995-Jagadish-Histo.pdf.

46. Event Store 3.5.0 Documentation // Event Store LLP, February 2016 [Электронный ресурс]. — Режим доступа: https://eventstore.org/docs/.
47. *Kleppmann M.* Making Sense of Stream Processing. Report, O'Reilly Media, May 2016 [Электронный ресурс]. — Режим доступа: http://www.oreilly.com/data/free/stream-processing.csp.
48. *Mak S.* Event-Sourced Architectures with Akka // JavaOne, September 2014 [Электронный ресурс]. — Режим доступа: https://www.slideshare.net/SanderMak/eventsourced-architectures-with-akka.
49. *Hyde J.* Personal communication. June 2016 [Электронный ресурс]. — Режим доступа: https://twitter.com/julianhyde/status/743374145006641153.
50. *Gupta A., Mumick I. S.* Materialized Views: Techniques, Implementations, and Applications. — MIT Press, 1999.
51. *Griffin T., Libkin L.* Incremental Maintenance of Views with Duplicates // ACM International Conference on Management of Data (SIGMOD), May 1995 [Электронный ресурс]. — Режим доступа: http://homepages.inf.ed.ac.uk/libkin/papers/sigmod95.pdf.
52. *Helland P.* Immutability Changes Everything // 7th Biennial Conference on Innovative Data Systems Research (CIDR), January 2015 [Электронный ресурс]. — Режим доступа: http://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf.
53. *Kleppmann M.* Accounting for Computer Scientists. March 7, 2011 [Электронный ресурс]. — Режим доступа: http://martin.kleppmann.com/2011/03/07/accounting-for-computer-scientists.html.
54. *Helland P.* Accountants Don't Use Erasers. June 14, 2007 [Электронный ресурс]. — Режим доступа: https://blogs.msdn.microsoft.com/pathelland/2007/06/14/accountants-dont-use-erasers/.
55. *Yang F.* Dogfooding with Druid, Samza, and Kafka: Metametrics at Metamarkets. June 3, 2015 [Электронный ресурс]. — Режим доступа: https://metamarkets.com/2015/dogfooding-with-druid-samza-and-kafka-metametrics-at-metamarkets/.
56. *Li G., Lv J., Qi H.* Pistachio: Co-Locate the Data and Compute for Fastest Cloud Compute. April 13, 2015 [Электронный ресурс]. — Режим доступа: http://yahoohadoop.tumblr.com/post/116365275781/pistachio-co-locate-the-data-and-compute-for.
57. *Paramasivam K.* Stream Processing Hard Problems — Part 1: Killing Lambda. June 27, 2016 [Электронный ресурс]. — Режим доступа: https://engineering.linkedin.com/blog/2016/06/stream-processing-hard-problems-part-1-killing-lambda.
58. *Fowler M.* CQRS. July 14, 2011 [Электронный ресурс]. — Режим доступа: https://martinfowler.com/bliki/CQRS.html.

59. *Young G.* CQRS Documents. November 2010 [Электронный ресурс]. — Режим доступа: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf.
60. *Schwartz B.* Immutability, MVCC, and Garbage Collection. December 28, 2013 [Электронный ресурс]. — Режим доступа: https://www.xaprb.com/blog/2013/12/28/immutability-mvcc-and-garbage-collection/.
61. *Eloff D., Akhmechet S., Kreps J., et al.* Re: Turning the Database Inside-out with Apache Samza // Hacker News discussion, March 4, 2015 [Электронный ресурс]. — Режим доступа: https://news.ycombinator.com/item?id=9145197.
62. Datomic Development Resources: Excision // Cognitect, Inc. [Электронный ресурс]. — Режим доступа: http://docs.datomic.com/excision.html.
63. Fossil Documentation: Deleting Content from Fossil. 2016 [Электронный ресурс]. — Режим доступа: http://fossil-scm.org/index.html/doc/trunk/www/shunning.wiki.
64. *Kreps J.* The irony of distributed systems is that data loss is really easy but deleting data is surprisingly hard. March 30, 2015 [Электронный ресурс]. — Режим доступа: https://twitter.com/jaykreps/status/582580836425330688.
65. *Luckham D. C.* What's the Difference Between ESP and CEP? August 1, 2006 [Электронный ресурс]. — Режим доступа: http://www.complexevents.com/2006/08/01/what%E2%80%99s-the-difference-between-esp-and-cep/.
66. *Perera S.* How Is Stream Processing and Complex Event Processing (CEP) Different? December 3, 2015 [Электронный ресурс]. — Режим доступа: https://www.quora.com/How-is-stream-processing-and-complex-event-processing-CEP-different.
67. *Arasu A., Babu S., Widom J.* The CQL Continuous Query Language: Semantic Foundations and Query Execution // The VLDB Journal, volume 15, number 2, pages 121–142, June 2006 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/the-cql-continuous-query-language-semantic-foundations-and-query-execution/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F77607%2Fcql.pdf.
68. *Hyde J.* Data in Flight: How Streaming SQL Technology Can Help Solve the Web 2.0 Data Crunch // ACM Queue, volume 7, number 11, December 2009 [Электронный ресурс]. — Режим доступа: http://queue.acm.org/detail.cfm?id=1667562.
69. Esper Reference, Version 5.4.0 // EsperTech, Inc., April 2016 [Электронный ресурс]. — Режим доступа: http://www.espertech.com/esper/release-5.4.0/esper-reference/html_single/index.html.
70. *Nabi Z., Bouillet E., Bainbridge A., Thomas C.* Of Streams and Storms // IBM technical report, April 2014 [Электронный ресурс]. — Режим доступа: https://developer.ibm.com/streamsdev/wp-content/uploads/sites/15/2014/04/Streams-and-Storm-April-2014-Final.pdf.
71. *Pathirage M., Hyde J., Pan Y., Plale B.* SamzaSQL: Scalable Fast Data Management with Streaming SQL // IEEE International Workshop on High-Performance Big Data Computing (HPBDC), May 2016 [Электронный ресурс]. — Режим

доступа: https://github.com/milinda/samzasql-hpbdc2016/blob/master/samzasql-hpbdc2016.pdf.

72. *Flajolet P., Fusy É., Gandouet O., Meunier F.* HyperLog Log: The Analysis of a Near-Optimal Cardinality Estimation Algorithm // Conference on Analysis of Algorithms (AofA), June 2007 [Электронный ресурс]. — Режим доступа: http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf.
73. *Kreps J.* Questioning the Lambda Architecture. July 2, 2014 [Электронный ресурс]. — Режим доступа: https://www.oreilly.com/ideas/questioning-the-lambda-architecture.
74. *Hellström I.* An Overview of Apache Streaming Technologies. March 12, 2016 [Электронный ресурс]. — Режим доступа: https://databaseline.wordpress.com/2016/03/12/an-overview-of-apache-streaming-technologies/.
75. *Kreps J.* Why Local State Is a Fundamental Primitive in Stream Processing. July 31, 2014 [Электронный ресурс]. — Режим доступа: https://www.oreilly.com/ideas/why-local-state-is-a-fundamental-primitive-in-stream-processing.
76. *Banon S.* Percolator. February 8, 2011 [Электронный ресурс]. — Режим доступа: https://www.elastic.co/blog/percolator.
77. *Woodward A., Kleppmann M.* Real-Time Full-Text Search with Luwak and Samza. April 13, 2015 [Электронный ресурс]. — Режим доступа: http://martin.kleppmann.com/2015/04/13/real-time-full-text-search-luwak-samza.html.
78. Apache Storm 1.0.1 Documentation. May 2016 [Электронный ресурс]. — Режим доступа: https://storm.apache.org/releases/1.0.1/index.html.
79. *Akidau T.* The World Beyond Batch: Streaming 102. January 20, 2016 [Электронный ресурс]. — Режим доступа: https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102.
80. *Ewen S.* Streaming Analytics with Apache Flink // Kafka Summit, April 2016 [Электронный ресурс]. — Режим доступа: https://www.confluent.io/resources/kafka-summit-2016/advanced-streaming-analytics-apache-flink-apache-kafka/.
81. *Akidau T., Balikov A., Bekiroğlu K., et al.* MillWheel: Fault-Tolerant Stream Processing at Internet Scale // 39th International Conference on Very Large Data Bases (VLDB), August 2013 [Электронный ресурс]. — Режим доступа: https://research.google.com/pubs/pub41378.html.
82. *Dean A.* Improving Snowplow's Understanding of Time. September 15, 2015 [Электронный ресурс]. — Режим доступа: https://snowplowanalytics.com/blog/2015/09/15/improving-snowplows-understanding-of-time/.
83. Windowing (Azure Stream Analytics) // Microsoft Azure Reference, April 2016 [Электронный ресурс]. — Режим доступа: https://msdn.microsoft.com/en-us/library/azure/dn835019.aspx.
84. State Management // Apache Samza 0.10 Documentation, December 2015 [Электронный ресурс]. — Режим доступа: http://samza.apache.org/learn/documentation/0.10/container/state-management.html.

85. *Ananthanarayanan R., Basker V., Das S., et al.* Photon: Fault-Tolerant and Scalable Joining of Continuous Data Streams // ACM International Conference on Management of Data (SIGMOD), June 2013 [Электронный ресурс]. — Режим доступа: https://research.google.com/pubs/pub41318.html.

86. *Kleppmann M.* Samza Newsfeed Demo. September 2014 [Электронный ресурс]. — Режим доступа: https://github.com/ept/newsfeed.

87. *Kirwin B.* Doing the Impossible: Exactly-Once Messaging Patterns in Kafka. November 28, 2014 [Электронный ресурс]. — Режим доступа: http://ben.kirw.in/2014/11/28/kafka-patterns/.

88. *Helland P.* Data on the Outside Versus Data on the Inside // 2nd Biennial Conference on Innovative Data Systems Research (CIDR), January 2005 [Электронный ресурс]. — Режим доступа: http://cidrdb.org/cidr2005/papers/P12.pdf.

89. *Kimball R., Ross M.* The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling, 3rd edition. John Wiley & Sons, 2013.

90. *Klang V.* I'm coining the phrase 'effectively-once' for message processing with at-least-once + idempotent operations. October 20, 2016 [Электронный ресурс]. — Режим доступа: https://twitter.com/viktorklang/status/789036133434978304.

91. *Zaharia M., Das T., Li H., et al.* Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters // 4th USENIX Conference in Hot Topics in Cloud Computing (HotCloud), June 2012 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/system/files/conference/hotcloud12/hotcloud12-final28.pdf.

92. *Tzoumas K., Ewen S., Metzger R.* High-Throughput, LowLatency, and Exactly-Once Stream Processing with Apache Flink // data- August 5, 2015 [Электронный ресурс]. — Режим доступа: https://data-artisans.com/blog/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink.

93. *Carbone P., Fóra G., Ewen S., et al.* Lightweight Asynchronous Snapshots for Distributed Dataflows. June 29, 2015 [Электронный ресурс]. — Режим доступа: https://arxiv.org/abs/1506.08603.

94. *Betts R., Hugg J.* Fast Data: Smart and at Scale. Report, O'Reilly Media, October 2015 [Электронный ресурс]. — Режим доступа: http://www.oreilly.com/data/free/fast-data-smart-and-at-scale.csp.

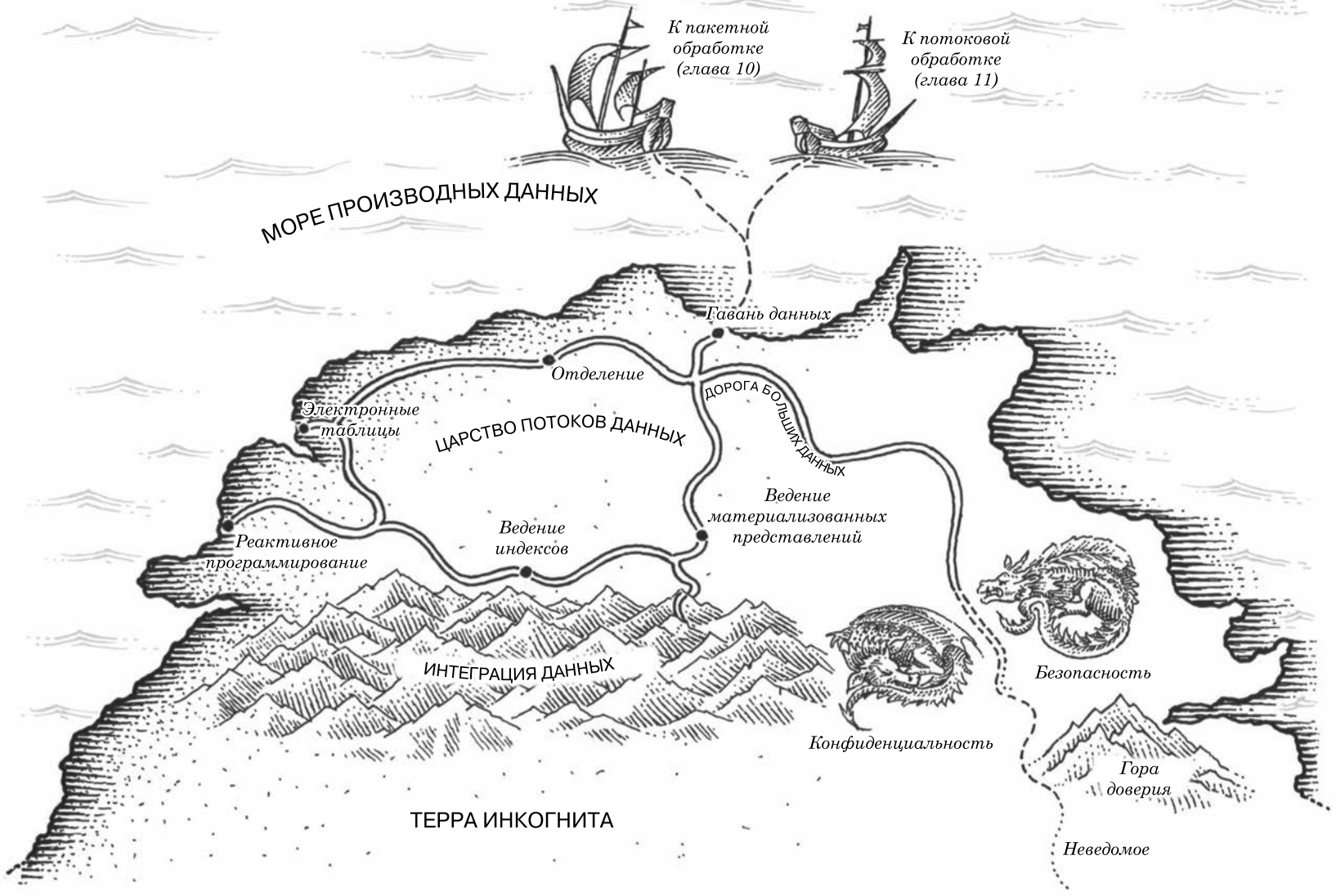
95. *Junqueira F.* Making Sense of Exactly-Once Semantics // Strata+Hadoop World London, June 2016 [Электронный ресурс]. — Режим доступа: https://conferences.oreilly.com/strata/strata-eu-2016/public/schedule/detail/49690.

96. *Gustafson J., Junqueira F., Mehta A., Subramanian S., Wang G.* KIP-98 — Exactly Once Delivery and Transactional Messaging. November 2016 [Электронный ресурс]. — Режим доступа: https://cwiki.apache.org/confluence/display/KAFKA/KIP-98+-+Exactly+Once+Delivery+and+Transactional+Messaging.

97. *Helland P.* Idempotence Is Not a Medical Condition // Communications of the ACM, volume 55, number 5, page 56, May 2012 [Электронный ресурс]. — Режим

доступа: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.401.1539&rep=rep1&type=pdf.

98. *Kreps J.* Re: Trying to Achieve Deterministic Behavior on Recovery/Rewind // email to samza-dev mailing list, September 9, 2014 [Электронный ресурс]. — Режим доступа: http://mail-archives.apache.org/mod_mbox/samza-dev/201409.mbox/%3CCAOeJiJg%2Bc7Ei%3DgzCuOz30DD3G5Hm9yFY%3DUJ6SafdNUFbvRgorg%40mail.gmail.com%3E.
99. *Elnozahy (Mootaz) E. N., Alvisi L., Wang Y.-M., Johnson D. B.* A Survey of Rollback-Recovery Protocols in Message-Passing Systems // ACM Computing Surveys, volume 34, number 3, pages 375–408, September 2002 [Электронный ресурс]. — Режим доступа: http://www.cs.utexas.edu/~lorenzo/papers/SurveyFinal.pdf.
100. *Warski A.* Kafka Streams — How Does It Fit the Stream Processing Landscape? June 1, 2016 [Электронный ресурс]. — Режим доступа: https://softwaremill.com/kafka-streams-how-does-it-fit-stream-landscape/.


К пакетной обработке (глава 10)
К потоковой обработке (глава 11)
МОРЕ ПРОИЗВОДНЫХ ДАННЫХ
Гавань данных
Отделение
Электронные таблицы
ЦАРСТВО ПОТОКОВ ДАННЫХ
ДОРОГА БОЛЬШИХ ДАННЫХ
Ведение материализованных представлений
Ведение индексов
Реактивное программирование
ИНТЕГРАЦИЯ ДАННЫХ
Безопасность
Конфиденциальность
Гора доверия
ТЕРРА ИНКОГНИТА
Неведомое

12 Будущее информационных систем

> ...Если одна вещь определена как к своей конечной цели к другой вещи, то ее конечная цель не может состоять в сохранении собственно бытия.
> Так, конечной целью капитана не может быть сохранение вверенного ему судна, поскольку судно определено к другой цели, а именно к навигации.
>
> (Часто цитируется так: если бы высшей целью капитана было сохранение судна, то он навсегда остался бы в порту.)
>
> *Св. Фома Аквинский.*
> *Сумма теологии (1265–1274)*

До сих пор в данной книге в основном описывалось существующее *сейчас*. В этой заключительной главе мы переместим взгляд на перспективу и обсудим то, что *должно быть*: я предложу те или иные идеи и подходы, которые, как я считаю, могут фундаментально улучшить способы проектирования и создания приложений.

Мнения и предположения о будущем, конечно, субъективны, и поэтому в данной главе я, высказывая свои личные взгляды, буду говорить от первого лица. Вы можете не согласиться с ними и сформировать собственное мнение. Но я надеюсь, что идеи, изложенные в настоящей главе, по крайней мере станут отправной точкой для продуктивной дискуссии и внесут ясность в концепции, которые часто путают.

Цель книги была изложена в главе 1: изучить, как создаются *надежные*, *масштабируемые* и *удобные в сопровождении* приложения и системы. Эти темы прошли красной нитью через все главы: например, мы обсудили многие алгоритмы отказоустойчивости, позволяющие усовершенствовать надежность, секционирование как инструмент улучшения масштабируемости и механизмы эволюции и абстракции, повышающие удобство сопровождения. В данной главе мы объединим все эти идеи и попробуем представить их будущее. Наша цель — узнать, как разрабатывать приложения лучше сегодняшних — еще более надежные, корректные, эволюционирующие и в конечном счете полезные для человечества.

12.1. Интеграция данных

В этой книге многократно повторялась идея о том, что у любой проблемы есть несколько решений, у каждого из которых — свои достоинства, недостатки и компромиссы. Например, в главе 3, изучая системы хранения информации, мы познакомились с журналированным и столбцовым хранилищами и B-деревьями. Обсуждая репликацию в главе 5, мы увидели варианты с одним ведущим узлом, несколькими такими узлами и вовсе без него.

В случае проблемы — например, «я хочу сохранить некие данные и позже снова их просмотреть» — нет ни одного правильного решения, но зато есть много разных подходов, более или менее приемлемых, в зависимости от ситуации. При программной реализации обычно приходится выбирать какой-то один из них. Достаточно сложно получить код, который был бы одновременно надежным и очень быстрым. Попытка получить все сразу в одном программном обеспечении почти гарантированно ведет к неудачной реализации.

Следовательно, выбор ПО зависит от обстоятельств. Любой программный продукт, даже так называемые базы данных «общего назначения», предназначен для конкретного стиля использования.

При таком изобилии альтернатив первая задача заключается в том, чтобы разобраться, в каких обстоятельствах те или иные программные продукты проявят себя лучше всего. Продавцы по понятным причинам неохотно расскажут вам о тех рабочих процессах, для которых их ПО подходит плохо. Но я надеюсь, предыдущие главы снабдили вас знанием о том, какие вопросы нужно задать, чтобы прочитать ответ между строк и найти наилучший компромисс.

Однако даже если вы прекрасно понимаете соответствие между инструментами и областями их применения, возникает еще одна проблема: в сложных приложениях данные часто задействуются несколькими способами. Вряд ли найдется один программный продукт, который подойдет для *всех* вариантов использования данных, поэтому вам, чтобы обеспечить функционирование приложения, неизбежно приходится объединять несколько таких продуктов.

Объединение специализированных инструментов путем сбора информации

Например, часто требуется интегрировать в базу данных OLTP полнотекстовый поисковый индекс для обработки запросов по произвольным ключевым словам. В некоторых БД (таких как PostgreSQL) реализована функция полнотекстового индексирования, и для приложений ее может быть достаточно [1]. Но более сложные поисковые системы требуют специализированных средств поиска информации. И наоборот, поисковые индексы, как правило, не очень подходят для создания надежных систем записи, поэтому во многих приложениях необходимо комбинировать два разных инструмента, чтобы удовлетворить все требования.

Мы затронули тему интеграции информационных систем в подразделе «Синхронизация систем» раздела 11.2. По мере увеличения количества различных представлений данных проблема интеграции становится все сложнее. Помимо БД и поискового индекса, возможно, вам нужно хранить копии в аналитических системах (складах данных или системах пакетной и потоковой обработки); поддерживать кэши или денормализованные версии объектов, полученные из исходных данных. Или, как вариант, пропускать информацию через системы машинного обучения, классификации, ранжирование или выдачу рекомендаций; а также отправлять уведомления, основанные на изменении данных.

Удивительно часто мне приходится слышать от разработчиков программного обеспечения заявления такого рода: «По моему опыту, 99 % людей нуждаются только в X» или «...не нуждаются в X» (для разных X). Я думаю, подобные утверждения говорят больше об опыте докладчика, чем о фактической пользе технологии. Диапазон различных операций, которые вы, возможно, захотите выполнить с данными, до головокружения широк. То, что один человек считает странным или бессмысленным, для другого вполне может быть насущным требованием. Необходимость интеграции данных часто становится очевидной, если посмотреть на потоки данных шире — в рамках всей организации.

Рассуждения об информационных потоках

Если копии одних и тех же данных должны обслуживаться несколькими системами хранения информации, в соответствии с разными шаблонами доступа, то необходимо четко разделить входные и выходные данные: куда они записываются вначале, какие представления и из каких источников формируются? Как гарантировать, что данные будут поступать куда надо и в правильных форматах?

Например, можно вначале построить систему записей в базу данных и прежде всего заносить информацию туда, фиксируя внесенные изменения (см. подраздел «Перехват изменений данных» раздела 11.2), а затем применяя их к поисковому индексу в том же порядке. Если единственным способом обновления индекса является сбор

данных об изменениях (CDC), то вы можете быть уверены, что индекс полностью получен из системы записи и, следовательно, будет с ней совместим (запрет ошибок в программном обеспечении). Запись в базу является единственным способом ввести данные в эту систему.

Позволяя приложению напрямую записывать данные в поисковый индекс и базу, мы рискуем получить проблему, показанную на рис. 11.4, где два клиента конкурентно отправляют конфликтующие записи, а две системы хранения обрабатывают их в другом порядке. В этом случае ни БД, ни поисковый индекс не «отвечают» за определение последовательности записей. Как следствие, они могут принимать противоречивые решения и соответствие между ними нарушится.

Если есть возможность пропустить пользовательские входные данные через единую систему, которая принимает решение о последовательности всех операций записи, то будет намного проще получить другие представления данных, обрабатывая записи в том же порядке. Так выглядит применение метода репликации конечных автоматов, описанного в подразделе «Рассылка общей последовательности» раздела 9.3. Независимо от того, что вы задействуете — фиксацию изменений данных или журнал источников событий, — это менее важно, чем четкое соглашение об общем порядке событий.

Обновление производной информационной системы на основе журнала событий часто бывает детерминированным и идемпотентным (см. пункт «Идемпотентность» подраздела «Отказоустойчивость» раздела 11.3), что позволяет легко восстанавливать ее после сбоя.

Производные данные и распределенные транзакции

Классический способ обеспечить совместимость разных информационных систем, в каждой из которых используются распределенные транзакции, описан в подразделе «Атомарная и двухфазная фиксация (2PC)» раздела 9.4. Чем технология производных систем передачи данных отличается от распределенных транзакций?

На абстрактном уровне они достигают сходных целей, но различными способами. При использовании распределенных транзакций последовательность записей определяется благодаря блокировкам взаимного исключения (см. подраздел «Двухфазная блокировка (2PL)» раздела 7.3), в то время как с помощью CDC и источников событий последовательность событий выстраивается по журналу. В распределенных транзакциях для гарантии того, что изменения вступают в силу ровно один раз, применяется технология атомарных фиксаций; системы на основе журнала, напротив, часто опираются на детерминированные повторы и идемпотентность.

Главное различие заключается в том, что транзакционные системы обычно обеспечивают линеаризуемость (см. раздел 9.2). Это дает такие полезные гарантии, как чтение только своих записей (см. подраздел «Читаем свои же записи» раздела 5.2). Если говорить о производных информационных систем, то они часто

обновляются асинхронно, поэтому по умолчанию не обеспечивают такие же гарантии хронометража.

В ограниченных средах, готовых нести вычислительные расходы, связанные с распределенными транзакциями, эта технология задействуется успешно. Тем не менее я считаю, что у XA низкие показатели отказоустойчивости и производительности (см. подраздел «Распределенные транзакции на практике» раздела 9.4), и это серьезно ограничивает ее применимость. Я уверен в возможности создать лучший протокол для распределенных транзакций, но получить такой протокол, сделать его широко используемым и интегрировать с существующими инструментами будет сложной задачей и вряд ли это произойдет в ближайшем будущем.

В отсутствие хорошего протокола распределенных транзакций, пользующегося широкой поддержкой, наиболее перспективной технологией интеграции различных систем я считаю производные данные на основе журналирования. Тем не менее такие гарантии, как чтение собственных писем, полезны, и я не думаю, что было бы продуктивно говорить всем, что «конечная согласованность неизбежна — смиритесь и научитесь с ней жить» (по крайней мере не без хорошего руководства по тому, *как* это делать).

В разделе 12.3 мы обсудим некоторые методы обеспечения более надежных гарантий в асинхронных производных системах и попробуем найти промежуточное решение между распределенными транзакциями и асинхронными системами на основе журналирования.

Ограничения тотального журналирования

Пока система относительно мала, создание полностью упорядоченного журнала событий вполне возможно (о чем свидетельствует популярность баз данных с репликацией с одним ведущим узлом, которые строят именно такой журнал). Однако по мере развития системы становятся все крупнее, потоки данных в них усложняются, и возникают следующие ограничения.

- В большинстве случаев для построения полностью упорядоченного журнала требуется, чтобы все события проходили через *единый ведущий узел*, определяющий последовательность. Если плотность событий больше, чем может обрабатывать один компьютер, то необходимо разделить нагрузку на несколько машин (см. подраздел «Секционирование журналов» раздела 11.1), и тогда порядок событий в разных секциях будет неоднозначен.
- Если серверы находятся в разных, *географически разделенных* ЦОДах, то, например, чтобы предотвратить отключение всего центра, обычно в каждом ЦОДе есть по одному ведущему узлу, поскольку задержки в работе сети делают синхронную межцентровую координацию неэффективной (см. раздел 5.3). Это приводит к неопределенной последовательности событий, происходящих в разных центрах.

- При развертывании приложений в виде *микросервисов* (см. подраздел «Поток данных через сервисы: REST и RPC» раздела 4.2) построение общей архитектуры заключается в том, чтобы развернуть каждый сервис и его долговременное хранилище как отдельный модуль, без общего долговременного хранилища для всех сервисов. Когда два события поступают из разных сервисов, у этих событий нет определенной последовательности.
- Отдельные приложения поддерживают хранение состояния на стороне клиента, которое обновляется сразу после ввода пользователем данных (не дожидаясь подтверждения от сервера) и даже продолжает работать при отключении от сети (см. пункт «Офлайн-клиенты» подраздела «Сценарии использования репликации с несколькими ведущими узлами» раздела 5.3). В таких приложениях клиенты и серверы, скорее всего, будут видеть события в разной последовательности.

Формально решение об общей последовательности событий известно как *рассылка общей последовательности*, что эквивалентно консенсусу (см. пункт «Консенсусные алгоритмы и рассылка общей последовательности» подраздела «Отказоустойчивый консенсус» раздела 9.3). Большинство консенсусных алгоритмов предназначено для ситуаций, когда пропускная способность каждого узла достаточна для обработки всего потока событий. Эти алгоритмы не обеспечивают механизм совместного упорядочения событий между несколькими узлами. Исследовательская задача разработки консенсусных алгоритмов, масштабируемых за пределы пропускной способности одного узла и хорошо приспособленных для географически распределенных систем, по-прежнему остается открытой.

Упорядочение событий для сохранения причинно-следственных связей

Когда между событиями нет причинно-следственной связи, отсутствие общей последовательности не является большой проблемой, поскольку конкурентные события могут быть упорядочены произвольно. Некоторые другие случаи легко разрешимы: например, несколько обновлений одного и того же объекта можно полностью упорядочить, направляя все обновления для объекта с определенным ID в один раздел журнала. Однако бывают и более тонкие причинно-следственные связи (см. также подраздел «Порядок и причинность» раздела 9.3).

Например, предположим, в социальной сети были два пользователя, муж и жена, но позже они разошлись. Один из пользователей удаляет другого из списка друзей, а затем отправляет сообщение остальным своим друзьям, жалуясь на бывшего партнера. Пользователь ожидает, что тот не увидит грубое сообщение, поскольку оно было отправлено после отмены статуса друга.

Однако в системе статус дружбы хранится в одном месте, а сообщения — в другом, и последовательность между событиями *удаления из списка друзей* и *отправки*

сообщения может быть потеряна. Если не была установлена причинно-следственная зависимость, то сервис, отправляющий уведомления о новых сообщениях, может обработать событие *отправки сообщения* раньше, чем событие *удаления из списка друзей* и, таким образом, по ошибке отправлять уведомление бывшему партнеру.

В данном примере уведомления, в сущности, являются объединением сообщений и списка друзей, следовательно, важно учитывать рассмотренные ранее аспекты хронометража объединений (см. пункт «Зависимость объединений от времени» подраздела «Объединения потоков» раздела 11.3). К сожалению, у этой проблемы нет простого решения [2, 3], выделим лишь отправные точки.

- Логические метки времени обеспечивают полное упорядочение без согласования (см. подраздел «Упорядоченность по порядковым номерам» раздела 9.3), так что могут помочь в тех случаях, когда рассылка общей последовательности невозможна. Но и в этом случае получателям приходится обрабатывать события, поступившие с нарушением последовательности, для чего требуется передача дополнительных метаданных.
- Если можно внести в журнал событие с записью состояния системы, которое пользователь видел перед принятием решения, и присвоить этому событию уникальный идентификатор, то для сохранения причинно-следственной связи все последующие события могут ссылаться на этот ID [4]. Мы вернемся к данной идее в пункте «Чтение тоже событие» подраздела «Наблюдение за производными состояниями» раздела 12.2.
- Алгоритмы разрешения конфликтов (см. врезку «Автоматическое разрешение конфликтов» в пункте «Пользовательская логика разрешения конфликтов» подраздела «Обработка конфликтов записи» раздела 5.3) помогают обрабатывать события, которые были доставлены с нарушением последовательности. Они полезны для обработки состояния, но не помогают, если действия имеют внешние побочные эффекты (такие как отправка уведомления пользователю).

Возможно, со временем появятся паттерны для разработки приложений, которые позволят эффективно определять причинно-следственные зависимости, и производные состояния будут формироваться правильно, не пропуская все события через бутылочное горлышко рассылки общей последовательности.

Пакетная и потоковая обработка

По моему мнению, цель интеграции данных — гарантировать, что данные попадают во все нужные места, будучи представленными в нужной форме. Для этого следует прочитать входные данные, преобразовать их, объединить, отфильтровать, агрегировать, применить модели обучения, оценки и, наконец, записать результаты. Инструментами для достижения цели являются пакетные и потоковые процессоры.

Результатами пакетных и потоковых процессов являются производные наборы данных, такие как поисковые индексы, материализованные представления, рекомендации для пользователей, совокупные показатели и т. п. (см. подраздел «Выходные данные пакетных потоков» раздела 10.2 и подраздел «Применение обработки потоков» раздела 11.3).

Как было показано в главах 10 и 11, у пакетной и потоковой обработки есть много общих принципов, но главное принципиальное отличие между ними заключается в том, что потоковые процессоры работают с неограниченными наборами данных, тогда как входные данные пакетного процесса имеют известный конечный размер. Есть также много мелких различий в способах реализации систем обработки, но сейчас они начинают размываться.

В Spark потоковая обработка выполняется на базе процессора пакетной обработки с разбиением потока на *микропакеты*, в Apache Flink же пакетная обработка выполняется на базе системы потоковой обработки [5]. В принципе, один тип обработки может быть эмулирован на основе другого, хотя их вычислительные характеристики могут отличаться, например, микропотоки могут плохо работать для «прыгающих» или скользящих окон [6].

Сопровождение производного состояния

У пакетной обработки довольно сильный функциональный привкус (даже если код не написан на языке функционального программирования): он поощряет детерминированные, чистые функции, у которых выходные данные зависят только от входных и у которых нет побочных эффектов, кроме явного выхода. Входные данные обрабатываются как неизменяемые, а выходные — в режиме дописывания. Обработка потоков аналогична, но расширяет возможности операторов, чтобы обеспечить управляемое отказоустойчивое состояние (см. пункт «Восстановление состояния после сбоя» подраздела «Отказоустойчивость» раздела 11.3).

Принцип детерминированных функций с четко определенными входными и выходными данными не только хорош для обеспечения отказоустойчивости (см. пункт «Идемпотентность» подраздела «Отказоустойчивость» раздела 11.3), но и упрощает определение информационных потоков в организации [7]. Независимо от того, что собой представляют производные данные: поисковый индекс, статистическую модель или кэш, — полезно подумать о них в терминах информационных конвейеров, формирующих одну вещь на основе другой, передающих изменения состояния системы через функциональный код приложения и применяющих результат к производным системам.

В принципе, производные информационные системы могут поддерживаться синхронно, подобно тому, как реляционная база данных синхронно обновляет вторичные индексы в той же транзакции, к которой относится операция записи в индексируемую таблицу. Однако асинхронность является тем, что делает си-

стемы, основанные на журналах событий, надежными: благодаря ей локальные ошибки не выходят за пределы своей части системы. Тогда как распределенные транзакции прерываются в случае сбоя одного из участников, из-за чего сбои усиливаются, распространяясь на остальные системы (см. пункт «Ограничения для распределенных транзакций» подраздела «Распределенные транзакции на практике» раздела 9.4).

Как было показано в разделе 6.3, вторичные индексы часто выходят за границы секций. Секционированная система с вторичными индексами должна выполнять либо запись в несколько секций (если индекс разбит по терминам), либо чтение (если разбит по документам). Такая связь между секциями является наиболее надежной и масштабируемой в случае асинхронной поддержки индекса [8] (см. также пункт «Обработка данных, хранящихся в нескольких разделах» подраздела «Наблюдение за производными состояниями» раздела 12.2).

Повторная обработка данных при развитии приложения

Пакетная и потоковая обработка полезны и при обслуживании производных данных. Потоковая обеспечивает низкую задержку при отражении входных данных в производных представлениях, тогда как пакетная позволяет обрабатывать большие объемы накопленной исторической информации и получать новые представления на основе существующего набора данных.

В частности, обработка существующей информации — хороший механизм для обслуживания схемы и ее развития, в ходе которого появляются новые функции и изменяются требования к системе (см. главу 4). Без повторной обработки эволюция схемы ограничивается простыми изменениями наподобие добавления к записи необязательного поля или создания нового типа записей. Это касается как схем записи, так и схем чтения (см. пункт «Гибкость схемы в документной модели» подраздела «Реляционные и документоориентированные базы данных сегодня» раздела 2.1). Однако при повторной обработке можно преобразовать набор данных в совершенно другую модель, которая лучше соответствует новым требованиям.

Миграции схемы на железных дорогах

Крупномасштабные миграции схемы встречаются также и в некомпьютерных системах. Например, на заре строительства железных дорог в Англии XIX века существовали разные конкурирующие стандарты колеи (расстояния между рельсами). Поезда, построенные для одной ширины колеи, не могли ходить по рельсам, проложенным по другому стандарту. Это ограничивало возможные пересечения железнодорожных путей [9].

После того как в 1846 году был окончательно утвержден единый стандарт, нужно было перестроить дороги с другой шириной колеи. Но как это сделать, не останавливая движение поездов на несколько месяцев или даже лет? Решение было найдено: сначала перестроить дороги на *двойную*, или *смешанную, колею*, добавив третий рельс. Это можно было сделать постепенно, а потом поезда обоих стандартов могли ходить по одной линии, используя два из трех рельсов. В конце концов, когда все поезда переоборудовали в расчете на стандартную колею, дополнительный рельс, обеспечивавший нестандартным поездам возможность ходить по рельсам, мог быть удален.

Таким образом, «повторная обработка» существующих дорог и одновременное существование старой и новой версий позволило за несколько лет постепенно поменять ширину колеи. Однако это обошлось дорого — вот почему до сих пор существуют железные дороги с нестандартной колеей. Например, система BART в районе залива Сан-Франциско использует ширину колеи, отличную от остальных железных дорог США.

Производные представления позволяют развивать систему *постепенно*. Чтобы реструктуризировать набор данных, не нужно выполнять всю миграцию одновременно и сразу. Вместо этого можно некоторое время поддерживать старую и новую схемы параллельно, как два независимых производных представления для одних и тех же базовых данных. Затем переключить на новое представление небольшую группу пользователей, чтобы проверить его производительность и выявить ошибки, в то время как остальные пользователи будут перенаправляться на старое представление. Постепенно можно увеличить количество пользователей, имеющих доступ к новому представлению, и в итоге отказаться от старого [10].

Красота такой постепенной миграции заключается в том, что каждый этап процесса легко обратим: если некий момент пойдет не так, то у вас всегда есть рабочая система, к которой можно вернуться. Уменьшая риск необратимого ущерба, вы можете двигаться вперед более уверенно и, следовательно, быстрее улучшить систему [11].

Лямбда-архитектура

Если пакетная обработка используется для повторной обработки исторических данных, а потоковая обработка — для последних обновлений, то как их объединить? Для решения этой задачи была предложена *лямбда-архитектура* [12], которая привлекла большое внимание.

Основная идея лямбда-архитектуры заключается в том, чтобы записывать входные данные путем добавления неизменяемых событий к постоянно растущему набору данных, как в источниках событий (см. подраздел «Источники событий» раздела 11.2). По этим событиям строятся оптимизированные для чтения пред-

ставления. В лямбда-архитектуре предлагается запустить две разные системы параллельно: систему пакетной обработки, такую как Hadoop MapReduce, и отдельную систему потоковой обработки, например Storm.

В лямбда-архитектуре потоковый процессор потребляет события и быстро создает приблизительное обновление представления; пакетный процессор позже потребляет *тот же* набор событий и выстраивает уточненную версию производного представления. Смысл такой структуры заключается в том, что пакетная обработка проще и, следовательно, в меньшей степени подвержена ошибкам, в то время как потоковые процессоры считаются менее надежными и более сложными для устранения неполадок (см. подраздел «Отказоустойчивость» раздела 11.3). Кроме того, потоковый процесс может использовать быстрые приближенные алгоритмы, а у пакетного процесса алгоритмы более медленные и точные.

Идеи лямбда-архитектуры оказали благотворное влияние на структуру информационных систем. В особенности это касается популяризации принципа получения представлений из потоков неизменных событий и повторной обработки событий в случае необходимости. Однако я также полагаю, что у нее есть следующие практические проблемы.

- Необходимость поддерживать одну и ту же логику как для пакетной, так и для потоковой системы обработки — это значительные дополнительные усилия. Хотя такие библиотеки, как Summingbird [13], предоставляют абстракцию для вычислений, которые могут выполняться как в пакетном, так и в потоковом контексте, операционная сложность отладки, настройки и поддержки двух разных систем остается [14].
- Поскольку потоковый и пакетный конвейеры производят разные выходные данные, для ответа на запросы пользователей эти данные необходимо объединить. Выполнить такое слияние довольно просто, когда вычисление суть простое агрегирование по «падающему» окну. Но если представление получается с помощью более сложных операций, таких как объединения и сессионизация, или выходные данные не являются временным рядом, то задача значительно усложняется.
- Иметь возможность повторной обработки всего исторического набора данных — это прекрасно, однако большие наборы данных нередко требуют значительных вычислительных затрат. Следовательно, пакетный конвейер зачастую необходимо настроить на обработку дополняющих пакетов (таких как данные, поступившие за час, в конце каждого часа), а не повторную обработку всего сразу. Это вызывает проблемы, рассмотренные в подразделе «Рассуждения о времени» раздела 11.3, например, обработку отставших пакетов и поддержку окон, пересекающих границы между пакетами. Инкрементализация пакетного вычисления увеличивает его сложность и делает его более похожим на потоковый уровень, что противоречит цели сделать пакетную обработку максимально простой.

Унификация пакетной и потоковой обработки

В более поздней работе реализованы преимущества лямбда-архитектуры без ее недостатков. Это позволило реализовать в одной системе как пакетные вычисления (повторная обработка исторических данных), так и потоковые (обработка событий по мере их поступления) [15].

Унификация пакетной и потоковой обработки в одной системе требует следующих функций, становящихся все более доступными.

- Возможность воспроизведения исторических событий в том же процессоре, который обрабатывает поток последних событий. Например, брокеры сообщений на основе журналирования позволяют воспроизводить сообщения (см. пункт «Повторение старых сообщений» подраздела «Секционирование журналов» раздела 11.1), а ряд потоковых процессоров позволяет читать входные данные из распределенной файловой системы, такой как HDFS.
- Семантика однократного выполнения для потоковых процессоров, гарантирующая, что в случае сбоя выходные данные будут такими же, как если бы сбоя не произошло (см. подраздел «Отказоустойчивость» раздела 11.3). Подобно пакетной обработке, это требует отбрасывания частично полученных выходных данных после неудачно завершившихся задач.
- Инструменты для построения окон по времени события, а не по времени обработки, поскольку при повторной обработке исторических событий время обработки не имеет смысла (см. подраздел «Рассуждения о времени» раздела 11.3). Например, в Apache Beam есть API для описания таких вычислений, которые затем могут запускаться с помощью Apache Flink или Google Cloud Dataflow.

12.2. Отделение от баз данных

На самом абстрактном уровне базы данных Hadoop и операционные системы выполняют одни и те же функции: хранят данные, позволяют их обрабатывать и запрашивать [16]. В базе данные хранятся в виде записей, построенных по определенной информационной модели (строки в таблицах, документы, вершины графа и т. п.). В файловой системе они хранятся в файлах, но по своей сути то и другое — системы «управления информацией» [17]. Как было показано в главе 10, экосистема Hadoop представляет собой нечто похожее на распределенную версию Unix.

Конечно, на практике здесь есть много различий. Например, многие файловые системы не очень хорошо справляются с каталогом, содержащим 10 млн небольших файлов, тогда как для базы данных такое количество маленьких записей — совершенно нормальная, ничем не примечательная ситуация. Тем не менее сходства

и различия между операционными системами и базами заслуживают более подробного исследования.

У Unix и реляционных БД очень разные философии в отношении управления информацией. Целью Unix считается предоставление программистам логической, но довольно низкоуровневой аппаратной абстракции, тогда как назначение реляционных баз — предоставить разработчикам приложений высокоуровневую абстракцию, скрывающую сложности информационных структур на диске, а также конкурентный доступ, восстановление после сбоев и т. п. В Unix были разработаны конвейеры и файлы, которые представляют собой просто последовательности байтов, тогда как в базах данных появились SQL и транзакции.

Какой из описанных подходов лучше? Конечно, выбор зависит от того, чего вы хотите. Unix «проще» в том смысле, что это довольно тонкая надстройка над аппаратными ресурсами. Реляционные базы данных «проще» в том смысле, что с помощью коротких декларативных запросов можно строить различные сложные структуры (оптимизация запросов, индексы, объединения, контроль конкурентного доступа, репликация и т. д.) и автору запроса не приходится вдаваться в детали его реализации.

Противоречия между этими философиями сохранялись десятки лет (и Unix, и реляционная модель баз данных появились в начале 1970-х годов) и до сих пор не разрешены. Например, я бы интерпретировал движение NoSQL как желание применить абстракции низкого уровня Unix в области распределенных хранилищ данных OLTP.

В этом разделе я попытаюсь примирить обе философии в надежде, что мы сможем объединить лучшее из этих двух миров.

Объединение технологий хранения данных

В книге мы обсудили различные возможности, предоставляемые базами данных, и технологии их работы. В частности, рассмотрели следующие функции:

- вторичные индексы, позволяющие быстро находить записи по значению поля (см. подраздел «Другие индексные структуры» раздела 3.1);
- материализованные представления, которые являются своего рода заранее вычисленным кэшем результатов запроса (см. подраздел «Агрегирование: кубы данных и материализованные представления» раздела 3.3);
- журналы репликации, сохраняющие актуальные копии данных в других узлах сети (см. подраздел «Реализация журналов репликации» раздела 5.1);
- полнотекстовые поисковые индексы, позволяющие выполнять поиск в тексте по ключевым словам (см. пункт «Полнотекстовый поиск и нечеткие индексы» подраздела «Другие индексные структуры» раздела 3.1), встроенные в некоторые реляционные базы данных [1].

Аналогичные темы затрагивались в главах 10 и 11. Мы говорили о создании полнотекстовых поисковых индексов (см. подраздел «Выходные данные пакетных потоков» раздела 10.2), об обслуживании материализованных представлений (см. пункт «Поддержка материализованных представлений» подраздела «Применение обработки потоков» раздела 11.3) и о репликации изменений из базы данных в производные информационные системы (см. подраздел «Перехват изменений данных» раздела 11.2).

Похоже, есть параллели между функциями, встроенными в базы данных, и производными информационными системами, которые строят на базе пакетных и потоковых процессоров.

Создание индекса

Что происходит, когда мы делаем запрос `CREATE INDEX` с целью создать новый индекс в реляционной БД? База проверяет согласованный снимок таблицы, выбирает все индексируемые значения полей, сортирует их и генерирует на выходе индекс. Затем она будет обрабатывать записи, сделанные с момента создания согласованного снимка (при условии, что при создании индекса таблица не была заблокирована, вследствие чего в ней могут появляться новые записи). После того как это будет сделано, база данных должна будет обновлять индекс всякий раз при выполнении записи в таблицу в ходе транзакции.

Этот процесс очень похож на создание копии ведомого узла (см. подраздел «Создание новых ведомых узлов» раздела 5.1), а также на настройку перехвата изменений данных в потоковой системе (см. пункт «Исходный снимок» подраздела «Перехват изменений данных» раздела 11.2).

Всякий раз при выполнении запроса `CREATE INDEX` база данных, по сути заново перерабатывает существующий набор данных (как было описано в пункте «Повторная обработка данных при развитии приложения» подраздела «Пакетная и потоковая обработка» раздела 12.1) и генерирует индекс как новое представление существующих данных. Они могут представлять собой моментальный снимок состояния, а не журнал всех когда-либо сделанных изменений, но тесно взаимосвязаны (см. подраздел «Состояние, потоки и неизменяемость» раздела 11.2).

Тотальная метабаза данных

В этом свете я считаю, что поток данных всей организации начинает выглядеть как одна огромная БД [7]. Всякий раз, когда пакет, поток или ETL-процесс переносит данные из одних места и формы в другие место и форму, он действует подобно подсистеме базы, которая постоянно обновляет индексы и материализованные представления.

С этой точки зрения пакетные и потоковые процессоры подобны сложным триггерам, хранимым процедурам и процедурам обслуживания материализованных представлений. Поддерживаемые ими производные системы данных подобны индексам разных типов. Например, реляционная БД может поддерживать индексы B-деревьев, хеш- и пространственные индексы (см. пункт «Составные индексы» подраздела «Другие индексные структуры» раздела 3.1) и другие их типы. В новой архитектуре производных информационных систем эти возможности не реализованы как функции единой интегрированной базы данных, а предоставляются различными программными продуктами, работающими на разных машинах и администрируемыми разными группами людей.

Как эти события повлияют на нас в будущем? Если исходить из того, что не существует единой модели данных или формата хранения, подходящих для всех паттернов доступа, то я предполагаю наличие двух основных способов, с помощью которых разные средства хранения и обработки можно тем не менее объединить в связную систему.

- *Объединенные базы данных: унифицированное чтение*. Можно обеспечить единый интерфейс запросов для широкого круга базовых механизмов хранения и методов обработки — подход, известный как *объединенная база данных* или *полихранилище* [18, 19]. Например, этому паттерну соответствует *адаптер внешних данных* в PostgreSQL [20]. Приложения, которые нуждаются в специализированной модели данных или интерфейсе запросов, по-прежнему могут напрямую обращаться к основным хранилищам, тогда как пользователям, желающим объединить данные из разных мест, легко сделать это через объединенный интерфейс.

  Интерфейс объединенного запроса соответствует реляционной традиции единой интегрированной системы с языком запросов высокого уровня и элегантной семантикой, но имеет сложную реализацию.
- *Разделенные базы данных: унифицированная запись*. Идея объединения баз касается запросов на чтение в разных системах, она не предлагает хороших решений для синхронизации записи в этих системах. Как уже отмечалось, создание согласованного индекса в единой БД является встроенной функцией. Но при объединении нескольких систем хранения информации также необходимо гарантировать, что все измененные данные будут попадать в нужные места даже в случае сбоев. Упрощение надежного подключения систем хранения (например, путем сбора данных об изменениях и создания журналов событий) подобно *разделению* функций обслуживания индекса в базе таким образом, чтобы можно было синхронизировать записи, сделанные по разным технологиям [7, 21].
- Принцип разделения соответствует традициям Unix: малые инструменты, хорошо выполняющие одну конкретную операцию [22] и обменивающиеся данными через унифицированный API (конвейеры) низкого уровня и которые можно объединять, используя язык более высокого уровня (оболочку) [16].

Как сделать, чтобы разделение работало

Объединение и разделение — две стороны одной медали: надежной, масштабируемой и удобной в сопровождении системы, построенной из различных компонентов. Объединенные запросы на чтение требуют сопоставления данных из двух моделей. Чтобы решить эту задачу, надо как следует подумать, но, в принципе, от нее вполне можно избавиться. Я думаю, что синхронизация записей, находящихся в нескольких системах хранения информации, — более сложная инженерная проблема, и поэтому сосредоточусь на ней.

Традиционный подход к синхронизации записи требует распределенных транзакций в гетерогенных системах хранения информации [18], и это, по моему мнению, является ошибочным решением (см. пункт «Производные данные и распределенные транзакции» подраздела «Объединение специализированных инструментов путем сбора информации» раздела 12.1). Транзакции в рамках единой системы хранения или потоковой обработки возможны, но при пересечении данными границы между различными технологическими решениями я считаю гораздо более надежным и практичным подходом асинхронный журнал событий с идемпотентной записью.

Например, распределенные транзакции применяются в некоторых потоковых процессорах для получения семантики однократного выполнения (см. пункт «Подтверждение малых изменений» подраздела «Отказоустойчивость» раздела 11.3) и могут быть весьма эффективны. Однако когда транзакция затрагивает системы, написанные разными группами людей (например, данные записываются из потокового процессора в распределенное хранилище типа «ключ — значение» или в поисковый индекс), отсутствие стандартизованного протокола транзакций значительно усложняет интеграцию. Упорядоченный журнал событий с идемпотентными потребителями (см. пункт «Идемпотентность» подраздела «Отказоустойчивость» раздела 11.3) является гораздо более простой абстракцией, и, следовательно, его гораздо проще реализовать в гетерогенных системах [7].

Большим преимуществом интеграции на основе журнала является *отсутствие зависимостей* между компонентами. Это проявляется следующими двумя способами.

1. На системном уровне благодаря потокам асинхронных событий система в целом становится более устойчивой к отказам и падению производительности отдельных компонентов. Если потребитель замедляется или у него происходит сбой, то журнал событий может буферизовать сообщения (см. пункт «Использование дискового пространства» подраздела «Секционирование журналов» раздела 11.1), что позволяет инициатору и другим потребителям продолжать работать без изменений. Когда сбой будет устранен, этот потребитель сможет наверстать упущенное — он не пропускает данные, и сбой остается локальным. При распределенных транзакциях с синхронным взаимодействием, наоборот, локальные ошибки распространяются по системе, превращаясь в крупномасштабные сбои

(см. пункт «Ограничения для распределенных транзакций» подраздела «Распределенные транзакции на практике» раздела 9.4).

2. На уровне человеческого восприятия разделение информационных систем позволяет разрабатывать, улучшать и поддерживать различные программные компоненты и сервисы независимо друг от друга, разными командами. Специализация позволяет каждой группе разработчиков сосредоточиться на чем-то одном и сделать это качественно, с четко определенными интерфейсами для других систем. Журналы событий предоставляют достаточно эффективный интерфейс, обеспечивающий четкую согласованность (вследствие долговременного хранения и упорядочения событий) и одновременно вполне общедоступный, применимый практически к любому типу данных.

Разделенные и интегрированные системы

Даже если разделение систем действительно станет технологией будущего, то все равно не заменит базы данных в их нынешнем виде — они будут так же нужны, как и сейчас. Базы по-прежнему необходимы для хранения состояния потоковых процессоров и обслуживания запросов при генерации выходных данных пакетной и потоковой обработки (см. подраздел «Выходные данные пакетных потоков» раздела 10.2 и раздел 11.3). Специализированные механизмы запросов будут попрежнему важны для определенных процессов: например, механизмы запросов в складах данных MPP оптимизированы для исследовательских аналитических запросов и очень хорошо обрабатывают этот вид операций (см. подраздел «Сравнение Hadoop и распределенных баз данных» раздела 10.2).

Сложность совместной работы нескольких различных объектов инфраструктуры способна стать проблемой: каждый программный продукт имеет свою кривую обучения, особенности настройки и операционные нюансы, поэтому стоит развертывать наименьшее количество программных компонентов. Единый интегрированный программный продукт также может обеспечить лучшую и более предсказуемую производительность для тех типов рабочих нагрузок, для которых он предназначен, по сравнению с системой, состоящей из нескольких инструментов, объединенных с помощью программного кода [23]. Как я отмечал в предисловии, разработка с расчетом на масштабирование, в котором вы не нуждаетесь, — зря потраченные усилия и вероятность получить негибкую архитектуру. По сути, это форма преждевременной оптимизации.

Цель разделения не конкуренция с отдельными базами данных по производительности для конкретных рабочих нагрузок, а возможность объединить несколько разных баз и получить хорошую производительность для гораздо более широкого диапазона рабочих нагрузок, чем если бы это было с помощью лишь одного программного продукта. Речь идет не о глубине, а о широте — в том же смысле, что и разнообразие моделей хранения и обработки, о которых говорилось в подразделе «Сравнение Hadoop и распределенных баз данных» раздела 10.2.

Таким образом, если существует одна технология, которая делает все, что нужно, то вам, скорее всего, лучше просто использовать этот продукт, а не пытаться собрать его самостоятельно из компонентов более низкого уровня. Преимущества разделения и компоновки вступают в силу, когда нет единого программного обеспечения, удовлетворяющего всем вашим требованиям.

Чего не хватает

Инструменты для построения информационных систем становятся все совершеннее, но я считаю, что им все еще не хватает одной важной части: эквивалента оболочки Unix в виде разделенной базы данных (другими словами, простого декларативного языка высокого уровня для построения систем хранения и обработки).

Например, мне бы понравилось, если бы можно было просто объявить `mysql | elasticsearch` по аналогии с конвейерами Unix и эта команда была бы разделенным эквивалентом `CREATE INDEX`. То есть взяла бы все документы из базы данных MySQL и проиндексировала их в кластере Elasticsearch, а затем постоянно регистрировала бы все изменения, внесенные в БД, и автоматически вносила бы их в поисковый индекс, не требуя написания специального программного кода. Подобная интеграция должна быть осуществима практически с любой системой хранения или индексации.

Точно так же было бы здорово иметь более простую возможность заранее вычислять и обновлять кэши. Напомню, что материализованное представление, по существу, представляет собой предварительно вычисленный кэш, вследствие чего можно представить создание кэша путем декларативного определения материализованных представлений для сложных запросов, включая рекурсивные запросы на графах (см. раздел 2.3) и логику приложения. В этой области уже есть интересные начальные исследования, такие как *дифференциальный поток данных* [24, 25], и я надеюсь, что описанные идеи найдут реализацию в промышленных системах.

Проектирование приложений на основе потока данных

Принцип разделения баз данных путем построения специализированных систем хранения и обработки, объединенных с помощью программного кода, в последнее время известен как «база данных наоборот» [26] по названию пресс-конференции, которую я дал в 2014 году [27]. Однако назвать его «новой архитектурой» было бы слишком претенциозно. Я рассматриваю его скорее как образец архитектуры, отправную точку для обсуждения, и мы дали этому принципу название, просто чтобы было удобнее его обсуждать.

Сами идеи не мои; это просто объединение идей других людей, у которых, как я считаю, стоит поучиться. В частности, существует много совпадений с языками *пото-*

ков данных, такими как Oz [28] и Juttle [29], языками *функционального реактивного программирования* (functional reactive programming, FRP), подобно Elm [30, 31], и языками *логического программирования*, например Bloom [32]. Термин «*разделение*» в этом контексте был предложен Джеем Крепсом (Jay Kreps) [7].

Даже в электронных таблицах есть возможности программирования потока данных, и они намного опережают большинство обычных языков программирования [33]. В электронной таблице можно ввести формулу в ячейку (например, вычисление суммы значений из ячеек другого столбца), и всякий раз, когда изменится одна из этих ячеек, результат формулы будет автоматически пересчитан. Именно это мы хотим получить на уровне информационной системы: при изменении записи в БД все индексы для данной записи должны автоматически обновляться. Кроме того, следует автоматически обновиться и всем кэшированным представлениям и агрегациям, зависящим от этой записи. Вы не должны думать о технических деталях происходящего, но притом должны быть уверены в правильной работе всех элементов-участников.

Итак, я считаю, что большинству информационных систем все еще есть чему поучиться у функций, которые были реализованы в VisiCalc еще в 1979 году [34]. Отличие современных информационных систем от электронных таблиц заключается вот в чем: информационные системы должны быть отказоустойчивыми, масштабируемыми и рассчитанными на длительное хранение данных. Им также следует интегрироваться с разнообразными программными продуктами, написанными разными группами людей в разное время, многократно использовать существующие библиотеки и сервисы: едва ли стоит ожидать, что все ПО будет разработано на каком-то одном языке, с помощью одной среды разработки или другого инструмента.

В данном подразделе я разовью эти идеи, и мы рассмотрим некоторые способы создания приложений на основе разделенных БД и информационных потоков.

Программный код как функция построения вторичных данных

Когда один набор данных происходит от другого, он обрабатывается какой-то функцией преобразования. Приведу следующие примеры.

- Вторичный индекс — своего рода производный набор данных с простой функцией преобразования: для каждой строки или документа в базовой таблице он выбирает значения проиндексированных столбцов или полей и сортирует результат по этим значениям (исходя из предположения, что это B-дерево или индекс SSTable с сортировкой по ключу, как описано в главе 3).
- Полнотекстовый поисковый индекс создается путем применения различных функций обработки естественного языка, таких как определение языка, сегментация слов, стемминг или лемматизация, коррекция орфографии и идентификация

синонимов, а затем построение информационной структуры для эффективного поиска (например, инвертированного индекса).

- В системе машинного обучения мы можем рассматривать эту модель как получение информации из данных обучения с применением различных функций извлечения и статистического анализа. Когда модель применяется к новым входным данным, ее выходные данные являются производными из входных и самой модели (и, следовательно, косвенно, из данных обучения).
- Кэш часто содержит агрегирование данных в той форме, в которой они будут отображены в пользовательском интерфейсе. Таким образом, заполнение кэша требует знания того, на какие поля ссылается UI; изменения последнего могут потребовать обновления правил заполнения и перестройки кэша.

Функция построения вторичного индекса применяется настолько часто, что встроена в ядро многих систем управления базами данных и ее можно вызвать, просто введя команду `CREATE INDEX`. Основные лингвистические функции полнотекстового индексирования для обычных языков программирования также могут быть встроены в базу данных, но для более сложных функций часто требуется более тонкая и специфичная настройка. В машинном обучении разработка функций очень зависит от конкретных областей применения, и в них часто приходится учитывать подробные знания о взаимодействии пользователей и развертывании приложения [35].

Когда функция, создающая производный набор данных, не является стандартной, такой как создание вторичного индекса, для обработки специфических аспектов конкретного приложения нужно писать специальный, нестандартный, код. Именно он камень преткновения для многих баз данных. В реляционных БД обычно поддерживаются триггеры, хранимые процедуры и пользовательские функции, которые могут применяться для выполнения кода приложения в базе, но эти инструменты в архитектуре БД в каком-то смысле создавались задним числом (см. раздел 11.1).

Разделение кода приложения и состояния

Теоретически база данных может служить средой развертывания для произвольного приложения, например операционной системы. Однако на практике базы плохо подходят для этих целей. Они не соответствуют современным требованиям разработки приложений, таким как управление зависимостями и пакетами, контроль версий, последовательные обновления, эволюция, мониторинг, измерение показателей, вызовы сетевых сервисов и интеграция с внешними системами.

А вот инструменты развертывания и управления кластерами, подобные Mesos, YARN, Docker, Kubernetes и др., созданы специально для запуска приложений. Сосредотачиваясь на одной задаче, они делают это гораздо лучше, чем базы данных,

для которых выполнение пользовательских функций — лишь одна из многочисленных обязанностей.

Я полагаю, есть смысл предназначить отдельные части системы для долговременного хранения данных, в то время как другие части будут специализироваться на запуске приложений. Эти две части могут взаимодействовать, оставаясь независимыми.

Большинство веб-приложений сегодня развертываются как сервисы, без учета состояния. Любой запрос пользователя в них может быть перенаправлен на любой сервер приложений, и последний забывает о запросе сразу после отправки ответа. Такой стиль развертывания удобен, поскольку позволяет добавлять или удалять серверы по мере необходимости, но состояние нужно где-то хранить, и обычно это база данных. Тенденция заключается в том, чтобы отделить логику приложения без учета состояния от управления состоянием (с помощью БД): не размещать логику приложения в базе и не хранить постоянное состояние в приложении [36]. Как любят шутить функциональные программисты: «Мы верим в отделение церкви от государства» [37][1].

В этой типичной модели веб-приложения БД действует как разновидность изменяемой общей переменной, к которой можно получить доступ синхронно по сети. Приложение способно читать и обновлять переменную, а база данных заботится о ее долговечности, обеспечивая контроль конкурентного доступа и отказоустойчивость.

Однако в большинстве языков программирования нельзя подписаться на изменения переменной — ее можно читать только периодически. В отличие от таблицы, здесь объекты, считывающие переменную, не получают уведомления, если ее значение изменяется. (Можно написать специальный код для создания таких уведомлений, реализовав известный *паттерн «Наблюдатель»*, но в большинстве языков программирования нет встроенной функции, реализующей данный паттерн.)

Базы унаследовали этот пассивный подход к изменяемым данным: часто единственный способ узнать, изменилось ли содержимое БД, — провести опрос (то есть периодически повторять запрос). Подписка на изменения как функция еще только начинает проявляться (см. пункт «API для потоков изменений» подраздела «Перехват изменений данных» раздела 11.2).

---

[1] Объяснение шутки редко идет на пользу, но я не хочу, чтобы кто-то из читателей почувствовал себя не у дел. Здесь «церковь» (church) — намек на математика Алонсо Черча (Alonzo Church), изобретшего лямбда-исчисление, раннюю форму вычислений, ставших основой для большинства функциональных языков программирования. Данное исчисление не имеет изменяемого состояния (то есть в нем нет переменных, которые могут быть перезаписаны), поэтому можно сказать, что Черч отделил изменяемое состояние (игра слов: по-английски state — и «состояние», и «государство». — *Примеч. пер.*) от работы приложения.

Потоки данных: взаимодействие между изменениями состояния и приложением

Если думать о приложениях в терминах потоков данных, то надо пересмотреть отношения между кодом приложения и управлением состоянием. Вместо того чтобы рассматривать БД как пассивную переменную, которой манипулирует приложение, мы уделяем гораздо больше внимания взаимодействию и сотрудничеству между состояниями, изменению состояния и коду, их обрабатывающему. Код приложения реагирует на изменение состояния в одном месте, запуская изменение состояния в другом месте.

Мы уже встречали этот принцип в разделе 11.2, когда обсуждали журнал изменений в БД как поток событий, на которые можно подписаться. В системах передачи сообщений, таких как акторы (см. подраздел «Поток данных передачи сообщений» раздела 4.2), также реализована данная концепция реагирования на события. Уже в 1980-х годах существовала модель *пространства кортежей* для описания распределенных вычислений в терминах процессов, отслеживающих изменения состояний и реагирующих на них [38, 39].

Как уже обсуждалось, подобные вещи происходят внутри базы, когда триггер реагирует на изменение данных или обновляется вторичный индекс, отражая изменение индексируемой таблицы. Разделение БД и приложений означает принятие этой идеи и применение ее к созданию производных наборов данных за пределами первичной базы: кэши, полнотекстовые поисковые индексы, машинное обучение или аналитические системы. Для этой цели можно использовать потоковую обработку и системы обмена сообщениями.

Важно понимать: сохранение производных данных — не то же самое, что асинхронное выполнение задачи, для которого традиционно применяются системы обмена сообщениями (см. пункт «Что лучше: журналы или традиционный обмен сообщениями?» подраздела «Секционирование журналов» раздела 11.1).

- При сохранении производных данных часто важна последовательность изменений состояния (если по данным из журнала событий строится несколько представлений, то они должны обрабатывать события в том же порядке, чтобы сохранить их согласованность). Как обсуждалось в пункте «Подтверждение и повторная доставка» подраздела «Системы обмена сообщениями» раздела 11.1, многие брокеры сообщений не обладают этим свойством при повторной доставке неподтвержденных сообщений. Кроме того, недопустимо дублирование записей (см. подраздел «Синхронизация систем» раздела 11.2).
- Отказоустойчивость — обязательное условие для производных данных: потеря одного сообщения приводит к тому, что производный набор данных необратимо рассинхронизируется с их источником. И доставка сообщений, и обновление

производных состояний должны быть надежными. Например, многие системы акторов по умолчанию сохраняют состояние актора и сообщения в памяти, но в случае сбоя эта информация теряется.

Стабильное сохранение последовательности сообщений и отказоустойчивая обработка сообщений являются довольно жесткими требованиями, но это намного дешевле и быстрее, чем распределенные транзакции. Современные потоковые процессоры способны предоставить эти гарантии и обеспечить масштабируемость и позволяют использовать код приложения в качестве потоковых операторов.

Приложения могут выполнять произвольную обработку, которую обычно не выполняют встроенные функции в БД. Подобно инструментам Unix, выстроенным в конвейер, потоковые операторы могут объединяться для создания больших систем, обрабатывающих поток данных. Каждый оператор принимает потоки изменений состояния в качестве входных данных и создает на выходе другие потоки изменений состояния.

Потоковые процессоры и сервисы

Современная тенденция разработки приложений стремится к разделению функциональности на набор *сервисов*, которые обмениваются данными с помощью синхронных сетевых запросов, таких как REST API (см. подраздел «Поток данных через сервисы: REST и RPC» раздела 4.2). Преимущество такой сервис-ориентированной архитектуры, по сравнению с одним монолитным приложением, в первую очередь заключается в организационной масштабируемости за счет ослабления связей: разные команды программистов могут работать над разными сервисами, что снижает интенсивность координации между командами (поскольку сервисы развертываются и обновляются независимо друг от друга).

Объединение потоковых операторов в системы потоков данных имеет много общих свойств с технологией микросервисов [40]. Однако базовый механизм коммуникации у них сильно различается: это однонаправленные, асинхронные потоки сообщений, а не синхронные взаимодействия типа «запрос — ответ».

Кроме преимуществ, перечисленных в подразделе «Поток данных передачи сообщений» раздела 4.2, таких как повышенная отказоустойчивость, системы потоков данных обеспечивают и более высокую производительность. Например, предположим, что покупатель приобретает товар, стоимость которого оценивается в одной валюте, но оплачивается в другой. Для выполнения конвертации валюты необходимо знать текущий обменный курс. Эта операция может быть реализована двумя способами [40, 41].

1. Методом микросервисов. Программный код, обрабатывающий покупки, вероятно, обратится к сервису обменных курсов или БД, чтобы получить текущий курс для данной валюты.

2. Методом потока данных. Программный код, обрабатывающий покупки, заранее подпишется на поток обновлений обменного курса и при каждом изменении текущих значений будет записывать их в локальную базу данных. При необходимости обработать покупку ему нужно только сделать в нее запрос.

Во втором варианте вместо синхронного сетевого запроса другого сервиса использован запрос локальной базы данных (которая может находиться на том же компьютере, даже в том же процессе)[1]. Технология потоков данных не только быстрее, но и более устойчива к отказу другого сервиса. Самый быстрый и надежный сетевой запрос — это отсутствие сетевых запросов! Теперь вместо RPC у нас есть потоковое объединение между событиями купли-продажи и событиями обновления обменного курса (см. пункт «Объединения "поток — таблица" (обогащение потока)» подраздела «Объединения потоков» раздела 11.3).

Это объединение зависит от времени: если события покупки будут обработаны позже, то обменный курс изменится. Чтобы восстановить исходный результат, нужно получить исторический обменный курс в момент покупки. Независимо от того, запрашиваете ли вы сервис или подписываетесь на поток обновлений обменного курса, необходимо учитывать данную зависимость от времени (см. пункт «Зависимость объединений от времени» подраздела «Объединения потоков» раздела 11.3).

Подписка на поток изменений вместо запроса текущего состояния в момент необходимости приближает нас к модели расчета, подобной таблице: когда какая-то часть данных изменяется, все производные данные, зависящие от нее, могут быть быстро обновлены. Есть еще много открытых вопросов, например таких, как зависящие от времени объединения, но я считаю, что создание приложений на базе идей потока данных является очень перспективным направлением.

Наблюдение за производными состояниями

На абстрактном уровне системы потоков данных, обсуждавшиеся в предыдущем подразделе, позволяют создавать производные наборы данных (такие как индексы поиска, материализованные представления и модели прогнозирования) и поддерживать их в актуальном состоянии. Назовем этот процесс *путем записи*: всякий раз, когда в систему записывается какая-то информация, она проходит через несколько этапов пакетной и потоковой обработки. В итоге каждый производный набор обновляется и включает в себя эти вновь записанные данные. На рис. 12.1 показан пример обновления поискового индекса.

---

[1] В технологии микросервисов можно избежать синхронного сетевого запроса, кэшируя обменный курс локально в том сервисе, который обрабатывает покупку. Однако, чтобы этот кэш всегда содержал актуальные данные, необходимо периодически запрашивать изменения обменных курсов или подписаться на поток изменений — именно так и происходит в технологии потока данных.

Рис. 12.1. В поисковом индексе записи (обновления документов) соответствуют чтению (запросам)

Но зачем сразу создавать производный набор данных? Очевидно, затем, чтобы иметь возможность впоследствии направить туда запрос. Это *путь чтения*: при обслуживании пользовательского запроса, который мы считываем из производного набора данных, вероятно, придется выполнить некую дополнительную обработку результатов и сформировать ответ для пользователя.

Вместе путь записи и путь чтения формируют весь путь данных от пункта их сбора до пункта потребления (возможно, другим человеком). Путь записи — та часть пути, где данные вычисляются предварительно; другими словами, это делается максимально быстро в момент поступления данных, независимо от того, поступал ли запрос на них. Путь чтения — часть маршрута, которая выполняется только тогда, когда кто-то сделал запрос. Если вы знакомы с функциональными языками программирования, то могли заметить, что путь записи похож на упреждающие вычисления, а путь чтения — на отложенные.

Производный набор данных — точка соединения пути записи и пути чтения, как показано на рис. 12.1. Он представляет собой компромисс между объемами работы, которые должны быть выполнены во время записи и чтения соответственно.

Материализованные представления и кэширование

Хорошим примером описанного подхода является полнотекстовый поиск: путь записи — это обновление индекса, а путь чтения — поиск в индексе ключевых слов. Каждая из операций — и чтение, и запись — выполняет свою работу. При записи обновляются элементы индекса для всех терминов, представленных в документе. При чтении выполняется поиск каждого слова в запросе и с помощью булевой логики выполняется поиск документов, содержащих *все* слова в запросе (оператор *AND*), или *любого* синонима каждого из слов (оператор *OR*).

Если бы индекса не было, то поисковый запрос должен был бы сканировать все документы (как команда `grep`), что в случае большого количества документов потребовало бы значительных вычислительных затрат. Отсутствие индекса означает меньший объем работы на пути записи (индекс не обновляется), но гораздо больший — на пути чтения.

И наоборот, можно представить предварительные вычисления результатов поиска для всех вероятных запросов. В этом случае меньше работы на пути чтения: никакой булевой логики, просто найти результаты для своего запроса и выдать их пользователю. Однако путь записи будет намного дороже: набор всех возможных поисковых запросов бесконечен и, таким образом, для предварительного вычисления всех потенциальных результатов поиска потребуется неограниченное время и пространство для их хранения, так что это не будет работать[1].

[1] А вот набор различных поисковых запросов с непустыми результатами поиска выглядит не столь фантастично. Он конечен и предъявляет конечные требования по времени и памяти. Плохая новость в том, что эти требования экспоненциально зависят от числа терминов.

Другим вариантом было бы заранее вычислить результаты поиска только для фиксированного набора наиболее распространенных запросов, чтобы их можно было быстро обслуживать, не обращаясь к индексу, а нетипичные запросы все равно получать из индекса. Такой метод принято называть *кэшем* стандартных запросов, хотя мы могли бы также назвать его материализованным представлением, поскольку его необходимо обновлять при появлении новых документов, которые должны быть включены в результаты одного из типичных запросов.

Из этого примера видно, что индекс не является единственно возможной границей между путем записи и путем чтения. Можно организовать кэширование типичных результатов поиска или `grep`-подобное сканирование без индекса в случае небольшого количества документов. Таким образом, роль кэшей, индексов и материализованных представлений проста: они сдвигают границу между путем чтения и путем записи. Они позволяют выполнять основную работу на пути записи, предварительно вычисляя результаты, чтобы сэкономить усилия на пути чтения.

Сдвигание границы между работой, выполненной на пути записи и чтения, на самом деле актуально в примере с Twitter, рассмотренном в начале книги, в подразделе «Описание нагрузки» раздела 1.3. В этом примере также было показано, что граница между путями записи и чтения может быть разной для популярных и обычных пользователей. Спустя почти 500 страниц мы замкнули круг!

Клиенты, сохраняющие состояние при отключении от сети

Я считаю, что идея границы между путями записи и чтения заслуживает внимания, поскольку можно обсуждать смещение этой границы и исследовать значение данного сдвига на практике. Рассмотрим идею в другом контексте.

Огромная популярность веб-приложений за последние два десятилетия привела нас к определенным предположениям о разработке программ, легко принимаемым как должное. В частности, модель «клиент — сервер», в которой клиенты по большей части не хранят состояний, и серверы, на которых, возложена обязанность хранения и обработки данных, настолько распространена, что мы почти забываем о существовании других вариантов. Тем не менее технологии продолжают развиваться, и, я полагаю, время от времени важно подвергать сомнению статус-кво.

Традиционно браузеры были клиентами без хранения состояния, которые были полезны только при подключении к Интернету (поскольку единственное, что вы могли сделать в автономном режиме, — это прокручивать предварительно загруженную страницу). Однако современные «одностраничные» веб-приложения JavaScript получили много возможностей с поддержкой состояния, включая взаимодействие с пользовательским интерфейсом на стороне клиента и постоянное локальное хранилище в браузере. Мобильные приложения также могут хранить много состояний на устройстве и для большинства пользовательских операций не требуют обращения к серверу.

Эти новые возможности привели к возобновлению интереса к *изначально автономным приложениям*, которые по максимуму задействуют локальную базу данных, расположенную на том же устройстве, не требуя подключения к Интернету, и синхронизируются с удаленными серверами в фоновом режиме при наличии сетевого соединения [42]. Поскольку мобильные устройства часто имеют медленные и ненадежные сотовые интернет-соединения, для пользователей большим преимуществом является то, что UI не нужно ожидать синхронных сетевых запросов, а также то, что у приложения есть возможность работать в автономном режиме (см. пункт «Офлайн-клиенты» подраздела «Сценарии использования репликации с несколькими ведущими узлами» раздела 5.3).

При отказе от предположения, что клиенты без хранения состояний, которые обращаются к центральной базе данных, и переходе к идее о том, что состояния хранятся на устройствах конечных пользователей, открывается целый мир новых возможностей. В частности, это позволяет рассматривать состояние на устройстве как *кэш состояния на сервере*. Пикселы на экране являются материализованным представлением объектов модели клиентского приложения; объекты модели — локальная реплика состояния удаленного ЦОДа [27].

Передача изменений состояния клиентам

При загрузке типичной страницы в браузер и последующем изменении данных на сервере браузер не узнает об изменении, пока вы не перезагрузите страницу. Он только считывает данные в определенный момент времени, предполагая, что они являются статичными, — браузер не подписывается на обновления информации на сервере. Таким образом, состояние устройства является устаревшим кэшем, который не обновляется, если явно не проводить опрос изменений. (HTTP-протоколы подписки на канал, такие как RSS, на самом деле представляют собой всего-навсего простейшую форму опроса.)

Более новые протоколы вышли за рамки базового HTTP-шаблона «запрос — ответ»: события, отправленные сервером (eventsource API), и технология WebSocket предоставляют каналы связи, с помощью которых браузер может поддерживать открытое TCP-соединение с сервером, и последний способен активно передавать сообщения в браузер, пока он остается подключенным. Это позволяет серверу активно сообщать конечному клиенту о любых изменениях состояния, хранящегося локально, обновляя состояние на стороне клиента.

Что касается нашей модели пути записи и пути чтения, постоянное активное обновление изменений состояния на клиентских устройствах означает продление пути записи до конечного пользователя. Когда клиент инициализируется, ему все равно нужно применять путь чтения, чтобы получить начальное состояние, но после этого он может полагаться на поток изменений состояния, отправляемых сервером.

Идеи, рассмотренные в главах, посвященных обработке потоков и обмену сообщениями, не ограничиваются только применением в ЦОДах: на них можно смотреть шире и распространять до конечных пользовательских устройств [43].

Мобильные устройства некоторое время находятся в автономном режиме, и тогда они не могут получать от сервера уведомления об изменениях состояния. Но мы уже решили эту проблему: в пункте «Смещения потребителей» подраздела «Секционирование журналов» раздела 11.1 было показано, как потребитель брокера сообщений на основе журналирования может повторно подключиться после сбоя или отключения и гарантировать, что не пропустил сообщений за время отсоединения. Тот же метод работает для пользователей, где каждое устройство является небольшим подписчиком на маленький поток событий.

Сквозные потоки событий

Современные инструменты для разработки клиентских приложений с сохранением состояния и пользовательских интерфейсов для них, такие как язык Elm [30] и набор инструментальных средств React, Flux и Redux для Facebook [44], уже способны управлять внутренним состоянием на стороне клиента. Они делают это, подписываясь на потоки событий, отражающих данные, вводимые пользователем, и ответы сервера, структурированные аналогично источнику событий (см. подраздел «Источники событий» раздела 11.2).

Было бы вполне естественно расширить эту модель программирования и позволить серверу вносить события изменения состояния в данный конвейер событий на стороне клиента. Таким образом, изменения состояния могли бы проходить через сквозной путь записи: от взаимодействия на одном устройстве, которое запускает изменение состояния, через журналы событий и несколько производных информационных систем и потоковых процессоров, вплоть до пользовательского интерфейса и человека, наблюдающего за состоянием другого устройства. Эти изменения состояния могут распространяться с довольно низкой задержкой, например, проходить весь путь за секунду.

В некоторых приложениях, например в обмене мгновенными сообщениями и онлайн-играх, подобная архитектура «реального времени» уже реализована (в смысле взаимодействия с низкой задержкой, а не «гарантии времени отклика», см. одноименный пункт подраздела «Паузы при выполнении процессов» раздела 8.3). Но почему бы не строить таким образом все приложения?

Сложность в том, что представление о клиентах без хранения состояния и взаимодействиях «запрос — ответ» очень глубоко укоренилось в базах, библиотеках, структурах разработки и протоколах. Многие хранилища данных поддерживают операции чтения и записи, при которых запрос возвращает один ответ, но гораздо меньше таких, что позволяют подписываться на изменения, то есть

создавать запросы, возвращающие поток ответов, меняющийся с течением времени (см. пункт «API для потоков изменений» подраздела «Перехват изменений данных» раздела 11.2).

Чтобы продлить путь записи до конечного пользователя, необходимо принципиально переосмыслить принципы построения многих из этих систем: от взаимодействий типа «запрос — ответ» перейти к информационному потоку публикаций и подписок [27]. Я считаю, что преимущества более гибких пользовательских интерфейсов и лучшая автономная поддержка того стоят. Если вы разрабатываете информационные системы, то, надеюсь, будете учитывать возможности подписки на изменения, а не ограничитесь только запросами текущих состояний.

Чтение тоже событие

Как уже говорилось, когда потоковый процессор записывает производные данные в хранилище (базу данных, кэш или индекс) и затем пользователь делает запрос в это хранилище, то оно играет роль границы между путем записи и путем чтения. Оно обеспечивает чтение данных с произвольным доступом — в противном случае для этого потребовалось бы просканировать весь журнал событий.

Во многих случаях хранилище данных отделено от потоковой системы. Но вспомним, что потоковые процессоры также должны хранить состояние для выполнения агрегаций и объединений (см. подраздел «Объединения потоков» раздела 11.3). Это состояние обычно скрыто внутри потокового процессора, но некоторые системы разработки позволяют внешним клиентам его запрашивать [45], превращая процессор потока в своеобразную простую базу данных.

Я хотел бы развить эту идею. Как было сказано выше, операции записи в хранилище регистрируются в журнале событий, в то время как операции чтения представляют собой кратковременные сетевые запросы, которые поступают непосредственно к узлам, где хранятся запрашиваемые данные. Это разумный принцип, но, кроме него, есть и другие. Запросы чтения тоже можно представить в качестве потоков событий и пропускать через потоковый процессор как события записи, так и события чтения; в ответ на запросы чтения событий процессор будет помещать результат чтения в выходной поток [46].

Если и операции записи, и операции чтения представлять в виде событий и направлять для обработки к одному и тому же потоковому оператору, то получится фактически объединение «поток — таблица» между потоком запросов чтения и базой данных. События чтения следует направлять в раздел базы, содержащий нужную информацию (см. раздел 6.5), подобно тому как пакетные и потоковые процессоры при объединении сопоставляют данные по общему ключу (см. подраздел «Объединение и группировка на этапе сжатия» раздела 10.2).

Эта аналогия между обслуживанием запросов и выполнением объединений весьма фундаментальна [47]. Одноразовый запрос на чтение просто пропускает данные через оператор объединения, а затем сразу забывает их; запрос на подписку — это постоянное объединение с прошлыми и будущими событиями на другой стороне объединения.

Еще одно потенциальное преимущество от записи событий чтения в журнал — возможность отслеживать причинно-следственные зависимости и происхождение данных в системе. Это позволяет узнать, что увидел пользователь перед тем, как принял то или иное решение. Например, в интернет-магазине, вполне вероятно, предложенная дата доставки и наличие товара на складе, показанные клиенту, повлияют на выбор им товара для покупки [4]. Для анализа этой взаимосвязи необходимо записать результат пользовательского запроса об условиях доставки и наличии товара.

Таким образом, запись событий чтения в долговременное хранилище позволяет лучше отслеживать причинно-следственные зависимости (см. пункт «Упорядочение событий для сохранения причинно-следственных связей» подраздела «Объединение специализированных инструментов путем сбора информации» раздела 12.1), но требует дополнительных затрат на хранение и ввод-вывод информации. Оптимизация таких систем для снижения вычислительных расходов по-прежнему остается открытой исследовательской задачей [2]. Но если запросы на чтение и так регистрируются для операционных целей в качестве побочного эффекта обработки запросов, то будет не таким уж значительным изменением вместо этого сделать источником запросов журнал.

Обработка данных, хранящихся в нескольких разделах

Если запросы касаются только одного раздела, то вычислительные затраты на отправку запросов через поток с последующим сбором потока ответов, вероятно, слишком высоки. Однако эта идея открывает возможности распределенного выполнения сложных запросов, для которых необходимо объединить данные из нескольких разделов, используя преимущества инфраструктуры для маршрутизации сообщений, секционирования и объединения, предоставляемые потоковыми процессорами.

Функция распределенного RPC в Storm позволяет задействовать данный паттерн (см. пункт «Передача сообщений и RPC» подраздела «Применение обработки потоков» раздела 11.3). Например, он послужил для вычисления количества людей, которые видели в Twitter определенный URL, — другими словами, множество всех подписчиков пользователя, написавшего сообщение с этим адресом [48]. Поскольку множество пользователей этой соцсети секционировано, то такое вычисление требует объединения результатов, полученных из многих разделов.

Другой пример применения данного паттерна имеет место при предотвращении мошенничества: чтобы оценить, является ли конкретная покупка мошеннической, можно проверить показатели репутации IP-адреса пользователя, его адреса электронной почты, платежного адреса, адреса доставки и т. п. Каждая из этих баз данных репутации секционирована, поэтому для сбора показателей определенного события покупки применяется последовательность объединений с наборами данных, разделенными разными способами [49].

Схожие характеристики имеют графы выполнения внутренних запросов баз данных MPP (см. подраздел «Сравнение Hadoop и распределенных баз данных» раздела 10.2). Если нужно выполнить такое объединение нескольких разделов, то, вероятно, проще использовать БД с уже встроенной функцией, чем реализовать ее с помощью потокового процессора. Однако обработка запросов в виде потоков позволяет реализовывать широкомасштабные приложения, требования к которым выходят за пределы возможностей готовых решений.

12.3. Стремление к корректности

Если сервисы не сохраняют состояние, а только считывают данные, то сбой не имеет большого значения: чтобы вернуть систему в норму, достаточно исправить ошибку и перезапустить сервис. Системы с хранением состояний наподобие баз данных не так просты: они предназначены для запоминания информации навсегда (более или менее) и в случае неполадок последствия тоже могут длиться вечно. А это значит, что они требуют более тщательной проработки [50].

Мы хотим создавать надежные и *корректные* приложения (программы, семантика которых четко определена и понятна даже в случае различных ошибок). В течение последних 40 лет основой построения корректных приложений служили такие свойства транзакций, как атомарность, изоляция и долговечность (см. главу 7). Однако этот фундамент слабее, чем кажется: вспомним хотя бы слабые уровни изоляции (см. раздел 7.2).

В отдельных областях применения от транзакций полностью отказываются, заменяя их моделями, имеющими более высокую производительность и масштабируемость, но семантика их гораздо менее понятна (см., например, раздел 5.4). О *согласованности* много говорят, но ей так и не дали четкого определения (см. пункт «Согласованность» подраздела «Смысл аббревиатуры ACID» раздела 7.1 и главу 9). Некоторые утверждают, что следует «смириться со слабой согласованностью» ради лучшей доступности, не имея четкого представления о том, что это значит на практике.

Наше понимание и инженерные методы в этой столь важной области удивительно фрагментарны. Например, очень сложно определить, безопасно ли запускать конкретное приложение на определенном уровне изоляции транзакции или конфигурации

репликации [51, 52]. Часто обнаруживается, что простые решения работают корректно при незначительном количестве параллельных процессов и отсутствии сбоев, но в более сложных условиях выдают много труднообнаруживаемых ошибок.

Например, эксперименты Кайла Кингсбери (Kyle Kingsbury) с Jepsen [53] выявили резкие расхождения между заявленными гарантиями безопасности некоторых продуктов и их фактическим поведением при наличии сетевых проблем и сбоев. Даже если инфраструктурные продукты, такие как базы данных, не создают подобных проблем, приложение все равно должно правильно использовать предоставляемые ему функции. И в этот момент возможны ошибки в случае трудностей с пониманием конфигурации (что имеет место при слабых уровнях изоляции, конфигурациях кворума и т. п.).

Если приложение способно справляться с тем, что данные иногда бывают повреждены или потеряны непредсказуемым образом, то жизнь становится намного проще и можно уйти, просто скрестив пальцы, и надеяться на лучшее. Но при необходимости иметь более прочные гарантии корректной работы следует использовать сериализируемость и атомарные фиксации. Однако это требует больших вычислительных затрат: обычно они работают в общем ЦОДе (что исключает применение географически распределенной архитектуры), ограничивают масштабируемость и отказоустойчивость.

Несмотря на присутствие традиционной технологии транзакций, я также полагаю, что это не последнее слово в области разработки корректных и устойчивых к сбоям приложений. В текущем разделе я предложу несколько способов представления о корректности в контексте архитектуры потоков данных.

Сквозные аргументы в базе данных

То обстоятельство, что приложение использует информационную систему, обеспечивающую сравнительно высокую надежность, — например, сериализируемые транзакции, — еще не означает гарантию от потери или повреждения данных. Например, если в коде приложения есть ошибка, которая приводит к записи некорректных данных или удалению данных из базы, то сериализируемые транзакции не помогут.

Приведенный пример может показаться легкомысленным, но к нему стоит отнестись серьезно: ошибки приложений нередки и люди тоже делают ошибки. Я использовал этот пример в подразделе «Состояние, потоки и неизменяемость» раздела 11.2 как аргумент в пользу неизменяемых данных и журналов с разрешением дописывания — с их помощью легче исправлять такие ошибки после того, как код будет исправлен и больше не сможет уничтожать нужные данные.

Неизменяемость полезна, но сама по себе не является панацеей. Рассмотрим более тонкий пример возможного повреждения данных.

Однократное выполнение операций

В подразделе «Отказоустойчивость» раздела 11.3 нам встретилась идея, которую мы назвали семантикой *однократного* (или *эффективно-однократного*) выполнения задачи. Если во время обработки сообщения что-то пойдет не так, то вы можете либо отказаться от результата (отбросить сообщение, потерять данные), либо повторить попытку. При повторной попытке выполнить задачу существует вероятность, что в действительности предыдущая попытка была успешной, но вы об этом просто не узнали и, как следствие, сообщение будет обработано дважды.

Двойная обработка — форма искажения данных: нежелательно заставлять клиента дважды платить за один и тот же сервис (или выставлять двойной счет) или два раза увеличивать значение счетчика (завышая отдельную метрику). В этом контексте *однократное выполнение* означает организацию вычислений, при которой результат будет таким же, как если бы никаких ошибок не произошло, пусть даже операция фактически была повторена вследствие некоторой неисправности. Ранее мы обсудили несколько вариантов достижения данной цели.

Один из наиболее эффективных подходов — превратить операцию в *идемпотентную* (см. пункт «Идемпотентность» подраздела «Отказоустойчивость» раздела 11.3), то есть сделать так, чтобы ее результат не зависел от того, сколько раз выполняется операция — один или несколько. Однако если операция по своей природе не является идемпотентной, то для того, чтобы сделать ее таковой, необходимы определенные усилия и затраты: может потребоваться сохранить некоторые дополнительные метаданные (например, набор идентификаторов операций, обновивших значение) и обеспечить границы при переходе от одного узла к другому (см. пункт «Ведущий узел и блокировки» подраздела «Истина определяется большинством» раздела 8.4).

Подавление дублирования

Паттерн отбрасывания дубликатов встречается не только в потоковой обработке, но и во многих других местах. Например, в TCP используются порядковые номера пакетов — для размещения последних в правильном порядке на стороне получателя и определения того, были ли какие-либо пакеты потеряны или дублированы при прохождении по сети. Все потерянные передаются повторно, а все дубликаты отбрасываются стеком TCP, прежде чем он передает данные приложению.

Однако такое подавление дублирования работает только в контексте одного TCP-соединения. Представьте, что будет, если TCP-соединение является соединением клиента с базой и выполняется транзакция, показанная в примере 12.1. Во мно-

гих БД транзакция привязана к клиентскому соединению (при отправке клиентом нескольких запросов база данных знает, что они относятся к одной транзакции, поскольку отправлены по одному и тому же TCP-соединению). Если клиент прерывает соединение с сетью и время ожидания соединения истекает после отправки `COMMIT`, но до получения ответа от сервера базы данных, то он не знает, была ли транзакция завершена или прервана (см. рис. 8.1).

Пример 12.1. Неидемпотентная передача денег с одного счета на другой

```
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance + 11.00 WHERE account_id = 1234;
UPDATE accounts SET balance = balance - 11.00 WHERE account_id = 4321;
COMMIT;
```

Клиент может заново подключиться к базе данных и повторить транзакцию, но теперь она выходит за рамки подавления дубликатов TCP. Поскольку транзакция, показанная в примере на рис. 12.1, не является идемпотентной, может случиться так, что вместо желаемой суммы $11 будет переведено $22. Таким образом, хотя на рис. 12.1 показан стандартный пример атомарной транзакции, на самом деле это неверно и реальные банки так не работают [3].

Протоколы двухфазной фиксации (см. подраздел «Атомарная и двухфазная фиксация (2PC)» раздела 9.4) нарушают соотношение 1:1 между TCP-соединением и транзакцией, поскольку приходится разрешить координатору транзакции повторное подключение к базе данных после сетевого сбоя и указать, следует ли принять или прервать транзакцию. Достаточно ли этого, чтобы транзакция была выполнена только один раз? К сожалению, нет.

Даже при наличии возможности подавить дублирование транзакций между клиентом и сервером базы данных нам все равно нужно побеспокоиться о сети между конечным пользователем и сервером приложений. Например, если конечный пользователь является браузером, то для отправки инструкции на сервер он, вероятно, применяет запрос HTTP POST. Вероятно, у пользователя слабое сотовое соединение и ему удалось отправить POST-запрос, но получить ответ от сервера он не смог, так как в этот момент сигнал стал слишком слабым.

В подобном случае пользователю, как вариант, будет показано сообщение об ошибке и он сможет повторить попытку вручную. Браузеры обычно предупреждают: «Вы действительно хотите отправить эту форму еще раз?» — и пользователь говорит «да», потому что ему нужно выполнить данную операцию. (При нормальной работе в паттерне Post/Redirect/Get [54] такого предупреждающего сообщения нет, но это не помогает, если срок запроса POST истечет.) С точки зрения веб-сервера повторение является отдельным запросом, а с точки зрения БД — отдельной транзакцией. Обычные механизмы дедупликации здесь не помогают.

Идентификаторы операций

Чтобы сделать операцию идемпотентной при нескольких переходах по сети, недостаточно полагаться только на механизм транзакций, предоставляемый базой данных, — необходимо *сквозное* прохождение запроса.

Например, можно создать уникальный идентификатор операции (например, UUID) и сделать его скрытым полем формы в клиентском приложении или получить значение идентификатора операции, вычислив хеш всех соответствующих полей формы [3]. Если браузер дважды отправит POST-запрос, то у этих запросов будут одинаковые идентификаторы операции. Затем можно передать идентификатор операции в базу и убедиться, что выполнена только одна операция с данным идентификатором, как показано в примере 12.2.

Пример 12.2. Подавление повторяющихся запросов с использованием уникального идентификатора

```
ALTER TABLE requests ADD UNIQUE (request_id);

BEGIN TRANSACTION;

INSERT INTO requests
  (request_id, from_account, to_account, amount)
  VALUES('0286FDB8-D7E1-423F-B40B-792B3608036C', 4321, 1234, 11.00);

UPDATE accounts SET balance = balance + 11.00 WHERE account_id = 1234;
UPDATE accounts SET balance = balance - 11.00 WHERE account_id = 4321;

COMMIT;
```

В примере 12.2 используется ограничение уникальности в столбце `request_id`. Если транзакция пытается вставить идентификатор, который уже существует, то `INSERT` не срабатывает и транзакция прерывается, так что она не будет выполнена дважды. Реляционные базы данных обычно поддерживают ограничение уникальности даже при слабых уровнях изоляции (тогда как на уровне приложения операция проверки перед вставкой при несериализируемой изоляции может быть неудачной, как показано в подразделе «Асимметрия записи и фантомы» раздела 7.2).

Кроме подавления повторяющихся запросов, таблица `requests` в примере 12.2 играет роль своеобразного журнала, двигаясь в сторону источника событий (см. подраздел «Источники событий» раздела 11.2). Обновление балансов учетной записи на самом деле не должно происходить в той же транзакции, что и вставка события, поскольку обновления избыточны и могут быть получены следующим клиентом-потребителем из события запроса — в том случае, если событие обрабатывается только один раз, а это можно гарантировать, используя идентификатор запроса.

Сквозной аргумент

Этот сценарий подавления дублирующихся транзакций — лишь один из примеров более общего принципа, называемого *сквозным аргументом*, который был сформулирован Солтцером (Saltzer), Ридом (Reed) и Кларком (Clark) в 1984 году [55].



Эта функция может быть полностью правильно реализована только с учетом знаний и с помощью приложения, стоящего в конечных точках системы связи. Таким образом, реализовать данную функцию как функцию самой системы связи нельзя. (Иногда для повышения производительности потенциально полезна неполная версия этой функции, предоставляемая коммуникационной системой.)

В нашем примере *такой функцией* было подавление дублирования. Мы увидели, что TCP подавляет дублирование пакетов на уровне TCP-соединения, а некоторые потоковые процессоры обеспечивают так называемую семантику однократного выполнения на уровне обработки сообщений. Но этого недостаточно, чтобы пользователь не отправлял повторные запросы, если время первого запроса истекло. Сами по себе протокол TCP, транзакции базы данных и потоковые процессоры не могут полностью исключить эти дубликаты. Для устранения проблемы требуется комплексное решение: идентификатор транзакции, передаваемый от конечного пользователя в БД.

Сквозные аргументы также применяются для проверки целостности данных: контрольные суммы, встроенные в протоколы Ethernet, TCP и TLS, способны обнаруживать повреждение пакетов в сети, но не находят повреждения, вызванные ошибками программного обеспечения на этапе отправки и получения или повреждениями дисков, где хранятся данные. Чтобы отследить все возможные источники повреждения данных, необходимы сквозные контрольные суммы.

Подобный аргумент применим и к шифрованию [55]: пароль в вашей домашней сети Wi-Fi защищает от людей, отслеживающих Wi-Fi-трафик, но не от злоумышленников, расположенных в другой точке Интернета; соединение TLS/SSL между клиентом и сервером защищает от сетевых злоумышленников, но не от компрометаций самого сервера. Здесь помогают только сквозное шифрование и аутентификация.

Хотя низкоуровневые функции (подавление дубликатов в TCP, контрольные суммы в Ethernet, шифрование в Wi-Fi) сами по себе не обеспечивают желаемые сквозные функции, они по-прежнему полезны, поскольку уменьшают вероятность проблем на более высоких уровнях. Например, HTTP-запросы часто оказываются поврежденными, если не был использован протокол TCP, размещающий пакеты в правильном порядке. Просто нужно помнить, что низкоуровневые функции надежности сами по себе недостаточны для обеспечения сквозной корректности.

Применение сквозных аргументов в информационных системах

Это возвращает нас к первоначальному тезису: один лишь факт, что приложение использует информационную систему, обеспечивающую сравнительно высокую безопасность, например сериализируемые транзакции, еще не означает гарантию защиты от потери или повреждения данных. В самом приложении необходимо применять сквозные аргументы, такие как подавление дублирования.

Это позор, что механизмы отказоустойчивости настолько трудно реализовать. Низкоуровневые механизмы надежности, такие как в TCP, работают достаточно хорошо, вследствие чего оставшиеся ошибки более высокого уровня случаются довольно редко. Было бы неплохо обернуть оставшиеся механизмы отказоустойчивости высокого уровня в абстракции с целью не описывать их в коде приложения, но я боюсь, что мы еще не нашли подходящую абстракцию.

Долгое время хорошей абстракцией считались транзакции, и я уверен в их действительной пользе. Как обсуждалось во введении к главе 7, транзакции решают широкий круг возможных задач (конкурентная запись, нарушение ограничений, сбои, сетевые прерывания, ошибки дисков) и сводят их к двум вероятным результатам: принять или отбросить. Это огромное упрощение модели программирования, но я боюсь, что его недостаточно.

Транзакции требуют больших вычислительных затрат, особенно когда связаны с гетерогенными технологиями хранения (см. подраздел «Распределенные транзакции на практике» раздела 9.4). В случае нашего отказа от использования распределенных транзакций вследствие их дороговизны нам приходится переопределять механизмы отказоустойчивости в коде приложения. Как показали многочисленные примеры, представленные в настоящей книге, рассуждения о конкурентном доступе и частичных сбоях сложны и противоречивы, поэтому я подозреваю, что большинство механизмов на уровне приложений работает неправильно. Результат — потеря или повреждение данных.

По этим причинам я считаю, что стоит исследовать абстракции отказоустойчивости, которые облегчили бы разработку специальных функций для конечных приложений, а также имели бы высокую производительность и хорошие эксплуатационные характеристики в крупномасштабной распределенной среде.

Принудительные ограничения

Подумаем о корректности в контексте разделения баз данных (раздел 12.2). Мы видели, что сквозное подавление дублирования может быть достигнуто с помощью идентификаторов запроса, которые проходят весь путь от клиента до базы, где фиксируются операции записи. Как насчет других ограничений?

В частности, обратим внимание на ограничения уникальности, в частности те, что были задействованы в примере 12.2. В пункте «Ограничения и гарантии уникальности» подраздела «Опора на линеаризуемость» раздела 9.2 было представлено еще несколько примеров свойств приложения, обеспечивающих уникальность: имя пользователя или адрес электронной почты однозначно идентифицируют его, в системе хранения файлов не может быть нескольких файлов с одинаковыми именами, и два человека не могут забронировать одно и то же место в самолете или театре.

Другие виды ограничений подобны этим: например, баланс на счету не может быть отрицательным; нельзя продать больше товаров, чем есть на складе; нельзя два раза продать одно и то же место в зале. Методы, обеспечивающие уникальность, часто могут использоваться и для таких ограничений.

Ограничения уникальности требуют согласованности

В главе 9 мы увидели, что в распределенной среде принудительное ограничение уникальности требует согласованности: если существует несколько конкурентных запросов с одинаковым значением, система каким-то образом должна решить, какая из конфликтующих операций будет принята, и отклонить остальные, как нарушающие ограничение.

Самый распространенный способ достижения такой согласованности — назначить один узел ведущим и возложить на него ответственность за принятие решений. Это отлично работает, если вы согласны направлять все запросы на один узел (даже когда клиент находится на другой стороне земного шара) и до тех пор, пока на данном узле не случится сбой. При необходимости же застраховаться от сбоя ведущего узла мы снова возвращаемся к проблеме согласованности (см. пункт «Репликация с одним ведущим узлом и консенсус» подраздела «Отказоустойчивый консенсус» раздела 9.4).

Проверка уникальности может быть масштабирована путем секционирования по значению, которое должно быть уникальным. Например, для обеспечения уникальности по идентификатору запроса, как в примере 12.2, все запросы с одинаковым идентификатором направляются в один раздел (см. главу 6). Если нужно, чтобы имена пользователей были уникальными, можно строить секционирование по хешу имени пользователя.

Однако асинхронная репликация с несколькими участниками исключена, поскольку существует вероятность того, что разные участники одновременно примут конфликтующие записи и, следовательно, их значения перестанут быть уникальными (см. подраздел «Реализация линеаризуемых систем» раздела 9.2). Чтобы иметь возможность немедленно отклонять любые записи, нарушающие ограничение, необходима синхронная координация [56].

Уникальность в сообщениях на основе журналирования

Ведение журнала гарантирует, что все пользователи видят сообщения в одинаковой последовательности — эта гарантия официально известна как рассылка общей последовательности и эквивалентна согласованности (см. подраздел «Рассылка общей последовательности» раздела 9.3). В разделенной базе данных с обменом сообщениями на основе журналирования можно задействовать подобный подход для обеспечения ограничений уникальности.

Потоковый процессор потребляет все сообщения из журнала данного раздела последовательно, в одном потоке (см. пункт «Что лучше: журналы или традиционный обмен сообщениями?» подраздела «Секционирование журналов» раздела 11.1). Таким образом, при секционировании журнала по значению, которое должно быть уникальным, потоковый процессор может однозначно решить, какая из нескольких конфликтующих операций была первой. Например, если несколько пользователей пытаются получить одно и то же имя, то выполняется такой алгоритм [57].

1. Каждый запрос имени пользователя кодируется как сообщение и добавляется к разделу, соответствующему хешу имени пользователя.
2. Потоковый процессор последовательно считывает запросы из журнала, применяя локальную базу данных, чтобы отслеживать, какие имена пользователей уже существуют. Для каждого запроса имени, которое еще доступно, процессор записывает имя как принятое и передает в выходной поток сообщение об успешном завершении операции. Если же имя пользователя занято, то он отправляет в выходной поток сообщение об отказе.
3. Клиент, который запросил имя пользователя, следит за выходным потоком и ждет сообщения об успехе или отказе, соответствующие его запросу.

В целом данный алгоритм такой же, как в пункте «Реализация линеаризуемого хранилища с помощью рассылки общей последовательности» подраздела «Рассылка общей последовательности» раздела 9.3. Он легко масштабируется для высокой пропускной способности запросов путем увеличения количества разделов, так как каждый раздел обрабатывается независимо от других.

Этот метод работает не только для ограничений уникальности, но и для многих других видов ограничений. Его основной принцип заключается в том, что любые потенциально конфликтующие записи направляются в один раздел и там обрабатываются последовательно. Как обсуждалось в пункте «Что такое конфликт» подраздела «Обработка конфликтов записи» раздела 5.3 и в подразделе «Асимметрия записи и фантомы» раздела 7.2, определение конфликта зависит от конкретного

приложения, но потоковый процессор может использовать произвольную логику для проверки запроса. Данная идея похожа на подход, впервые предложенный Байу (Bayou) в 1990-х годах [58].

Обработка запросов в нескольких разделах

Когда задействовано несколько разделов, задача обеспечения атомарности операции с учетом ограничений становится более интересной. В примере 12.2 потенциально есть три раздела: в одном хранится идентификатор запроса, во втором — счет получателя и в третьем — счет плательщика. Нет причин, по которым эти три показателя должны находиться в одном разделе, поскольку все они независимы друг от друга.

В традиционных базах данных выполнение этой транзакции потребует атомарной фиксации во всех трех разделах, что принудительно ставит ее в общую очередь всех транзакций во всех остальных секциях. Поскольку сейчас существует межсегментная координация, различные разделы больше не могут обрабатываться независимо, поэтому пропускная способность, вероятно, упадет.

Однако оказывается, что такая же корректность может быть достигнута без атомарной фиксации, с помощью секционированных журналов.

1. Запрос на перевод денег со счета А на счет Б получает от клиента уникальный идентификатор и добавляется в журнал раздела на основе этого ID.
2. Потоковый процессор читает журнал запросов. Для каждого сообщения запроса он отправляет в выходные потоки два сообщения: инструкцию дебетования для учетной записи плательщика А (секционируется по А) и инструкцию кредитования для учетной записи получателя Б (секционируется по Б). В оба исходящих сообщения включается идентификатор исходного запроса.
3. Другие процессоры потребляют потоки инструкций кредитования и дебетования, подавляют дублирование по идентификатору запроса и применяют изменения к балансам учетных записей.

Пункты 1 и 2 необходимы, поскольку если клиент отправил инструкции по кредиту и дебетованию непосредственно, то потребуется атомарная фиксация между этими двумя разделами, чтобы гарантировать либо выполнение обеих операций, либо отсутствие их совершения. Во избежание необходимости распределенной транзакции мы сначала регистрируем запрос как одно сообщение в журнале долговременного хранения, а затем на его основе генерируем инструкции по кредитованию и дебетованию. Запись одного объекта является атомарной почти во всех информационных системах (см. пункт «Однообъектные операции записи» подраздела «Однообъектные и многообъектные операции» раздела 7.1), поэтому запрос или появляется в журнале, или нет, без необходимости атомарной фиксации для нескольких разделов.

Если на шаге 2 потоковый процессор выйдет из строя, то возобновит обработку, начиная с последней контрольной точки. При этом он не пропустит никаких сообщений о запросах, но может обработать один запрос несколько раз и создать дубликаты инструкций по кредитованию и дебетованию. Однако, поскольку он является детерминированным, просто будет генерировать снова одни и те же инструкции и процессоры на шаге 3 смогут легко подавить дублирование, используя сквозной идентификатор запроса.

Для гарантий того, что в ходе данной операции на счету плательщика не будет превышен кредит, можно создать дополнительный процессор потока (с секционированием по номеру счета плательщика), который обслуживал бы балансы счетов и проверял транзакции. В этом случае на шаге 1 в журнал запросов будут помещаться только проверенные транзакции.

Разбивая транзакцию, затрагивающую несколько разделов, на два этапа, каждый из которых имеет дело со своим разделом, и используя сквозной идентификатор запроса, мы получаем такой же уровень корректности (каждый запрос выполняется только один раз, как для счета плательщика, так и для счета получателя платежа), даже в случае сбоев и без применения протокола атомарной фиксации. Идея задействовать несколько этапов с разным секционированием аналогична той, что обсуждалась в пункте «Обработка данных, хранящихся в нескольких разделах» подраздела «Наблюдение за производными состояниями» раздела 12.2, см. также пункт «Контроль конкурентных процессов» подраздела «Состояние, потоки и неизменяемость» раздела 11.2).

Своевременность и целостность

Удобное свойство транзакций состоит в том, что они обычно линеаризуемы (см. раздел 9.2): при записи сначала завершается транзакция, и только после этого ее результаты становятся доступны для чтения.

Это не разделение операции на нескольких этапах потоковых процессоров: потребители журнала асинхронны по своей архитектуре, вследствие чего отправитель не ждет, пока его сообщение будет обработано потребителями. Однако клиент может ожидать появления сообщения в потоке вывода. Речь о том, что было показано в пункте «Уникальность в сообщениях с журналированием» подраздела «Принудительные ограничения» текущего раздела при проверке ограничения уникальности.

В этом примере корректность проверки уникальности не зависит от того, ожидает ли отправитель сообщения выходного результата. Единственная цель ожидания — синхронно информировать отправителя о том, подтвердила ли проверка уникальность, но данное уведомление отделено от результатов обработки сообщения.

В более общем плане, я думаю, понятие *согласованности* объединяет два разных требования, заслуживающих особого внимания.

- *Своевременность*. Означает, что пользователи видят систему в ее актуальном состоянии. Ранее уже упоминалось: если пользователь читает устаревшую копию данных, то они могут поступать с нарушением последовательности (см. раздел 5.2). Однако эта несогласованность носит временный характер и в итоге будет устранена просто путем ожидания и повторения попытки.

  В теореме CAP (см. подраздел «Цена линеаризуемости» раздела 9.2) согласованность используется в смысле линеаризуемости, что является надежным способом достижения своевременности. Кроме того, могут быть полезны более слабые варианты своевременности, такие как согласованность типа *«чтение после записи»* (см. подраздел «Читаем свои же записи» раздела 5.2).
- *Целостность*. Означает отсутствие повреждения данных: потерь, противоречивых или ложных данных. В частности, если некий производный набор данных является представлением тех или иных базовых данных (см. пункт «Получение текущего состояния из журнала событий» подраздела «Источники событий» раздела 11.2), то производные данные должны быть корректными. Например, индекс БД должен правильно отражать ее содержимое — от индекса, в котором отсутствуют отдельные записи, мало пользы.

  При нарушении целостности несогласованность становится постоянной: ожидание и повторная попытка в большинстве случаев не исправят повреждение базы данных, нужны явная проверка и восстановление. В контексте транзакций ACID (см. подраздел «Смысл аббревиатуры ACID» раздела 7.1) согласованность обычно понимается как целостность в рамках приложения. Важными инструментами для сохранения последней являются атомарность и долговечность.

Сформулируем кратко: нарушение своевременности — «согласованность иногда», а нарушение целостности — «вечная несогласованность».

Я намерен утверждать, что в большинстве приложений целостность гораздо важнее, чем своевременность. Нарушения своевременности раздражают и вводят в заблуждение, но нарушения целостности могут иметь катастрофические последствия.

Например, в случае с кредитной картой неудивительно, что транзакция, сделанная за последние 24 часа, еще не появилась. Это нормально, у систем есть определенное отставание. Мы знаем, что банки сводят и обрабатывают транзакции асинхронно, а своевременность здесь не очень важна [3]. Однако было бы очень плохо, если бы баланс не был равен сумме транзакций плюс предыдущий баланс (ошибка в суммах) или если бы платеж был списан с вашего счета, но не получен продавцом (исчезающие деньги). Именно к таким проблемам приводят нарушения целостности системы.

Корректность потоков данных

Транзакции ACID обычно обеспечивают как своевременность (линеаризуемость), так и целостность (атомарную фиксацию). Таким образом, если рассматривать корректность приложений с точки зрения транзакций ACID, то различие между своевременностью и целостностью весьма несущественно.

Впрочем, интересное свойство систем информационных потоков, основанных на событиях, обсуждавшихся в этой главе, заключается в том, что в них своевременность и целостность разделены. При асинхронной обработке потоков событий гарантии своевременности отсутствуют, если только явно не созданы потребители, ожидающие сообщений, которые должны появиться перед завершением работы. Однако целостность в потоковых системах является необходимым условием.

Семантика *однократного* или *эффективно-однократного* выполнения (см. подраздел «Отказоустойчивость» раздела 11.3) является механизмом сохранения целостности. Если событие потеряно или происходит дважды, то целостность информационной системы может быть нарушена. Таким образом, отказоустойчивая доставка сообщений и подавление дублирования (например, с помощью идемпотентных операций) важны для обеспечения целостности информационной системы в случае сбоя.

Как мы увидели выше, надежные системы потоковой обработки способны сохранять целостность, не требуя распределенных транзакций и протокола атомарной фиксации. Это значит, что потенциально они могут достичь сопоставимой корректности при гораздо большей производительности и эксплуатационной надежности. Такая целостность достигается благодаря сочетанию следующих технологий.

- Представление содержимого операции записи в качестве отдельного сообщения, которое может быть легко записано атомарно, — метод, весьма подходящий для поиска событий (см. подраздел «Источники событий» раздела 11.2).
- Получение всех обновлений состояния из одного сообщения с помощью детерминированных производных функций, аналогично хранимым процедурам (см. подраздел «По-настоящему последовательное выполнение» раздела 7.3 и пункт «Программный код как функция построения вторичных данных» подраздела «Проектирование приложений на основе потока данных» раздела 12.2).
- Передача генерируемого клиентом идентификатора запроса через все уровни обработки, что обеспечивает сквозное подавление дублирования и идемпотентность.
- Неизменяемые сообщения и возможность повторно генерировать производные данные, что упрощает восстановление после выявления ошибок (см. пункт «Преимущества неизменяемых событий» подраздела «Состояние, потоки и неизменяемость» раздела 11.2).

Сочетание этих методов представляется мне очень перспективным направлением, которое позволит в будущем разрабатывать отказоустойчивые приложения.

Недостаточные ограничения

Как обсуждалось ранее, принудительное ограничение уникальности требует согласованности, обычно реализуемой с помощью объединения всех событий в одном разделе и пропуска их через один узел. При традиционной форме ограничения уникальности такое ограничение неизбежно, и потоковая обработка не поможет его избежать.

Однако следует понять еще одну вещь: многим реальным приложениям достаточно гораздо более слабых требований уникальности.

- Если два человека конкурентно регистрируют одно и то же имя пользователя или заказывают одно и то же место, можно отправить одному из них сообщение с извинениями и предложением выбрать другое имя или место. Такое изменение для исправления ошибки называется *компенсационной транзакцией* [59, 60].
- Если покупатели хотят купить больше товаров, чем есть на складе, то можно заказать больше товара, извиниться перед покупателями за задержку и предложить им скидку. Это, в сущности, то же самое, как если бы, например, погрузчик проехал по нескольким товарам на складе, повредив их и оставив меньше товара, чем предполагалось [61]. Процедура принесения извинений в любом случае должна быть частью бизнес-процессов, поэтому нет необходимости требовать линеаризуемое ограничение количества товара на складе.
- Подобным образом многие авиакомпании резервируют больше мест, чем есть в самолете, в расчете на то, что некоторые пассажиры пропустят вылет, а отели бронируют номера, ожидая отмены бронирования отдельными гостями. В этих случаях ограничение «один человек — одно место» намеренно нарушается по коммерческим соображениям, и применяются компенсационные процессы (возврат денег, обновление, предоставление бесплатной комнаты в соседнем отеле) для разрешения ситуаций, когда спрос превышает предложение. Даже при отсутствии избыточного бронирования извинения и компенсационные процессы все равно нужны на случай отмены рейсов из-за плохой погоды или забастовки персонала. Решение таких проблем — обычная часть бизнеса [3].
- Если кто-то снял больше денег, чем было на счету, то банк может взять комиссию за превышение лимита и потребовать возвратить долг. Вводя ограничение на снятие денег со счета в день, банк ограничивает свой риск.

Во многих бизнес-контекстах фактически приемлемо временно нарушать ограничения и исправлять их позже, принося извинения. Стоимость извинений (с точки зрения денег или репутации) бывает разной, но обычно довольно низкая: нельзя вернуть письмо, отправленное по электронной почте, но можно отправить вслед за ним другое письмо с исправлениями. Если случайно пополнить кредитную карту дважды, то можно отменить одно из пополнений, заплатив за это лишь стоимостью обработки и, вероятно, жалобой клиента. После того как деньги покинули пределы банкомата, нельзя напрямую вернуть их, хотя, в принципе,

можно отправить к клиенту сборщиков долгов, если лимит счета был превышен и клиент не собирается ничего возвращать.

Независимо от того, приемлема ли стоимость извинений, это коммерческое решение. Если она приемлема, то традиционная модель проверки всех ограничений перед записью данных является слишком строгой, а линеаризуемое ограничение не требуется. Возможно, разумный выбор — понадеяться на лучшее и сделать запись, а затем проверить ограничение. Проверка все выполняется, прежде чем будут выполнены операции, отмена которых обходится по-настоящему дорого. Но это не значит, что следует делать проверку перед обычной записью данных.

Подобные приложения *все равно* требуют целостности: вы вряд ли захотите потерять бронирование или деньги из-за несоответствующих операций кредитования и дебетования. Но они *не требуют* своевременности при принуждении к ограничению: если продать больше товара, чем есть на складе, то можно исправить проблему постфактум, принеся извинения. Это похоже на методы разрешения конфликтов, рассмотренные в подразделе «Обработка конфликтов записи» раздела 5.3.

Информационные системы без координации

Мы выявили два следующих интересных наблюдения.

1. Системы информационных потоков способны гарантировать целостность производных данных без атомарной фиксации, линеаризуемости или синхронной координации между разделами.
2. Строгие ограничения уникальности требуют своевременности и координации, но на практике многие приложения отлично работают с недостаточными ограничениями, которые могут быть временно нарушены и исправлены позже, при условии, что целостность сохраняется на всем протяжении процесса.

Сведенные вместе, эти наблюдения означают, что системы информационных потоков позволяют предоставлять сервисы управления данными для многих приложений, не требуя координации, но притом обеспечивая надежные гарантии целостности. Такие информационные системы *без координации* очень привлекательны: они обеспечивают более высокую производительность и отказоустойчивость, чем системы с синхронной координацией [56].

Например, такая система может быть распределенной между несколькими ЦОДами и иметь конфигурацию с несколькими ведущими узлами, асинхронно реплицируя данные между регионами. Любой центр способен работать независимо от других, потому что не требуется синхронная координация между регионами. Такая система будет иметь слабые гарантии своевременности — без координации она не может быть линеаризуемой, но по-прежнему будет иметь надежные гарантии целостности.

В этом контексте сериализируемые транзакции по-прежнему полезны как часть обслуживания производных состояний, но могут выполняться в той небольшой области, где хорошо работают [8]. Неоднородные распределенные транзакции, такие как транзакции XA (см. подраздел «Распределенные транзакции на практике» раздела 9.4), не требуются. Синхронную координацию можно по-прежнему применять по мере необходимости (например, для обеспечения строгих ограничений перед операциями, восстановление после которых невозможно), но не стоит платить за координацию всего подряд, если в ней нуждается только небольшая часть приложения [43].

Еще одна точка зрения на координацию и ограничения: они уменьшают количество извинений, которые приходится приносить из-за несоответствий, но потенциально снижают производительность и доступность системы и, следовательно, увеличивают количество извинений, которые приходится приносить из-за сбоев в работе. Нельзя свести количество извинений до нуля, но можно стремиться к наилучшему компромиссу для конкретного приложения — оптимальному варианту, где не слишком много нестыковок и мало проблем с доступностью.

Доверяй, но проверяй

До сих пор все наши рассуждения о корректности, целостности и отказоустойчивости были основаны на предположении, что некий один момент может пойти не так, но все остальное будет в порядке. Такие допущения называют *моделью систем* (см. пункт «Привязка моделей систем к реальному миру» подраздела «Модели системы на практике» раздела 8.4): например, нам приходится предположить, что процессы могут давать сбои, машины — внезапно выключаться, а сеть — произвольно задерживать или удалять сообщения. Но мы также можем предположить, что данные, записанные на диск, не потеряются после выполнения команды `fsync`, данные в памяти не повреждены и команда умножения в процессоре всегда возвращает правильный результат.

Такие предположения вполне разумны, поскольку большую часть времени они справедливы. Было бы трудно сделать что-либо, если бы мы постоянно беспокоились о вероятной ошибке компьютера. Традиционно модели систем принимают двоякий подход к ошибкам: мы предполагаем, что некоторые вещи могут случиться, а другие — никогда не произойдут. На самом деле это скорее вопрос вероятности: одни события более вероятны, другие — менее. Вопрос заключается в том, достаточно ли часто наши предположения нарушаются на практике.

Мы увидели, что данные могут быть повреждены во время хранения на дисках (см. врезку «Репликация и сохраняемость» в пункте «Сохраняемость» подраздела «Смысл аббревиатуры ACID» раздела 7.1), а повреждение данных в сети иногда способно остаться незамеченным даже с помощью контрольных сумм TCP

(см. пункт «Слабые формы “лжи”» подраздела «Византийские сбои» раздела 8.4). Может быть, этому стоит уделять больше внимания?

Однажды я работал над приложением, которое собирало отчеты о сбоях от клиентов. Часть полученных отчетов объяснялась только случайными битами в памяти устройств. Это может показаться маловероятным, но если программное обеспечение установлено на достаточно большом количестве устройств, то даже очень маловероятные вещи случаются. Помимо случайного повреждения памяти из-за аппаратных сбоев или излучений, возможны неверные способы доступа к памяти, способные изменять биты даже при отсутствии дефектов у самой памяти [62] — этот эффект используют для взлома механизмов безопасности в операционных системах [63] (метод, известный как *rowhammer*). Стоит присмотреться внимательнее, и станет ясно, что оборудование — совсем не такая идеальная абстракция, как может показаться.

Проясним ситуацию: случайные переключения битов в современном оборудовании встречаются очень редко [64]. Я лишь хочу указать, что они не являются невозможными и поэтому заслуживают внимания.

Обеспечение целостности при ошибках программного обеспечения

Кроме аппаратных проблем, всегда есть риск ошибок программного обеспечения, которые не улавливаются на более низких уровнях сети, памяти или контрольных сумм файловой системы. Даже у такого широко используемого программного обеспечения, как БД, есть ошибки: я лично видел случаи, когда в MySQL неправильно поддерживалось ограничение уникальности [65], а в PostgreSQL на уровне изоляции сериализируемых данных проявлялись аномалии с асимметрией записи [66], хотя MySQL и PostgreSQL являются надежными базами с хорошей репутацией, проверенной многими людьми в течение многих лет. В менее зрелом ПО ситуация, вероятно, будет намного хуже.

Несмотря на значительные усилия по тщательному проектированию, тестированию и проверкам, ошибки все еще появляются. Хотя они встречаются редко и в конце концов их находят и исправляют, остается период, в течение которого такие ошибки могут испортить данные.

Когда дело касается кода приложения, приходится предполагать большое количество ошибок, поскольку большинство приложений не тестируют и не проверяют так, как это делается в случае с базами данных. Многие приложения даже неправильно используют функции, предлагаемые базами для сохранения целостности, такие как внешние ключи или ограничения уникальности [36].

Согласованность в смысле ACID (см. пункт «Согласованность» подраздела «Смысл аббревиатуры ACID» раздела 7.1) основана на следующей идее: база

данных создана в согласованном состоянии, а транзакция преобразует ее из одного согласованного состояния в другое. Таким образом, мы ожидаем, что БД всегда находится в согласованном состоянии. Однако это имеет смысл только при условии, что в транзакции нет ошибок. Если же приложение использует базу некорректно — например, применяя слабый уровень изоляции, — то целостность БД не гарантируется.

Не верьте слепо обещаниям

Поскольку и аппаратное, и программное обеспечение не всегда соответствует желаемому идеалу, повреждение данных неизбежно произойдет, рано или поздно. Поэтому необходим по крайней мере способ выяснить, были ли данные повреждены, чтобы можно было их исправить и попытаться найти источник ошибки. Проверка целостности данных называется *аудитом*.

Как обсуждалось в пункте «Преимущества неизменяемых событий» подраздела «Состояние, потоки и неизменяемость» раздела 11.2, аудит предназначен не только для финансовых приложений. Тем не менее возможность аудита очень важна в финансах именно потому, что все знают о вероятности ошибок и признают необходимость выявления и устранения проблем.

Аналогичным образом зрелые системы стремятся учитывать возможность маловероятных сбоев и управлять этим риском. Например, крупномасштабные системы хранения информации, такие как HDFS и Amazon S3, не полностью доверяют дискам: они запускают фоновые процессы, которые постоянно считывают файлы, сравнивают их с другими репликами и перемещают файлы с одного диска на другой, чтобы уменьшить риск «тихого» повреждения [67].

Если вы хотите убедиться, что ваши данные все еще существуют, необходимо их прочитать и проверить. В большинстве случаев проверка будет успешной, но при другом развитии событий лучше узнать об этом раньше, чем позже. По той же причине важно периодически пытаться восстанавливать данные из резервных копий, иначе резервная копия может оказаться поврежденной, когда будет уже слишком поздно, и данные потеряются. Нельзя слепо верить, что все работает как надо.

Культура проверки

Системы, подобные HDFS и S3, все еще предполагают правильную работу дисков большую часть времени — это разумно, но не равняется предположению, что диски *всегда* работают правильно. Однако немного найдется современных систем с таким постоянным аудитом в стиле «доверяй, но проверяй». Часто гарантии корректности считаются абсолютными и возможность редкого повреждения данных не рассматривается. Надеюсь, что в будущем мы увидим системы с более высоким уровнем

самопроверки или *самоаудита*, которые бы постоянно проверяли свою целостность, не полагаясь на слепое доверие [68].

Я опасаюсь, что культура баз данных ACID привела нас к разработке приложений на основе слепо доверяющих технологий (таких как механизм транзакций), пренебрегая любыми проверками в процессе. Поскольку технология, которой мы доверяли, достаточно хорошо работала большую часть времени, считалось излишним вкладывать средства в механизмы аудита.

Но затем ландшафт баз данных изменился: с появлением NoSQL более слабые гарантии последовательности стали нормой и получили широкое распространение менее проверенные технологии хранения. Тем не менее, поскольку механизмы аудита не были разработаны, мы продолжали создавать приложения на основе слепого доверия, хотя теперь этот подход стал более опасным. Остановимся на минуту и поразмыслим о проектировании с учетом аудитопригодности.

Аудитопригодная архитектура

Если транзакция блокирует несколько объектов в базе данных, то после ее окончания сложно сказать, что именно она делала. Даже при наличии журнала транзакций (см. подраздел «Перехват изменений данных» раздела 11.2) все равно вставки, обновления и удаления, сделанные в разных таблицах, не всегда позволяют составить четкое представление о том, *зачем* были сделаны эти блокировки. Вызов логики приложения, которое их инициировало, является временным и не может быть воспроизведен.

Системы, основанные на событиях, напротив, способны обеспечить лучшую аудитопригодность. В случае применения источников данных пользовательский ввод в систему представляется как одно неизменяемое событие, и все результирующие обновления состояния производятся из него. Генерацию производных состояний можно сделать детерминированной и повторяемой, так что выполнение журнала событий через тот же код деривации приведет к тому же обновлению состояния.

Благодаря явному представлению информационного потока (см. пункт «Философия выходных данных пакетного процесса» подраздела «Выходные данные пакетных потоков» раздела 10.2) *происхождение* данных становится более понятным; это делает более выполнимой проверку целостности. Чтобы убедиться в отсутствии повреждений хранилища событий, можно использовать кэши журнала событий. Для любого производного состояния можно повторно запустить пакетные и потоковые процессоры, которые создали его на основе журнала событий, и проверить, получим ли мы тот же результат, или даже запустить параллельно создание избыточного производного состояния.

Детерминированный и четко определенный информационный поток также облегчает отладку и отслеживание выполнения системы с целью определить, почему она

работает так, а не иначе [4, 69]. При неожиданном развитии событий важно иметь доступ к проведению диагностики, уметь воспроизводить точные обстоятельства, которые привели к этому событию, — своего рода возможность отладки во времени.

И снова о сквозных аргументах

Если нельзя полностью полагаться на то, что каждый отдельный компонент системы полностью защищен от ошибок — все аппаратные средства безотказны и все программы работают без ошибок, — то необходимо хотя бы время от времени проверять целостность наших данных. Не сделав этого, не узнаем о повреждении до тех пор, пока не станет слишком поздно и результат ошибки не распространится по системе. На данном этапе будет намного сложнее и дороже отследить проблему.

Проверку целостности систем данных лучше всего выполнять в сквозном режиме (см. подраздел «Сквозные аргументы в базе данных» текущего раздела): чем больше систем включено в проверку целостности, тем меньше вероятность того, что на каком-то этапе процесса повреждение останется незамеченным. Если можно гарантировать корректное состояние всего конвейера генерации данных, то все диски, сети, сервисы и алгоритмы вдоль этого пути неявно включаются в проверку.

Непрерывные сквозные проверки целостности обеспечивают повышенную уверенность в корректности системы; это, в свою очередь, позволяет двигаться быстрее [70]. Как и в случае с автоматическим тестированием, аудит увеличивает вероятность быстрого обнаружения ошибок и, таким образом, уменьшит риск того, что изменение системы или новой технологии хранения приведет к ее повреждению. Если же вы не боитесь изменений, то можете значительно лучше развивать приложение для удовлетворения меняющихся требований.

Инструменты для аудитопригодных систем данных

В настоящее время не так уж много информационных систем уделяют первостепенное внимание аудитопригодности. В некоторых приложениях реализованы собственные механизмы аудита — например, путем регистрации всех изменений в отдельной таблице. Но гарантировать целостность журнала аудита и базы данных состояний все еще сложно. Журнал транзакций может быть защищен от несанкционированного доступа благодаря периодической вставке подписей его с помощью аппаратного модуля безопасности, но это не гарантирует попадания в журнал правильных транзакций в первую очередь.

Было бы интересно использовать криптографические инструменты для гарантии целостности системы таким образом, чтобы она была устойчивой к широкому спектру аппаратных и программных проблем и даже к потенциально злонамеренным действиям. В ходе исследований в этой области появились такие технологии, как

криптовалюты, блокчейны и распределенные реестры — Bitcoin, Ethereum, Ripple, Stellar и др. [71–73].

Мне не хватает квалификации, чтобы обсуждать достоинства упомянутых технологий как валюты или механизмы согласования контрактов. Однако с точки зрения информационных систем в них есть интересные идеи. По сути, они представляют собой распределенные базы с моделью данных и механизмом транзакций, в которых разные реплики могут размещаться организациями, не доверяющими друг другу. Эти реплики постоянно проверяют целостность друг друга и используют согласованный протокол для подтверждения выполняемых транзакций.

Я несколько скептически отношусь к византийским аспектам отказоустойчивости этих технологий (см. подраздел «Византийские сбои» раздела 8.4) и считаю технику *доказательства выполненной работы* (такую как Bitcoin mining) необычайно расточительной. Пропускная способность транзакции Bitcoin довольно низкая, хотя скорее по политическим и экономическим, чем по техническим причинам. Тем не менее ее аспекты проверки целостности интересны.

Криптографический аудит и проверка целостности часто зависят от *деревьев Меркле* [74], которые являются деревьями хешей и могут быть использованы для эффективного доказательства того, что запись появляется в отдельных наборах данных (и других объектах). Если же оставить в стороне криптовалюты, то есть еще *прозрачность сертификатов* — технология обеспечения безопасности, применяющая деревья Меркле для проверки сертификатов TLS/SSL [75, 76].

Можно предполагать, что алгоритмы проверки целостности и аудита, такие как прозрачность сертификатов и распределенные реестры, будут все более широко использоваться в информационных системах. Потребуется немного поработать с целью сделать их масштабируемыми в той же степени, что и системы без криптографического аудита, и максимально снизить вычислительные затраты. Но я считаю, что это интересная область и за ней будущее.

12.4. Делать что должно

В заключительном разделе данной книги я хотел бы сделать шаг назад. Мы рассмотрели широкий круг различных архитектур для информационных систем, оценили их достоинства и недостатки, исследовали методы построения надежных, масштабируемых и поддерживаемых приложений. Однако мы оставили в стороне фундаментально важную часть обсуждения, и сейчас я хочу восполнить этот пробел.

Каждая система предназначена для определенной цели; каждое действие, которое мы предпринимаем, имеет как предполагаемые, так и непреднамеренные последствия. Цель может быть простой, например заработать деньги, но ее последствия для всего мира могут выйти далеко за рамки первоначальной цели. Мы, разработчики

этих систем, несем ответственность за то, чтобы тщательно продумать последствия и принять сознательное решение о том, в каком мире хотим жить.

Мы говорим о данных как об абстракции, но будем помнить, что многие наборы данных описывают людей: их поведение, интересы, личность. Мы должны относиться к подобным данным с человечностью и уважением. Пользователи тоже люди, и человеческое достоинство имеет первостепенное значение.

Разработка программного обеспечения все чаще предполагает принятие важных этических решений. Существуют рекомендации, которые помогают создателям ПО решать эти проблемы, например Кодекс этики и профессиональной практики при разработке программного обеспечения ACM [77], но они редко обсуждаются и применяются на практике и на их использовании редко настаивают. В результате разработчики и менеджеры иногда слишком бесцеремонно относятся к конфиденциальности, игнорируя потенциальные негативные последствия для своих продуктов [78–80].

Сама по себе технология не хороша или плоха — все зависит от способа ее использования и влияния на людей. Это справедливо в равной степени и для программного обеспечения, такого как поисковые системы, и для пулемета. Я считаю, что разработчикам ПО недостаточно уделять внимание исключительно технологии, игнорируя ее последствия: этическая ответственность также лежит на нас. Рассуждать об этике сложно, но игнорировать ее слишком опасно.

Предсказательная аналитика

Возьмем, например, предсказательную аналитику. Она является важной частью шума вокруг «Больших данных». Использование анализа данных для составления прогноза о погоде или распространении болезни — это одно [81], и совсем другое — предсказать, нарушит ли снова осужденный закон, разорится ли заявитель на получение кредита или предъявит ли владелец страховки дорогостоящие претензии. Все перечисленное оказывает непосредственное влияние на жизнь конкретных людей.

Естественно, платежные сети стремятся предотвратить мошеннические транзакции, банки хотят избежать непогашенных кредитов, авиакомпании — угонов самолетов, а организации — найма неэффективных или ненадежных людей. С их точки зрения, потери от упущенных бизнес-возможностей низкие, но от невозвращенной ссуды или проблемного сотрудника они намного выше, поэтому естественно быть осторожными. Если есть сомнения, то лучше сказать «нет».

Однако по мере того, как алгоритмическое принятие решений становится все более распространенным, человек, которого какой-то алгоритм (оправданно или по ошибке) посчитал ненадежным, может столкнуться с большим количеством таких «нет». Систематический отказ от работы, авиапутешествий, предоставления страховки, аренды недвижимости, финансовых услуг и других ключевых аспектов

деятельности является таким большим ограничением свободы личности, что его называют «алгоритмической тюрьмой» [82]. В странах, где уважают права человека, система уголовного правосудия предполагает невиновность до тех пор, пока вина не будет доказана; однако автоматизированные системы могут систематически и произвольно исключать человека из участия в общественной жизни без каких-либо доказательств вины и с малой вероятностью обжалования.

Пристрастное отношение и дискриминация

Решения, обоснованные логикой алгоритма, не обязательно лучше или хуже тех, что приняты человеком. У каждого человека могут быть предвзятые суждения, даже если он активно пытается им противодействовать, и дискриминационная практика может стать требованием культуры. Есть надежда, что основополагающие решения, основанные на информации, а не на субъективных и инстинктивных оценках людей, будут более справедливыми и дадут больше шансов людям, которых в традиционной системе часто недооценивают [83].

Разрабатывая интеллектуальные аналитические системы, мы не просто автоматизируем решение человека, используя программное обеспечение для определения правил, по которым он будет говорить «да» или «нет»; мы позволяем самим правилам делать выводы на основе имеющихся данных. Однако модели, по которым обучались эти системы, непрозрачны: даже если в данных есть какая-то закономерность, мы, возможно, не знаем ее причины. При наличии во входных данных алгоритма систематической предвзятости система, скорее всего, изучит и усилит ее на выходе [84].

Во многих странах антидискриминационные законы запрещают обращение с людьми в зависимости от определенных «защищенных» признаков, таких как этническая принадлежность, возраст, пол, сексуальная ориентация, инвалидность или убеждения. Другие персональные данные могут быть проанализированы, но что произойдет, если они связаны с защищенными признаками? Например, в районах компактного проживания людей одной расы расовую принадлежность человека можно с большой вероятностью определить по почтовому индексу или даже IP-адресу. Поэтому было бы смешно полагать, что алгоритм способен каким-то образом использовать предвзятые данные в качестве входной информации и сделать на их основе справедливый и беспристрастный вывод [85]. Тем не менее такая убежденность, по-видимому, часто подразумевается сторонниками принятия решений, основанных на результатах машинного анализа. Это мнение получило едкую оценку: «машинное обучение подобно отмыванию денег за предвзятости» [86].

Прогнозирующие аналитические системы просто экстраполируют события по данным прошлого; в случае дискриминационного прошлого они кодифицируют эту дискриминацию. Если мы хотим, чтобы будущее было лучше прошлого, то необходимо моральное воображение, и это может сделать только человек [87]. Данные и модели должны быть нашими инструментами, но не хозяевами.

Ответственность и подотчетность

Автоматизированное принятие решений поднимает вопрос об ответственности и подотчетности [87]. Если человек совершает ошибку, то его можно привлечь к ответственности и пострадавший от этого решения способен подать апелляцию. Алгоритмы тоже допускают ошибки, но кто несет ответственность в случае их ошибки [88]? Кто ответит за дорожно-транспортное происшествие, виновником которого стал самоуправляемый автомобиль? К кому апеллировать, если автоматизированный алгоритм, принимающий решение о выдаче кредитов, систематически дискриминирует людей определенной расы или религии? Допустим, в основе судебного решения лежит вывод системы машинного обучения, сможете ли вы тогда объяснить судье, как алгоритм сделал этот вывод?

Одним из старейших примеров сбора данных для принятия решений о человеке являются агентства кредитной классификации. Плохая кредитная история усложняет жизнь, но по крайней мере кредитный рейтинг обычно основывается на соответствующих фактах из действительной финансовой истории человека и любые ошибки в записи могут быть исправлены (хотя в обычной практике агентств сделать это нелегко). Однако алгоритмы вычисления рейтинга, основанные на машинном обучении, обычно используют гораздо более широкий диапазон входных данных и гораздо менее прозрачны. Это затрудняет понимание того, как принимается конкретное решение, не было ли оно несправедливым или дискриминационным [89].

Кредитный рейтинг отвечает на вопрос «Как вы поступали в прошлом?», тогда как прогнозирующая аналитика обычно работает по принципу «Кто похож на вас и как люди, подобные вам, вели себя в прошлом?». Проведение параллелей с другими людьми предполагает стереотипное поведение, например, на основе того, где человек живет (с выводом о расе и социально-экономическом статусе). Как насчет людей, выбивающихся из статистики? Более того, если решение было принято неверно на основе неверных данных, то обратиться в суд практически невозможно [87].

Многие данные являются статистическими по своей природе, а это значит, что даже при в целом верном распределении вероятностей всегда найдутся отдельные ошибочные случаи. Например, средняя продолжительность жизни в вашей стране составляет 80 лет, но это не означает, что вы умрете в свой 80-й день рождения. Из среднего значения и распределения вероятности нельзя сделать существенные выводы о возрасте конкретного человека. Аналогичным образом вывод системы прогнозирования носит вероятностный характер и в отдельных случаях может быть ошибочным.

Слепо верить в превосходство машинных данных для принятия решений не только бредово, но и, безусловно, опасно. Поскольку принятие решений, основанных на машинных данных, становится все более распространенным, необходимо выяснить, как сделать алгоритмы подотчетными и прозрачными,

как избежать усиления существующих предубеждений и их исправить, ведь они неизбежно ошибаются.

Нам также необходимо узнать, как предотвратить применение данных с целью причинить вред людям и вместо этого реализовать их положительный потенциал. Например, аналитика способна выявлять финансовые и социальные характеристики жизни людей. С одной стороны, такую возможность стоит задействовать для поддержки, чтобы помочь тем людям, которые в этом больше всего нуждаются. С другой стороны, данным аспектом иногда пользуется хищнический бизнес, стремящийся выявить уязвимых людей и продать им рискованные продукты, например дорогостоящие кредиты и бесполезные дипломы колледжа [87, 90].

Замкнутая обратная связь

Даже если прогнозирующие приложения, такие как системы рекомендаций, не имеют столь непосредственного далеко идущего влияния на жизнь людей, все равно приходится сталкиваться с серьезными проблемами. Когда сервис способен хорошо предсказать, какое содержимое пользователь захочет видеть, он может показать людям только те мнения, с которыми они уже согласны. Это приводит к появлению замкнутых сообществ, где плодятся стереотипы, дезинформация и поляризация мнений. Мы уже наблюдаем влияние таких сообществ в социальных сетях на избирательные кампании [91].

Когда прогнозирующая аналитика влияет на жизнь людей, особенно пагубные проблемы возникают из-за самоусиливающихся замкнутых циклов обратной связи. Например, рассмотрим случай, когда работодатели используют кредитный рейтинг для оценки потенциальных наемных работников. Вы можете быть хорошим работником с хорошим кредитным рейтингом, но вдруг окажетесь в затруднительном финансовом положении из-за несчастья, случившегося не по вашей вине. Вы пропустите платежи по счетам, и ваш кредитный рейтинг пострадает, а потом из-за него вы вряд ли найдете работу. Безработица подталкивает к бедности, что еще больше ухудшает кредитный рейтинг, а потом становится еще труднее найти работу [87]. Этот порочный круг вызван губительным решением, скрытым за маскировкой математически строгих выводов на основе данных.

Мы не всегда можем предугадать, когда образуются такие замкнутые циклы. Однако многие последствия подвластны предсказанию, если рассмотреть всю систему (не только ее компьютеризированные части, но и людей, взаимодействующих с ними), — подход, известный как *системное мышление* [92]. Можно попытаться понять, как система анализа данных реагирует на различные типы поведения, структуры и характеристики. Подкрепляет ли система и усиливает ли существующие различия между людьми (например, делает богатых еще богаче, а бедных — беднее) или пытается бороться с несправедливостью? И даже с лучшими намерениями следует остерегаться непредусмотренных последствий.

Конфиденциальность и отслеживание

Помимо проблем предсказательной аналитики, таких как применение данных для принятия автоматических решений о людях, существуют этические проблемы, связанные с самим сбором информации. Какова взаимосвязь между организациями, собирающими данные, и людьми, чьи данные собираются?

Если система хранит только те данные, которые пользователь ввел сам, поскольку хочет, чтобы она их хранила и обрабатывала определенным образом, то система выполняет эту задачу для него: он является ее клиентом. Но когда активность пользователя отслеживается и регистрируется как побочный эффект других его действий, отношения менее ясны. Сервис теперь делает не только то, что пожелал пользователь, но исходит из собственных интересов, которые могут противоречить интересам человека.

Отслеживание поведенческих данных становится все более важным для пользовательских функций многих онлайн-сервисов. Например, отслеживание переходов по результатам поиска помогает улучшить ранжирование результатов поиска; рекомендации типа «людям, которым нравится товар X, также понравился товар Y» помогают пользователям находить интересные и полезные вещи. Кроме того, A/B-тестирование и анализ путей пользователей помогают определить, как улучшить UI. Эти функции требуют некоторого отслеживания поведения пользователя, и последний получает от них выгоду.

Однако, в зависимости от бизнес-модели компании, отслеживание часто не останавливается на этом. Если сервис финансируется за счет рекламы, то его фактическими клиентами являются рекламодатели, а интересы пользователей занимают второе место. Данные слежения становятся все более подробными, анализ — все более совершенным, а информация сохраняется в течение длительного времени, позволяя создавать подробные профили каждого человека в маркетинговых целях.

Теперь отношения между компанией и пользователем, чьи данные собираются, начинают выглядеть совсем по-другому. Пользователю предоставляется бесплатный сервис и уговаривают взаимодействовать с ним как можно больше. Отслеживание действий пользователя служит в первую очередь не интересам этого человека, а потребностям рекламодателей, которые финансируют данный сервис. Я полагаю, что эти отношения могут быть правильно описаны словом, имеющим более зловещие коннотации, — *слежка*.

Слежка

В качестве мысленного эксперимента попробуйте заменить слово «данные» на слово «слежка» и понаблюдать за тем, как меняется звучание обычных фраз [93]. Например: «В нашей организации, управляемой с помощью слежки, мы собираем

результаты слежки в реальном времени и храним их в нашем хранилище. Наши ученые — специалисты по слежке используют расширенную аналитику и обработку результатов слежки, чтобы получить новые идеи».

Это очень спорный мысленный эксперимент, как раз для книги «Проектирование приложений для слежки». Но, я думаю, сильные слова необходимы, чтобы подчеркнуть данный момент. Стремясь создать программное обеспечение, которое завоюет мир [94], мы создали самую большую инфраструктуру массовой слежки, какую когда-либо видел мир. Стремясь к Интернету вещей, мы быстро приближаемся к миру, в котором в каждой комнате найдется по крайней мере один подключенный к Интернету микрофон, встроенный в смартфон, интеллектуальный телевизор, устройство с голосовым управлением, детский монитор и даже детские игрушки, в которых используется распознавание речи на основе облачных вычислений. Многие из этих устройств имеют ужасную степень безопасности [95].

Даже самые тоталитарные и репрессивные режимы могли только мечтать о том, чтобы установить микрофон в каждое помещение и принудить каждого человека постоянно носить с собой устройство, способное отслеживать его местоположение и движения. Однако мы, по-видимому, добровольно, даже с энтузиазмом, бросаемся в этот мир тотальной слежки. Разница лишь в том, что данные собираются не государственными учреждениями, а корпорациями [96].

Не любой сбор данных обязательно квалифицируется как слежка, но рассмотрение его под таким углом поможет лучше понять наши отношения со сборщиком данных. Почему мы, по-видимому, счастливы принять слежку со стороны корпораций? Вероятно, вы считаете, что скрывать нечего, другими словами, полностью согласны с существующими силовыми структурами, не относитесь к маргинальному меньшинству и вам не нужно бояться преследования [97]. Не всем так повезло. Или, возможно, это потому, что цель кажется благородной, — это не откровенное принуждение и конформизм, а всего лишь улучшенные рекомендации и более персонализированный маркетинг. Однако если учесть рассуждения о предсказательной аналитике, сделанные в предыдущем разделе, то данное различие становится менее четким.

Уже встречаются страховые взносы за автомобили, связанные с устройствами слежения, встроенными в эти автомобили, и медицинские страховки, требующие, чтобы человек носил на себе устройство отслеживания физической формы. Когда слежка используется для определения вещей, влияющих на важные аспекты жизни, такие как страховка или занятость, она начинает казаться менее благородной. Кроме того, анализ данных способен стать удивительно навязчивым: например, датчик движения в «умных часах» или фитнес-трекере можно использовать для ввода (например, паролей) с довольно хорошей точностью [98]. А алгоритмы для анализа только улучшаются.

Согласие и свобода выбора

Но ведь пользователи добровольно решили применять сервис, который отслеживает их деятельность, не так ли? Они согласились с условиями обслуживания и политикой конфиденциальности, соответственно, согласились и на сбор данных. Можно даже утверждать, что пользователи получают ценный сервис в обмен на их данные и отслеживание необходимо для работы сервиса. Несомненно, социальные сети, поисковые системы и другие бесплатные онлайн-сервисы ценны для пользователей, но с этим аргументом не все гладко.

Пользователи не знают, какие именно сведения о них попадают в БД, как они сохраняются и обрабатываются, а большинство политик конфиденциальности больше скрывают, чем сообщают. Не понимая, что происходит с их данными, пользователи не могут дать сколько-нибудь осознанное согласие. Часто из сведений, поступающих от одного человека, извлекается информация о других людях, которые не являются пользователями сервиса и не давали согласия на сбор данных о них. Производные наборы данных, обсуждавшиеся в этой части книги, в которых информация из всей пользовательской базы, возможно, была объединена с данными поведенческого наблюдения и внешними источниками данных, — именно то, о чем пользователи не могут иметь представления.

Более того, извлечение данных из действий пользователя — это односторонний процесс, а не справедливый, равноценный обмен. Нет диалога, у пользователей нет возможности согласовывать, сколько данных они предоставляют и какой сервис получают взамен: связь между сервисом и пользователем очень асимметричная и односторонняя. Условия ставит сервис, а не пользователь [99].

Для пользователя, который не соглашается на отслеживание, единственной реальной альтернативой является просто не задействовать сервис. Но и этот выбор не является бесплатным: если сервис настолько популярен, что «рассматривается большинством людей как существенная для основного социального участия» [99], то едва ли разумно ожидать, что люди откажутся от него, — его применение *де-факто* является обязательным. Например, в большинстве западных социальных сообществ стало нормой иметь смартфон, общаться с помощью Facebook и использовать Google для поиска информации. При наличии у сервиса сетевых эффектов людям, *не желающим* его применять, приходится нести социальные потери.

Отказ от сервиса из-за того, что он следит за пользователями, возможен лишь для немногих достаточно привилегированных людей, обладающих временем и знаниями, чтобы разобраться в политике конфиденциальности, а также тех, кто может себе позволить упустить участие в общественной жизни или профессиональные возможности, которые возникают благодаря применению данного сервиса. У людей, находящихся в менее привилегированном положении, в сущности, нет свободы выбора: слежка становится неизбежной.

Использование данных и конфиденциальность

Иногда говорят, что «конфиденциальность умерла», на том основании, что некоторые пользователи готовы публиковать в социальных сетях всевозможные сведения о своей жизни, иногда повседневные, а иногда и глубоко личные. Однако такое утверждение неверно. Оно основано на неверном понимании слова *«конфиденциальность»*.

Конфиденциальность не означает полную секретность; здесь речь идет о свободе выбирать, что и кому сообщать, публиковать, а что хранить в секрете. Право на неприкосновенность частной жизни является правом выбора: оно позволяет человеку самому решать, где проходит граница между секретностью и прозрачностью в каждом конкретном случае [99]. Это важный аспект свободы и автономии личности.

Когда данные получают от людей через инфраструктуру слежения, это еще не значит, что права конфиденциальности обязательно нарушаются, скорее ответственность за них возлагается на сборщика данных. Компании, которые приобретают данные, по сути, заявляют: «Доверьтесь нам, мы будем хорошо поступать с вашими данными». Это значит, что право решать, какую информацию открыть, а какую хранить в тайне, переходит от индивидуума к компании.

Компании, в свою очередь, предпочитают хранить большую часть результатов наблюдения в секрете, поскольку раскрыть их означало бы вызвать отвращение к себе и повредить бизнес-модель (которая основана на то, чтобы знать о людях больше, чем другие компании). Знание личной информации о пользователях проявляется косвенно, например, в виде инструментов целевой рекламы для определенных групп людей (например, тех, кто страдает от конкретной болезни).

Даже если отдельные пользователи не могут быть лично выделены из группы людей, для которых было предназначено конкретное объявление, они все равно потеряли право решать, раскрывать ли некую интимную информацию, например факт страдания от определенной болезни. Теперь уже пользователь не решает, что и кому раскрывать, исходя из своих личных предпочтений, — компания применяет право собственности с целью максимизации прибыли.

У многих компаний есть цель не *казаться* столь ужасными. Они избегают вопросов о том, насколько сильно их сбор данных вмешивается в личную жизнь пользователей, и вместо этого обращают больше внимания на управление впечатлениями отдельного человека. Но даже ими часто управляют плохо: например, что-то может быть формально правильным, но если вызывает болезненные воспоминания, то пользователь может не захотеть, чтобы ему об этом напоминали [100]. Любые данные могут оказаться некорректными, нежелательными или неуместными в том или ином смысле, и необходимы механизмы для обработки этих ошибок. Конечно, понятие нежелательности или неуместности зависит от

мнения человека; алгоритмы не обращают внимания на такие вещи, если только явно не запрограммировать их на удовлетворение человеческих потребностей. Нам, как разработчикам таких систем, следует смиренно принимать и планировать подобные неудачи.

Параметры конфиденциальности, которые позволяют пользователю онлайн-сервиса контролировать, какие данные могут видеть другие пользователи, а какие — нет, являются отправной точкой для передачи некоего контроля пользователям. Однако, независимо от настройки, сам сервис по-прежнему имеет неограниченный доступ к данным и может свободно использовать их любым способом, разрешенным политикой конфиденциальности. Даже пообещав не продавать данные третьим сторонам, он обычно предоставляет себе неограниченные права на обработку и анализ данных внутри страны, часто при этом заходя гораздо дальше, чем способен представить пользователь.

Такой масштабный перенос прав на частную жизнь от частных лиц к корпорациям не имеет прецедентов в истории [99]. Наблюдение всегда существовало, но оно было дорогостоящим и производимым вручную, а не масштабируемым и автоматизированным. Доверительные отношения существовали всегда — например, между пациентом и врачом или между подсудимым и адвокатом, но в подобных случаях использование данных строго регулировалось этическими, юридическими и нормативными ограничениями. Интернет-сервисы значительно упростили сбор огромного количества конфиденциальной информации без осознанного согласия и ее массовое применение без понимания пользователями того, что происходит с их частными данными.

Данные как капитал и власть

Поскольку поведенческая информация является побочным продуктом взаимодействия пользователей с сервисом, ее иногда называют «выхлопом данных» — намекая на то, что эти данные являются бесполезными отходами. Таким образом, поведенческая и прогнозирующая аналитика может рассматриваться как форма повторной переработки, извлекающая ценность из данных, которые иначе были бы отброшены.

Правильнее было бы рассматривать все наоборот: с экономической точки зрения если сервис оплачивает целевая реклама, то поведенческие данные о людях являются основным его капиталом. А приложение, с которым взаимодействует пользователь, — всего лишь средство заставить людей вводить в инфраструктуру слежения как можно больше личной информации [99]. Удивительные творческие способности и социальные отношения, часто находящие выражение в онлайн-сервисах, цинично эксплуатируются машиной для извлечения данных.

Утверждение о том, что персональные данные являются ценным активом, подтверждается наличием брокеров данных, теневой индустрии, работающей в тайне,

покупкой, сбором, анализом, выводом и перепродажей конфиденциальных данных о людях, в основном в маркетинговых целях [90]. Новые онлайн-бизнесы оцениваются по их количеству пользователей, по «глазным яблокам» — другими словами, по возможностям слежения.

Поскольку данные ценны, многие хотят их получить. Конечно, этого хотят компании — вот почему они в первую очередь собирают данные. Но правительства тоже хотят получить информацию: с помощью секретных сделок, принуждения, в том числе юридического, или просто кражи [101]. Когда компания становится банкротом, собранные ею личные данные являются одним из активов, которые выставляются на продажу. Кроме того, сведения трудно сохранить в секрете, поэтому их утечки, к сожалению, случаются часто [102].

Подобные наблюдения заставили критиков признать, что данные — это не просто актив, а «токсичный актив» [101] или по крайней мере «опасный материал» [103]. Даже если предположить, что мы способны предотвращать злоупотребление данными, всякий раз в момент их сбора необходимо найти баланс между выгодой и риском попадания информации в чужие руки: есть риск взлома компьютерных систем преступниками или враждебными службами внешней разведки, утечку данных могут организовать инсайдеры, компания может попасть в руки недобросовестного руководства, не разделяющего наши ценности, или страна может быть захвачена режимом, который не остановится перед тем, чтобы заставить нас передать данные.

При сборе данных нужно учитывать не только сегодняшнюю политическую ситуацию, но и возможные будущие правительства. Нет гарантии, что все правительства, избранные в будущем, будут уважать права человека и гражданские свободы, поэтому «использовать технологии, которые однажды могут помочь полицейскому государству, — плохая гражданская гигиена» [104].

«Знание — сила», как гласит старая поговорка. Более того, «тщательно изучать других, избегая при этом тщательного изучения себя, — одна из важнейших форм власти» [105]. Вот почему тоталитарные правительства так любят слежку: она дает им возможность контролировать население. Хотя сегодняшние технологические компании не стремятся к явной политической власти, накопленные ими данные и знания дают им большую власть над нашей жизнью, львиная часть которой находится вне пределов общественного контроля [106].

Вспомним индустриальную революцию

Данные являются ключевым свойством информационной эпохи. Интернет, хранение, обработка, программная автоматизация данных оказывают большое влияние на мировую экономику и человеческое общество. Поскольку наша повседневная жизнь и социальная организация изменились за последнее десятилетие и, вероятно,

в ближайшие десятилетия по-прежнему будут кардинально меняться, мне хочется провести аналогию с промышленной революцией [87, 96].

Промышленная революция произошла благодаря значительным достижениям в области технологий и сельского хозяйства. В долгосрочной перспективе она привела к устойчивому экономическому росту и значительно повысила уровень жизни. Тем не менее она также сопровождалась серьезными проблемами: загрязнение воздуха (из-за дыма и химических процессов) и воды (из-за промышленных и человеческих отходов) было ужасным. Владельцы фабрик пребывали в роскоши, в то время как городские рабочие часто жили в очень плохих домах и работали многие часы в тяжелых условиях. Детский труд был обычным явлением, включая опасную и плохо оплачиваемую работу на шахтах.

Прошло много времени, прежде чем были установлены гарантии, такие как нормы охраны окружающей среды, протоколы безопасности для рабочих мест, запрет детского труда и санитарный надзор за продуктами питания. Конечно, когда фабрики больше не смогли сбрасывать отходы в реки, продавать испорченные продукты или эксплуатировать рабочих, стоимость ведения бизнеса выросла. Но общество в целом получило огромную выгоду, и немногие из нас хотели бы вернуться к тем временам, когда этих правил не существовало [87].

Подобно тому как промышленная революция имела оборотную сторону, с которой нужно было справляться, переход к информационной эпохе также имеет серьезные проблемы, требующие противостояния и методов решения. Я считаю, что сбор и использование данных — одна из таких проблем. Брюс Шнайер (Bruce Schneier) по этому поводу пишет следующее [96].

> «Данные — это аналог проблемы загрязнения окружающей среды в эпоху информации, а защита конфиденциальности — экологическая проблема. Почти все компьютеры производят информацию. Она остается вокруг, как нагноение. Как с этим справиться — как его остановить и как им распоряжаться — центральный вопрос для здоровья информационной экономики. Подобно тому как мы сегодня оглядываемся назад, в первые десятилетия индустриальной эпохи, и удивляемся, что наши предки могли игнорировать загрязнение окружающей среды, спеша построить индустриальный мир, наши внуки будут оглядываться на нас, живущих в первые десятилетия информационной эпохи, и судить нас по нашим способам решения проблем, касающихся сбора и неправильного использования данных.
>
> Мы должны постараться сделать так, чтобы они гордились нами».

Законодательство и саморегулирование

Законы о защите данных способны помочь сохранить права индивидуума. Например, в Европейской директиве по защите данных 1995 года указывается, что персональные данные должны «собираться для определенных, явных и законных целей и в дальнейшем не подвергаться обработке, несовместимой с этими целями».

Кроме того, данные должны быть «адекватными, релевантными и не избыточными в отношении тех целей, для которых собираются» [107].

Однако сомнительна эффективность этого закона в современном интернет-контексте [108]. Представленные правила напрямую противоречат философии Больших данных, которая заключается в том, чтобы собирать как можно больше данных, объединять их с другими, экспериментировать и исследовать с целью генерирования новых идей. Исследование означает применение данных в непредвиденных целях, что противоречит формулировке «конкретные и явные цели», на которые пользователь дал согласие (если мы вообще можем говорить о согласии [109]). В настоящее время разрабатываются обновленные правила [89].

Компании, собирающие множество данных о людях, выступают против регулирования, которое тормозит и препятствует инновациям. В какой-то степени такая оппозиция оправданна. Например, при обмене медицинскими данными существует явный риск для конфиденциальности, но есть и потенциальные возможности: сколько смертей можно было бы предотвратить, если бы анализ данных помог улучшить диагностику или найти лучшее лечение [110]? Чрезмерное регулирование способно помешать таким прорывам. Трудно найти баланс между потенциальными возможностями и рисками [105].

По сути, я считаю, нам нужно изменить технологическую культуру в отношении персональных данных. Мы должны перестать рассматривать пользователей как оптимизируемые показатели и помнить, что они люди, которые заслуживают уважения, признания их достоинства и равноправного взаимодействия. Нам следует самим регулировать свои методы сбора и обработки данных, чтобы заслужить и сохранить доверие людей, зависящих от нашего программного обеспечения [111]. И мы должны взять на себя ответственность за информирование пользователей о том, как применяются их данные, а не держать их в неведении.

Нужно позволить каждому человеку сохранять конфиденциальность, то есть контролировать свои данные, а не перехватывать контроль над ними с помощью слежения. Личное право каждого контролировать свои данные подобно природе в заповеднике: если мы не будем явно защищать и заботиться о нем, то оно будет уничтожено. Это была бы общая трагедия, и всем стало бы только хуже. Тотальное слежение не является неизбежным — мы все еще можем его предотвратить.

Вопрос, как именно этого достичь, остается открытым. Прежде всего, не следует сохранять данные навсегда. Вместо этого их нужно удалять сразу, как только надобность в них отпала [111, 112]. Очистка данных противоречит идее неизменяемости (см. пункт «Ограничения неизменяемости» подраздела «Состояние, потоки и неизменяемость» раздела 11.2), но этот вопрос можно решить. Мне представляется многообещающим подход, который заключается в обеспечении контроля доступа не только с помощью политик, но и через криптографические протоколы [113, 114]. В целом потребуются изменения культуры и мировоззрения.

12.5. Резюме

В настоящей главе мы обсудили новые подходы к проектированию информационных систем, и я включил в нее свои личные мнения и размышления о будущем. Мы начали с того, что нет ни одного инструмента, который бы эффективно справлялся со всеми возможными вариантами использования, поэтому каждый раз для достижения своих целей необходимо комбинировать разные программные продукты. Мы обсудили, как решить эту проблему *интеграции данных* с помощью пакетной обработки и потоков событий, чтобы обеспечить передачу данных между разными системами.

В предложенном методе одни системы были отнесены к системам записи, в то время как другие получают от них данные путем преобразований. Таким образом можно поддерживать индексы, материализованные представления, модели машинного обучения, статистические сводки и многое другое. Сделав эти преобразования и трансформации асинхронными и слабосвязанными, мы не даем локальным проблемам распространиться на другие части системы, тем самым повышая надежность и отказоустойчивость системы в целом.

Представление потоков данных как преобразований из одного набора данных в другой также помогает развивать приложения: чтобы изменить один из этапов обработки (например, структуру индекса или кэша), достаточно просто запустить новый код преобразования для всего набора входных данных и получить результат. Аналогичным образом, если что-то пойдет не так, можно исправить код и снова обработать данные с целью их восстановить.

Эти процессы очень похожи на те, что уже реализованы внутри баз данных, и, следовательно, мы переходим к идее приложений информационных потоков как к *разделению* компонентов БД и построению приложения путем составления этих слабосвязанных компонентов.

Производное состояние может быть обновлено с помощью отслеживания изменений в базовых данных. Более того, само производное состояние способно дополнительно отслеживаться потребителями, расположенными ниже по потоку. Можно даже довести этот поток данных до конечного пользователя, для которого отображаются данные, и таким образом создавать динамически обновляемые UI, чтобы отражать изменения данных и продолжать работать в автономном режиме.

Затем мы обсудили, как сделать так, чтобы вся эта обработка оставалась корректной в случае сбоев. Мы увидели следующее: надежные гарантии целостности могут быть реализованы с возможностью масштабирования при условии асинхронной обработки событий с помощью сквозных идентификаторов операций, которые позволяют сделать операции идемпотентными, и путем асинхронной проверки ограничений. Клиенты могут либо дождаться проверки, либо продолжить не дожидаясь и потом принести извинения за нарушение ограничений. Этот метод

гораздо более масштабируемый и надежный, чем традиционный подход с использованием распределенных транзакций, и именно так на практике работают многие бизнес-процессы.

Структурируя приложения вокруг потока данных и асинхронно проверяя ограничения, мы можем избежать избыточной координации и создавать системы, сохраняющие целостность, но при этом хорошо работающие даже в географически распределенных структурах и при наличии сбоев. Затем мы немного поговорили об использовании аудита для проверки целостности данных и обнаружения повреждений данных.

Наконец, мы немного отступили от темы и рассмотрели некоторые этические аспекты построения высоконагруженных данными приложений. Мы увидели, что, хотя данные можно применять для благих целей, они все же способны причинить значительный ущерб: например, стать обоснованием для решений, серьезно влияющих на жизнь людей. Такие решения трудно обжаловать, а это, в свою очередь, ведет к дискриминации, слежке и раскрытию конфиденциальной информации. Мы также исследовали риск утечки данных. Теперь мы можем сделать вывод, что использование данных с благими намерениями может иметь непредвиденные последствия.

Поскольку программное обеспечение и информация столь сильно влияют на окружающий мир, мы, разработчики, должны помнить, что несем ответственность за работу, направленную в мир, в котором хотим жить, — мир, относящийся к людям гуманно и уважительно. Я надеюсь, что вместе мы достигнем этой цели.

12.6. Библиография

1. *Belaid R.* Postgres Full-Text Search is Good Enough! July 13, 2015 [Электронный ресурс]. — Режим доступа: http://rachbelaid.com/postgres-full-text-search-is-good-enough/.
2. *Ajoux P., Bronson N., Kumar S., et al.* Challenges to Adopting Stronger Consistency at Scale // 15th USENIX Workshop on Hot Topics in Operating Systems (HotOS), May 2015 [Электронный ресурс]. — Режим доступа: https://www.usenix.org/system/files/conference/hotos15/hotos15-paper-ajoux.pdf.
3. *Helland P., Campbell D.* Building on Quicksand // 4th Biennial Conference on Innovative Data Systems Research (CIDR), January 2009 [Электронный ресурс]. — Режим доступа: https://database.cs.wisc.edu/cidr/cidr2009/Paper_133.pdf.
4. *Kerr J.* Provenance and Causality in Distributed Systems. September 25, 2016 [Электронный ресурс]. — Режим доступа: http://blog.jessitron.com/2016/09/provenance-and-causality-in-distributed.html.
5. *Tzoumas K.* Batch Is a Special Case of Streaming. September 15, 2015 [Электронный ресурс]. — Режим доступа: https://data-artisans.com/blog/batch-is-a-special-case-of-streaming.

6. *Kim S., Blafford R.* Stream Windowing Performance Analysis: Concord and Spark Streaming. July 6, 2016 [Электронный ресурс]. — Режим доступа: http://concord.io/posts/windowing_performance_analysis_w_spark_streaming.
7. *Kreps J.* The Log: What Every Software Engineer Should Know About RealTime Data's Unifying Abstraction. December 16, 2013 [Электронный ресурс]. — Режим доступа: https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying.
8. *Helland P.* Life Beyond Distributed Transactions: An Apostate's Opinion // 3rd Biennial Conference on Innovative Data Systems Research (CIDR), January 2007 [Электронный ресурс]. — Режим доступа: http://www-db.cs.wisc.edu/cidr/cidr2007/papers/cidr07p15.pdf.
9. Great Western Railway (1835–1948) // Network Rail Virtual Archive. [Электронный ресурс]. — Режим доступа: https://www.networkrail.co.uk/running-the-railway/our-routes/western/great-western-mainline/.
10. *Xu J.* Online Migrations at Scale. February 2, 2017 [Электронный ресурс]. — Режим доступа: https://stripe.com/blog/online-migrations.
11. *Dishman M. B., Fowler M.* Agile Architecture // O'Reilly Software Architecture Conference, March 2015 [Электронный ресурс]. — Режим доступа: https://conferences.oreilly.com/software-architecture/sa2015/public/schedule/detail/40388.
12. *Marz N., Warren J.* Big Data: Principles and Best Practices of Scalable Real-Time Data Systems. — Manning, 2015. https://www.manning.com/books/big-data.
13. *Boykin O., Ritchie S., O'Connell I., Lin J.* Summingbird: A Framework for Integrating Batch and Online MapReduce Computations // 40th International Conference on Very Large Data Bases (VLDB), September 2014 [Электронный ресурс]. — Режим доступа: http://www.vldb.org/pvldb/vol7/p1441-boykin.pdf.
14. *Kreps J.* Questioning the Lambda Architecture. July 2, 2014 [Электронный ресурс]. — Режим доступа: https://www.oreilly.com/ideas/questioning-the-lambda-architecture.
15. *Fernandez R. C., Pietzuch P., Kreps J., et al.* Liquid: Unifying Nearline and Offline Big Data Integration // 7th Biennial Conference on Innovative Data Systems Research (CIDR), January 2015 [Электронный ресурс]. — Режим доступа: http://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper25u.pdf.
16. *Ritchie D. M., Thompson K.* The UNIX Time-Sharing System // Communications of the ACM, volume 17, number 7, pages 365–375, July 1974 [Электронный ресурс]. — Режим доступа: http://www.cs.virginia.edu/~zaher/classes/CS656/p365-ritchie.pdf.
17. *Brewer E. A., Hellerstein J. M.* CS262a: Advanced Topics in Computer Systems // lecture notes, University of California, Berkeley, August 2011 [Электронный ресурс]. — Режим доступа: http://people.eecs.berkeley.edu/~brewer/cs262/systemr.html.
18. *Stonebraker M.* The Case for Polystores. July 13, 2015 [Электронный ресурс]. — Режим доступа: http://wp.sigmod.org/?p=1629.

19. *Duggan J., Elmore A. J., Stonebraker M., et al.* The BigDAWG Polystore System // ACM SIGMOD Record, volume 44, number 2, pages 11–16, June 2015 [Электронный ресурс]. — Режим доступа: http://dspace.mit.edu/openaccess-disseminate/1721.1/100936.
20. *Dybka P.* Foreign Data Wrappers for PostgreSQL. March 24, 2015 [Электронный ресурс]. — Режим доступа: http://www.vertabelo.com/blog/technical-articles/foreign-data-wrappers-for-postgresql.
21. *Lomet D. B., Fekete A., Weikum G., Zwilling M.* Unbundling Transaction Services in the Cloud // 4th Biennial Conference on Innovative Data Systems Research (CIDR), January 2009 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/unbundling-transaction-services-in-the-cloud/.
22. *Kleppmann M., Kreps J.* Kafka, Samza and the Unix Philosophy of Distributed Data // IEEE Data Engineering Bulletin, volume 38, number 4, pages 4–14, December 2015 [Электронный ресурс]. — Режим доступа: http://martin.kleppmann.com/papers/kafka-debull15.pdf.
23. *Hugg J.* Winning Now and in the Future: Where VoltDB Shines. March 23, 2016 [Электронный ресурс]. — Режим доступа: https://www.voltdb.com/blog/2016/03/23/winning-now-future-voltdb-shines/.
24. *McSherry F., Murray D. G., Isaacs R., Isard M.* Differential Dataflow // 6th Biennial Conference on Innovative Data Systems Research (CIDR), January 2013 [Электронный ресурс]. — Режим доступа: http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf.
25. *Murray D. G., McSherry F., Isaacs R., et al.* Naiad: A Timely Dataflow System // 24th ACM Symposium on Operating Systems Principles (SOSP), pages 439–455, November 2013 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/.
26. *Shapira G.* We have a bunch of customers who are implementing 'database inside-out' concept and they all ask 'is anyone else doing it? are we crazy?' // twitter.com, July 28, 2016 [Электронный ресурс]. — Режим доступа: https://twitter.com/gwenshap/status/758800071110430720.
27. *Kleppmann M.* Turning the Database Inside-out with Apache Samza // Strange Loop, September 2014 [Электронный ресурс]. — Режим доступа: http://martin.kleppmann.com/2015/03/04/turning-the-database-inside-out.html.
28. *Roy van P., Haridi S.* Concepts, Techniques, and Models of Computer Programming. — MIT Press, 2004. https://epsa.org/forms/uploadFiles/3B6300000000.filename.booksingle.pdf.
29. Juttle Documentation. 2016 [Электронный ресурс]. — Режим доступа: http://juttle.github.io/juttle/.
30. *Czaplicki E., Chong S.* Asynchronous Functional Reactive Programming for GUIs // 34th ACM SIGPLAN Conference on Programming Language Design and

Implementation (PLDI), June 2013 [Электронный ресурс]. — Режим доступа: http://people.seas.harvard.edu/~chong/pubs/pldi13-elm.pdf.

31. *Bainomugisha E., Carreton A. L., Cutsem van T., Mostinckx S., Meuter de W.* A Survey on Reactive Programming // ACM Computing Surveys, volume 45, number 4, pages 1–34, August 2013 [Электронный ресурс]. — Режим доступа: http://soft.vub.ac.be/Publications/2012/vub-soft-tr-12-13.pdf.
32. *Alvaro P., Conway N., Hellerstein J. M., Marczak W. R.* Consistency Analysis in Bloom: A CALM and Collected Approach // 5th Biennial Conference on Innovative Data Systems Research (CIDR), January 2011 [Электронный ресурс]. — Режим доступа: http://www.eecs.berkeley.edu/~palvaro/cidr11.pdf.
33. *Hermans F.* Spreadsheets Are Code // Code Mesh, November 2015 [Электронный ресурс]. — Режим доступа: https://vimeo.com/145492419.
34. *Bricklin D., Frankston B.* VisiCalc: Information from Its Creators [Электронный ресурс]. — Режим доступа: http://danbricklin.com/visicalc.htm.
35. *Sculley D., Holt G., Golovin D., et al.* Machine Learning: The HighInterest Credit Card of Technical Debt // NIPS Workshop on Software Engineering for Machine Learning (SE4ML), December 2014 [Электронный ресурс]. — Режим доступа: https://research.google.com/pubs/pub43146.html.
36. *Bailis P., Fekete A., Franklin M. J., et al.* Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity // ACM International Conference on Management of Data (SIGMOD), June 2015 [Электронный ресурс]. — Режим доступа: http://www.bailis.org/papers/feral-sigmod2015.pdf.
37. *Steele G.* Re: Need for Macros (Was Re: Icon) // email to ll1-discuss mailing list, December 24, 2001 [Электронный ресурс]. — Режим доступа: https://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg01134.html.
38. *Gelernter D.* Generative Communication in Linda // ACM Transactions on Programming Languages and Systems (TOPLAS), volume 7, number 1, pages 80–112, January 1985 [Электронный ресурс]. — Режим доступа: http://cseweb.ucsd.edu/groups/csag/html/teaching/cse291s03/Readings/p80-gelernter.pdf.
39. *Eugster P. Th., Felber P. A., Guerraoui R., Kermarrec A.-M.* The Many Faces of Publish/Subscribe // ACM Computing Surveys, volume 35, number 2, pages 114–131, June 2003 [Электронный ресурс]. — Режим доступа: http://www.cs.ru.nl/~pieter/oss/manyfaces.pdf.
40. *Stopford B.* Microservices in a Streaming World // QCon London, March 2016 [Электронный ресурс]. — Режим доступа: https://www.infoq.com/presentations/microservices-streaming.
41. *Posta C.* Why Microservices Should Be Event Driven: Autonomy vs Authority. May 27, 2016 [Электронный ресурс]. — Режим доступа: http://blog.christianposta.com/microservices/why-microservices-should-be-event-driven-autonomy-vs-authority/.
42. *Feyerke A.* Say Hello to Offline First. November 5, 2013 [Электронный ресурс]. — Режим доступа: http://hood.ie/blog/say-hello-to-offline-first.html.

43. *Burckhardt S., Leijen D., Protzenko J., Fähndrich M.* Global Sequence Protocol: A Robust Abstraction for Replicated Shared State // 29th European Conference on Object-Oriented Programming (ECOOP), July 2015 [Электронный ресурс]. — Режим доступа: http://drops.dagstuhl.de/opus/volltexte/2015/5238/.
44. *Soper M.* Clearing Up React Data Management Confusion with Flux, Redux, and Relay. December 3, 2015 [Электронный ресурс]. — Режим доступа: https://medium.com/@marksoper/clearing-up-react-data-management-confusion-with-flux-redux-and-relay-aad504e63cae.
45. *Thereska E., Guy D., Noll M., Narkhede N.* Unifying Stream Processing and Interactive Queries in Apache Kafka. October 26, 2016 [Электронный ресурс]. — Режим доступа: https://www.confluent.io/blog/unifying-stream-processing-and-interactive-queries-in-apache-kafka/.
46. *McSherry F.* Dataflow as Database. July 17, 2016 [Электронный ресурс]. — Режим доступа: https://github.com/frankmcsherry/blog/blob/master/posts/2016-07-17.md.
47. *Alvaro P.* I See What You Mean // Strange Loop, September 2015 [Электронный ресурс]. — Режим доступа: https://www.youtube.com/watch?v=R2Aa4PivG0g.
48. *Marz N.* Trident: A High-Level Abstraction for Realtime Computation. August 2, 2012 [Электронный ресурс]. — Режим доступа: https://blog.twitter.com/engineering/en_us/a/2012/trident-a-high-level-abstraction-for-realtime-computation.html.
49. *Bice E.* Low Latency Web Scale Fraud Prevention with Apache Samza, Kafka and Friends // Merchant Risk Council MRC Vegas Conference, March 2016 [Электронный ресурс]. — Режим доступа: https://www.slideshare.net/edibice/extremely-low-latency-web-scale-fraud-prevention-with-apache-samza-kafka-and-friends.
50. *Majors C.* The Accidental DBA. October 2, 2016 [Электронный ресурс]. — Режим доступа: https://charity.wtf/2016/10/02/the-accidental-dba/.
51. *Bernstein A. J., Lewis P. M., Lu S.* Semantic Conditions for Correctness at Different Isolation Levels // 16th International Conference on Data Engineering (ICDE), February 2000 [Электронный ресурс]. — Режим доступа: http://db.cs.berkeley.edu/cs286/papers/isolation-icde2000.pdf.
52. *Jorwekar S., Fekete A., Ramamritham K., Sudarshan S.* Automating the Detection of Snapshot Isolation Anomalies // 33rd International Conference on Very Large Data Bases (VLDB), September 2007 [Электронный ресурс]. — Режим доступа: http://www.vldb.org/conf/2007/papers/industrial/p1263-jorwekar.pdf.
53. *Kingsbury K.* Jepsen blog post series, 2013–2016 [Электронный ресурс]. — Режим доступа: https://aphyr.com/tags/jepsen.
54. *Jouravlev M.* Redirect After Post. August 1, 2004 [Электронный ресурс]. — Режим доступа: http://www.theserverside.com/news/1365146/Redirect-After-Post.
55. *Saltzer J. H., Reed D. P., Clark D. D.* End-to-End Arguments in System Design // ACM Transactions on Computer Systems, volume 2, number 4, pages 277–288, November 1984.

56. *Bailis P., Fekete A., Franklin M. J., et al.* Coordination-Avoiding Database Systems // Proceedings of the VLDB Endowment, volume 8, number 3, pages 185–196, November 2014 [Электронный ресурс]. — Режим доступа: https://arxiv.org/pdf/1402.2237.pdf.

57. *Yarmula A.* Strong Consistency in Manhattan. March 17, 2016 [Электронный ресурс]. — Режим доступа: https://blog.twitter.com/engineering/en_us/a/2016/strong-consistency-in-manhattan.html.

58. *Terry D. B., Theimer M. M., Petersen K., et al.* Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System // 15th ACM Symposium on Operating Systems Principles (SOSP), pages 172–182, December 1995 [Электронный ресурс]. — Режим доступа: http://css.csail.mit.edu/6.824/2014/papers/bayou-conflicts.pdf.

59. *Gray J.* The Transaction Concept: Virtues and Limitations // 7th International Conference on Very Large Data Bases (VLDB), September 1981 [Электронный ресурс]. — Режим доступа: http://jimgray.azurewebsites.net/papers/thetransactionconcept.pdf.

60. *Garcia-Molina H., Salem K.* Sagas // ACM International Conference on Management of Data (SIGMOD), May 1987 [Электронный ресурс]. — Режим доступа: http://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf.

61. *Helland P.* Memories, Guesses, and Apologies. May 15, 2007 [Электронный ресурс]. — Режим доступа: https://blogs.msdn.microsoft.com/pathelland/2007/05/15/memories-guesses-and-apologies/.

62. *Kim Y., Daly R., Kim J., et al.* Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors // 41st Annual International Symposium on Computer Architecture (ISCA), June 2014 [Электронный ресурс]. — Режим доступа: https://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf.

63. *Seaborn M., Dullien T.* Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. March 9, 2015 [Электронный ресурс]. — Режим доступа: https://googleprojectzero.blogspot.co.uk/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

64. *Gray J. N., Ingen van C.* Empirical Measurements of Disk Failure Rates and Error Rates // Microsoft Research, MSR-TR-2005-166, December 2005 [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/en-us/research/publication/empirical-measurements-of-disk-failure-rates-and-error-rates/.

65. *Gurusami A., Price D.* Bug #73170: Duplicates in Unique Secondary Index Because of Fix of Bug#68021. July 2014 [Электронный ресурс]. — Режим доступа: https://bugs.mysql.com/bug.php?id=73170.

66. *Fredericks G.* Postgres Serializability Bug. September 2015 [Электронный ресурс]. — Режим доступа: https://github.com/gfredericks/pg-serializability-bug.

67. *Chen X.* HDFS DataNode Scanners and Disk Checker Explained. December 20, 2016 [Электронный ресурс]. — Режим доступа: http://blog.cloudera.com/blog/2016/12/hdfs-datanode-scanners-and-disk-checker-explained/.

68. *Kreps J.* Getting Real About Distributed System Reliability. March 19, 2012 [Электронный ресурс]. — Режим доступа: http://blog.empathybox.com/post/19574936361/getting-real-about-distributed-system-reliability.
69. *Fowler M.* The LMAX Architecture. July 12, 2011 [Электронный ресурс]. — Режим доступа: https://martinfowler.com/articles/lmax.html.
70. *Stokes S.* Move Fast with Confidence. July 11, 2016 [Электронный ресурс]. — Режим доступа: http://blog.samstokes.co.uk/blog/2016/07/11/move-fast-with-confidence/.
71. Sawtooth Lake Documentation // Intel Corporation, intelledger.github.io, 2016.
72. *Brown R. G.* Introducing R3 CordaTM: A Distributed Ledger Designed for Financial Services. April 5, 2016 [Электронный ресурс]. — Режим доступа: https://gendal.me/2016/04/05/introducing-r3-corda-a-distributed-ledger-designed-for-financial-services/.
73. *McConaghy T., Marques R., Müller A., et al.* BigchainDB: A Scalable Blockchain Database. June 8, 2016 [Электронный ресурс]. — Режим доступа: https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf.
74. *Merkle R. C.* A Digital Signature Based on a Conventional Encryption Function // CRYPTO '87, August 1987 [Электронный ресурс]. — Режим доступа: https://people.eecs.berkeley.edu/~raluca/cs261-f15/readings/merkle.pdf.
75. *Laurie B.* Certificate Transparency // ACM Queue, volume 12, number 8, pages 10–19, August 2014 [Электронный ресурс]. — Режим доступа: http://queue.acm.org/detail.cfm?id=2668154.
76. *Ryan M. D.* Enhanced Certificate Transparency and End-to-End Encrypted Mail // Network and Distributed System Security Symposium (NDSS), February 2014 [Электронный ресурс]. — Режим доступа: https://www.internetsociety.org/doc/enhanced-certificate-transparency-and-end-end-encrypted-mail.
77. Software Engineering Code of Ethics and Professional Practice // Association for Computing Machinery, 1999 [Электронный ресурс]. — Режим доступа: http://www.acm.org/about/se-code.
78. *Chollet F.* Software development is starting to involve important ethical choices. October 30, 2016 [Электронный ресурс]. — Режим доступа: https://twitter.com/fchollet/status/792958695722201088.
79. *Perisic I.* Making Hard Choices: The Quest for Ethics in Machine Learning. November 2016 [Электронный ресурс]. — Режим доступа: https://engineering.linkedin.com/blog/2016/11/making-hard-choices--the-quest-for-ethics-in-machine-learning.
80. *Naughton J.* Algorithm Writers Need a Code of Conduct. December 6, 2015 [Электронный ресурс]. — Режим доступа: https://www.theguardian.com/commentisfree/2015/dec/06/algorithm-writers-should-have-code-of-conduct.
81. *Kugler L.* What Happens When Big Data Blunders? // Communications of the ACM, volume 59, number 6, pages 15–16, June 2016 [Электронный ресурс]. — Режим доступа: https://cacm.acm.org/magazines/2016/6/202655-what-happens-when-big-data-blunders/abstract.

82. *Davidow B.* Welcome to Algorithmic Prison. February 20, 2014 [Электронный ресурс]. — Режим доступа: https://www.theatlantic.com/technology/archive/2014/02/welcome-to-algorithmic-prison/283985/.
83. *Peck D.* They're Watching You at Work. December 2013 [Электронный ресурс]. — Режим доступа: https://www.theatlantic.com/magazine/archive/2013/12/theyre-watching-you-at-work/354681/.
84. *Alexander L.* Is an Algorithm Any Less Racist Than a Human? August 3, 2016 [Электронный ресурс]. — Режим доступа: https://www.theguardian.com/technology/2016/aug/03/algorithm-racist-human-employers-work.
85. *Emspak J.* How a Machine Learns Prejudice. December 29, 2016 [Электронный ресурс]. — Режим доступа: https://www.scientificamerican.com/article/how-a-machine-learns-prejudice/.
86. *Cegłowski M.* The Moral Economy of Tech. June 2016 [Электронный ресурс]. — Режим доступа: http://idlewords.com/talks/sase_panel.htm.
87. *O'Neil C.* Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy. Crown Publishing, 2016. https://weaponsofmathdestructionbook.com/.
88. *Angwin J.* Make Algorithms Accountable. August 1, 2016 [Электронный ресурс]. — Режим доступа: http://www.nytimes.com/2016/08/01/opinion/make-algorithms-accountable.html.
89. *Goodman B., Flaxman S.* European Union Regulations on Algorithmic Decision-Making and a 'Right to Explanation'. August 31, 2016 [Электронный ресурс]. — Режим доступа: https://arxiv.org/abs/1606.08813.
90. A Review of the Data Broker Industry: Collection, Use, and Sale of Consumer Data for Marketing Purposes // Staff Report, United States Senate Committee on Commerce, Science, and Transportation, December 2013 [Электронный ресурс]. — Режим доступа: https://www.commerce.senate.gov/public/index.cfm/reports?ID=57C428EC-8F20-44EE-BFB8-A570E9BE0CCC.
91. *Solon O.* Facebook's Failure: Did Fake News and Polarized Politics Get Trump Elected? November 10, 2016 [Электронный ресурс]. — Режим доступа: https://www.theguardian.com/technology/2016/nov/10/facebook-fake-news-election-conspiracy-theories.
92. *Meadows D. H., Wright D.* Thinking in Systems: A Primer. — Chelsea Green Publishing, 2008.
93. *Bernstein D. J.* Listening to a 'big data'/'data science' talk. May 12, 2015 [Электронный ресурс]. — Режим доступа: https://twitter.com/hashbreaker/status/598076230437568512.
94. *Andreessen M.* Why Software Is Eating the World // The Wall Street Journal, 20 August 2011 [Электронный ресурс]. — Режим доступа: https://genius.com/Marc-andreessen-why-software-is-eating-the-world-annotated.
95. *Porup J. M.* 'Internet of Things' Security Is Hilariously Broken and Getting Worse. January 23, 2016 [Электронный ресурс]. — Режим доступа: https://

arstechnica.com/information-technology/2016/01/how-to-search-the-internet-of-things-for-photos-of-sleeping-babies/.

96. *Schneier B.* Data and Goliath: The Hidden Battles to Collect Your Data and Control Your World. W. W. Norton, 2015. https://www.schneier.com/books/data_and_goliath/.
97. *Grugq T.* Nothing to Hide. April 15, 2016 [Электронный ресурс]. — Режим доступа: https://grugq.tumblr.com/post/142799983558/nothing-to-hide.
98. *Beltramelli T.* Deep-Spying: Spying Using Smartwatch and Deep Learning // Masters Thesis, IT University of Copenhagen, December 2015 [Электронный ресурс]. — Режим доступа: https://arxiv.org/abs/1512.05616.
99. *Zuboff S.* Big Other: Surveillance Capitalism and the Prospects of an Information Civilization // Journal of Information Technology, volume 30, number 1, pages 75–89, April 2015 [Электронный ресурс]. — Режим доступа: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2594754.
100. *Zona C. C.* Consequences of an Insightful Algorithm // GOTO Berlin, November 2016 [Электронный ресурс]. — Режим доступа: https://www.youtube.com/watch?v=YRI40A4tyWU.
101. *Schneier B.* Data Is a Toxic Asset, So Why Not Throw It Out? March 1, 2016 [Электронный ресурс]. — Режим доступа: https://www.schneier.com/essays/archives/2016/03/data_is_a_toxic_asse.html.
102. *Dunn J. E.* The UK's 15 Most Infamous Data Breaches. November 18, 2016 [Электронный ресурс]. — Режим доступа: https://www.techworld.com/security/uks-most-infamous-data-breaches-3604586/.
103. *Scott C.* Data is not toxic — which implies no benefit — but rather hazardous material, where we must balance need vs. want. March 6, 2016 [Электронный ресурс]. — Режим доступа: https://twitter.com/cory_scott/status/706586399483437056.
104. *Schneier B.* Mission Creep: When Everything Is Terrorism. July 16, 2013 [Электронный ресурс]. — Режим доступа: https://www.schneier.com/essays/archives/2013/07/mission_creep_when_e.html.
105. *Ulbricht L., Grafenstein von M.* Big Data: Big Power Shifts? // Internet Policy Review, volume 5, number 1, March 2016 [Электронный ресурс]. — Режим доступа: https://policyreview.info/articles/analysis/big-data-big-power-shifts.
106. *Goodman E. P., Powles J.* Facebook and Google: Most Powerful and Secretive Empires We've Ever Known. September 28, 2016 [Электронный ресурс]. — Режим доступа: https://www.theguardian.com/technology/2016/sep/28/google-facebook-powerful-secretive-empire-transparency.
107. Directive 95/46/EC on the protection of individuals with regard to the processing of personal data and on the free movement of such data, Official Journal of the European Communities No. L 281/31, November 1995 [Электронный ресурс]. — Режим доступа: http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:31995L0046.

108. *Alsenoy van B.* Regulating Data Protection: The Allocation of Responsibility and Risk Among Actors Involved in Personal Data Processing // Thesis, KU Leuven Centre for IT and IP Law, August 2016 [Электронный ресурс]. — Режим доступа: https://lirias.kuleuven.be/handle/123456789/545027.

109. *Rhoen M.* Beyond Consent: Improving Data Protection Through Consumer Protection Law // Internet Policy Review, volume 5, number 1, March 2016 [Электронный ресурс]. — Режим доступа: https://policyreview.info/articles/analysis/beyond-consent-improving-data-protection-through-consumer-protection-law.

110. *Leber J.* Your Data Footprint Is Affecting Your Life in Ways You Can't Even Imagine. March 15, 2016 [Электронный ресурс]. — Режим доступа: https://www.fastcompany.com/3057514/your-data-footprint-is-affecting-your-life-in-ways-you-cant-even-imagine.

111. *Cegłowski M.* Haunted by Data. October 2015 [Электронный ресурс]. — Режим доступа: http://idlewords.com/talks/haunted_by_data.htm.

112. *Thielman S.* You Are Not What You Read: Librarians Purge User Data to Protect Privacy. January 13, 2016 [Электронный ресурс]. — Режим доступа: https://www.theguardian.com/us-news/2016/jan/13/us-library-records-purged-data-privacy.

113. *Friedersdorf C.* Edward Snowden's Other Motive for Leaking. May 13, 2014 [Электронный ресурс]. — Режим доступа: https://www.theatlantic.com/politics/archive/2014/05/edward-snowdens-other-motive-for-leaking/370068/.

114. *Rogaway P.* The Moral Character of Cryptographic Work // Cryptology ePrint 2015/1162, December 2015 [Электронный ресурс]. — Режим доступа: http://web.cs.ucdavis.edu/~rogaway/papers/moral-fn.pdf.

Мартин Клеппман

Высоконагруженные приложения. Программирование, масштабирование, поддержка

Перевели с английского И. Пальти, А. Тумаркин

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*С. Заматевская*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 191123, Россия, город Санкт-Петербург,
улица Радищева, дом 39, корпус Д, офис 415. Тел.: +78127037373.

Дата изготовления: 04.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Импортер в Беларусь: ООО «ПИТЕР М», РБ, 220020, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс 208 80 01.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Подписано в печать 29.03.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 51,600. Доп. тираж. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

---

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

ВАША УНИКАЛЬНАЯ КНИГА

Хотите издать свою книгу? Она станет идеальным подарком для партнеров и друзей, отличным инструментом для продвижения вашего бренда, презентом для памятных событий! Мы сможем осуществить ваши любые, даже самые смелые и сложные, идеи и проекты.

МЫ ПРЕДЛАГАЕМ:

- издать вашу книгу
- издание книги для использования в маркетинговых активностях
- книги как корпоративные подарки
- рекламу в книгах
- издание корпоративной библиотеки

Почему надо выбрать именно нас:

Издательству «Питер» более 20 лет. Наш опыт — гарантия высокого качества.

Мы предлагаем:

- услуги по обработке и доработке вашего текста
- современный дизайн от профессионалов
- высокий уровень полиграфического исполнения
- продажу вашей книги во всех книжных магазинах страны

Обеспечим продвижение вашей книги:

- рекламой в профильных СМИ и местах продаж
- рецензиями в ведущих книжных изданиях
- интернет-поддержкой рекламной кампании

Мы имеем собственную сеть дистрибуции по всей России, а также на Украине и в Беларуси. Сотрудничаем с крупнейшими книжными магазинами. Издательство «Питер» является постоянным участником многих конференций и семинаров, которые предоставляют широкую возможность реализации книг.

Мы обязательно проследим, чтобы ваша книга постоянно имелась в наличии в магазинах и была выложена на самых видных местах.

Обеспечим индивидуальный подход к каждому клиенту, эксклюзивный дизайн, любой тираж.

Кроме того, предлагаем вам выпустить электронную книгу. Мы разместим ее в крупнейших интернет-магазинах. Книга будет сверстана в формате ePub или PDF — самых популярных и надежных форматах на сегодняшний день.

Свяжитесь с нами прямо сейчас:

Санкт-Петербург – *Анна Титова, (812) 703-73-73, titova@piter.com*
Москва – *Сергей Клебанов, (495) 234-38-15, klebanov@piter.com*