

ВЫСОКО- НАГРУЖЕННЫЕ ПРИЛОЖЕНИЯ

Программирование
масштабирование
поддержка



 ПИТЕР®

Мартин Клепман

Designing Data-Intensive Applications

*The Big Ideas Behind Reliable, Scalable,
and Maintainable Systems*

Martin Kleppmann

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Мартин Клепман

ВЫСОКО- НАГРУЖЕННЫЕ ПРИЛОЖЕНИЯ

Программирование,
масштабирование,
поддержка



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.

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

ISBN 978-5-4461-0512-0

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 Кей¹. Из интервью Dr. Dobb's Journal²

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. nep.

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 действовать 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 онлайн/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 (серIALIZации) 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, кэшней, индексов и т. д. В главе 10 мы начнем с пакетного подхода к обработке производных данных и приедем 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), Гюльу 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), Джейкона Хомана (Jakob Homan), Гейди Говард (Heidi Howard), Джона Хагга (John Hugg), Джулиана Хайда (Julian Hyde), Конрада Ирвина (Conrad Irwin), Эвана Джонса (Evan Jones), Флавио Жункеира (Flavio Junqueira), Джессику Кэрр (Jessica Kerr), Кайла Кингсбэри (Kyle Kingsbury), Джая 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), Дженифер 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 Вонга (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 Чая (Joseph Chow), Дереку Элкинсу (Derek Elkins), Сэму Эллиотту (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 Krupa), 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 Суммерсу (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 Аластера 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 колossalное 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

НАДЕЖНОСТЬ

УДОБСТВО СОПРОВОЖДЕНИЯ

МАСШТАБИРУЕМОСТЬ

НАДЕЖНОСТЬ

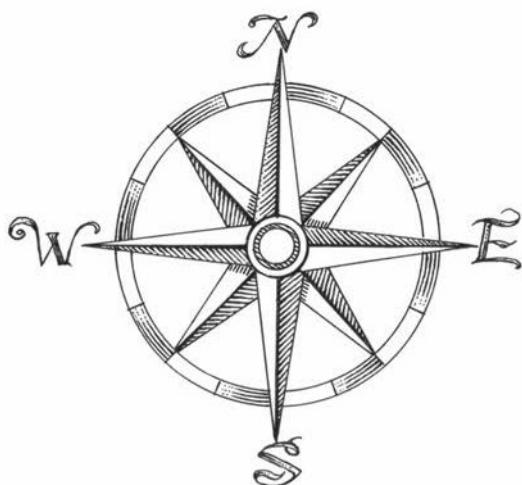
Устойчивость
к аппаратным
и программным
сбоям.
Человеческий
фактор

— 50 —

Показатели
нагрузки
и производи-
тельности.
Время ожидания,
процентили
и пропускная
способность

УДОБСТВО
СОПРОВОЖДЕНИЯ

Удобство
эксплуатации,
простота
и возможность
развития



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 Кей¹. Из интервью Dr. Dobb's Journal (2012)

Многие приложения сегодня относятся к категории *высоконагруженных данных* (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.

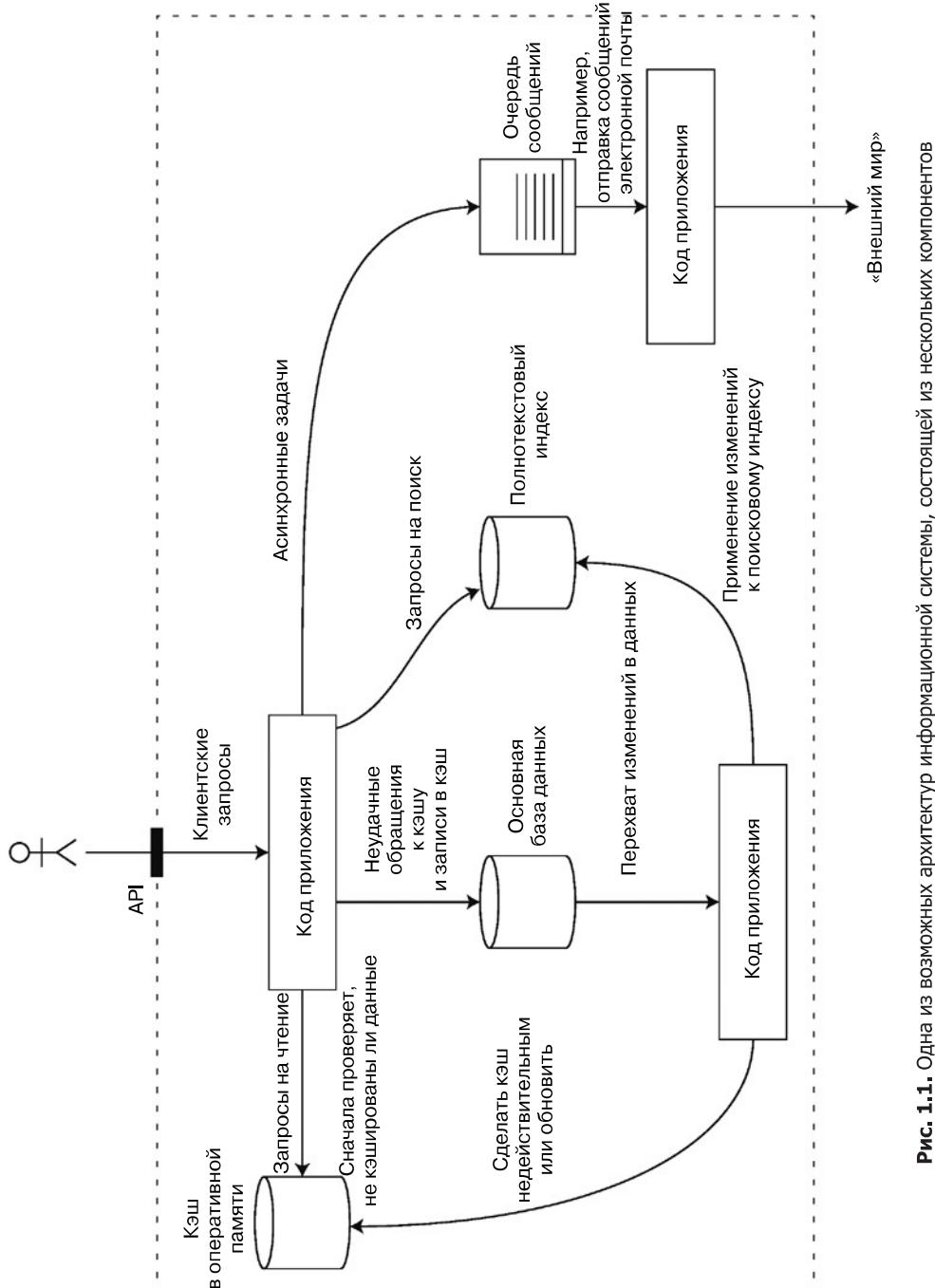


Рис. 1.1. Одна из возможных архитектур информационной системы, состоящей из нескольких компонентов

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 действуют 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 «естественному» 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 времяостоя 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 адаптироваться¹ 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 сбить. 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-«тесочницу», 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 управлеченческих 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 простой 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 больших объемов данных, чем ранее.

Масштабируемость (scalability) — термин, который я буду использовать для описания способности системы справляться с возросшей нагрузкой. Отмету, 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 подписчиков². 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 вентиляй, 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.*

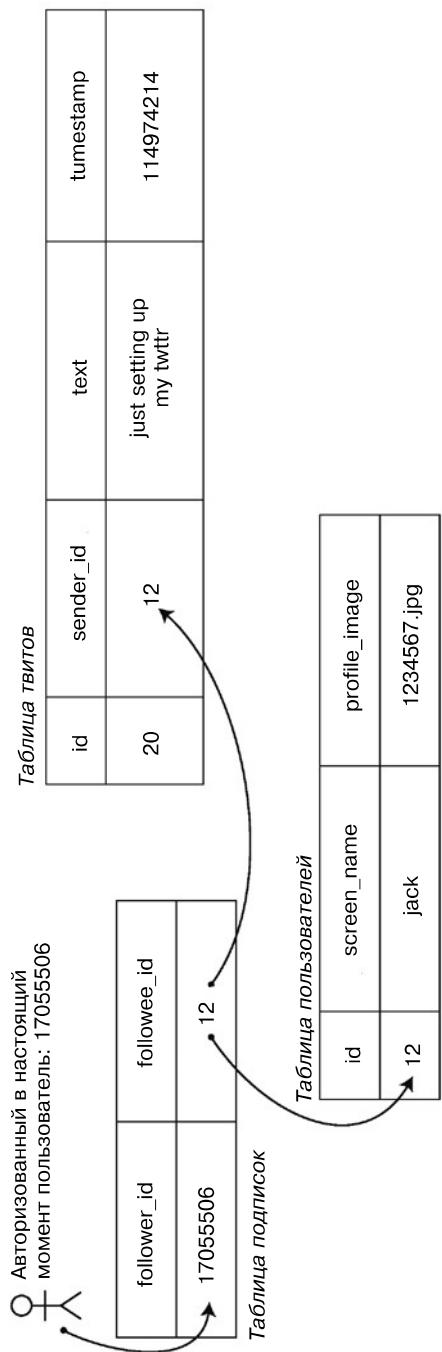


Рис. 1.2. Простая реляционная схема реализации ленты Twitter

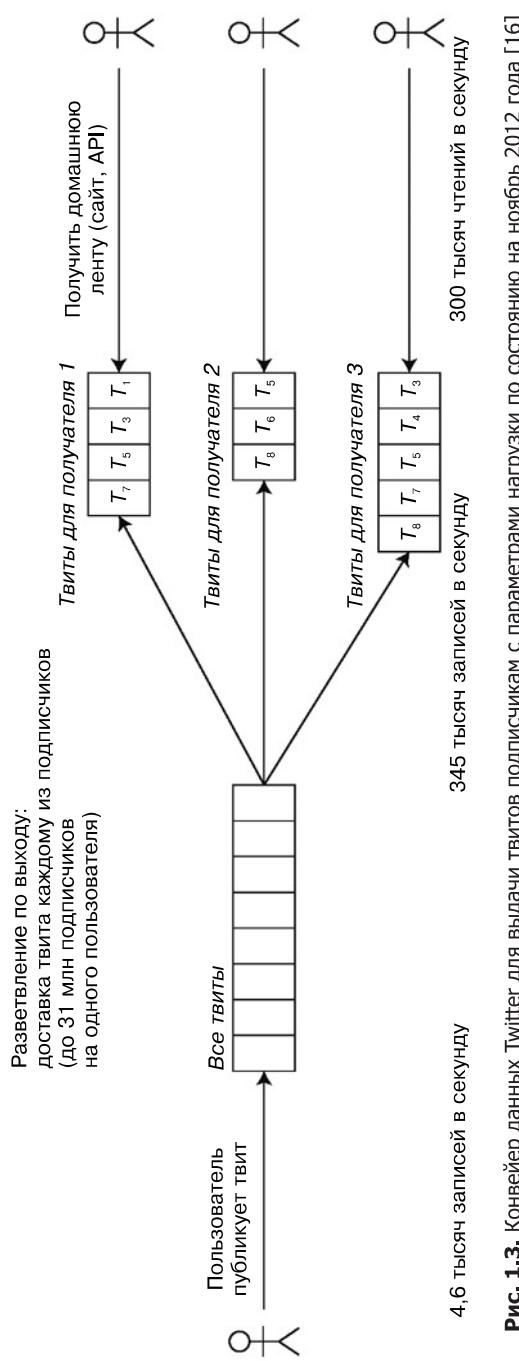


Рис. 1.3. Конвейер данных Twitter для выдачи твитов подписчикам с параметрами нагрузки по состоянию на ноябрь 2012 года [16]

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 размера¹. 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.

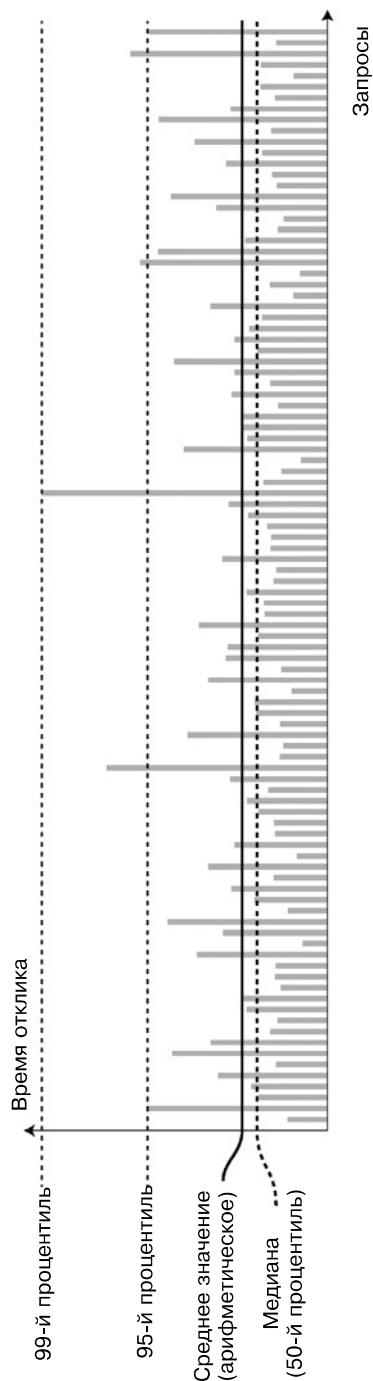


Рис. 1.4. Иллюстрирует среднее значение и процентили: время отклика для выборки из 100 запросов к сервису

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 кциальному 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 99.9x процентиляй, 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 достаточных 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 процентиляй 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 процентиляй при минимальных затратах процессорного времени и оперативной памяти, например forward decay [25], t-digest [26] и HdrHistogram [27]. Учтите, что усреднение процентиляй с целью понижения временного разрешения или объединения данных с нескольких машин математически бессмысленно — правильнее будет агрегировать данные о времени отклика путем сложения гистограмм [28].

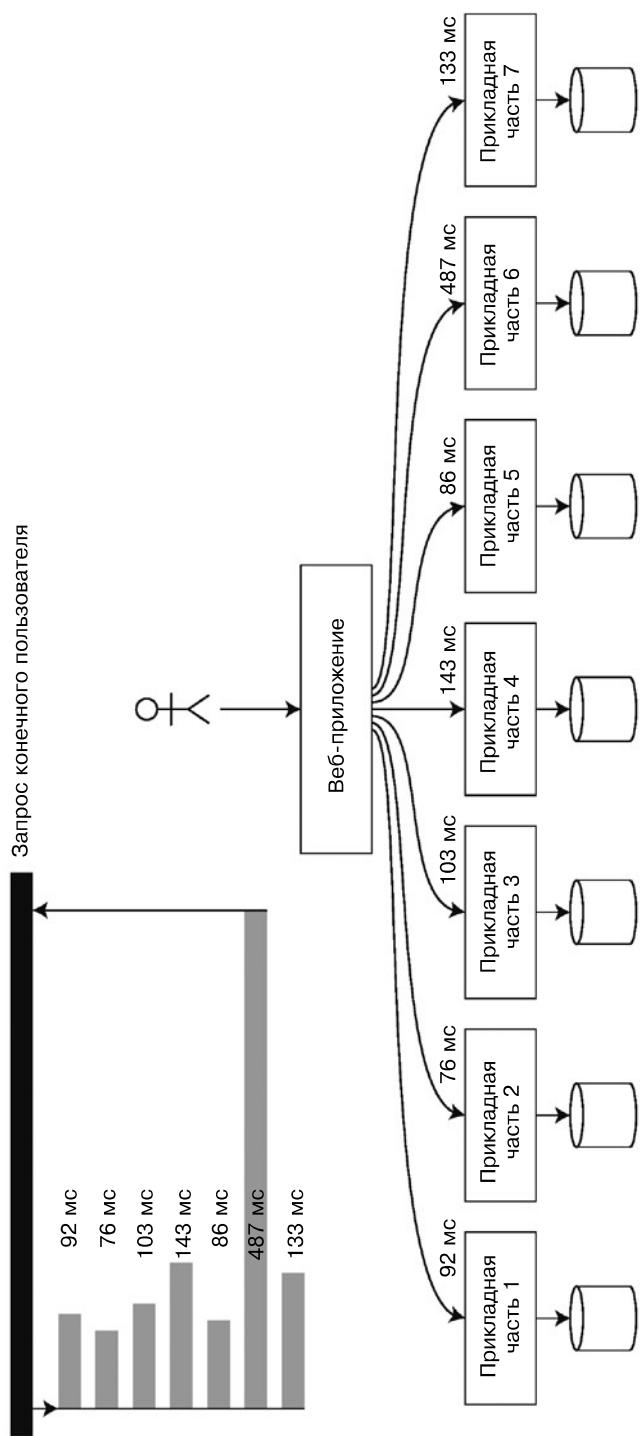


Рис. 1.5. Если для обслуживания запроса необходимо несколько обращений к прикладным сервисам, то один-единственный медленный прикладной запрос может затормозить работу всего запроса конечного пользователя

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 pragматичную 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 масштабирующими 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 не-проверенном 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 долга¹ 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 контролируются) 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/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/>.

победа•••

ПОСАДОЧНЫЙ ТАЛОН / BOARDING PASS

ИМЯ КЛИЕНТА / NAME OF CUSTOMER

ПОНАМАРЕВ СЕРГЕЙ ВЛАДИМИРОВИЧ

ОТ / FROM

МОСКВА / VKO

ТЕРМИНАЛ / TERMINAL

-

ДОКУМЕНТ, УДОСТОВЕРЯЮЩИЙ ЛИЧНОСТЬ / ID

4523617295

ДО / TO

МИНСК / MSQ

НОМЕР БИЛЕТА / TICKET NUMBER

ETKT4256117558824/1

РЕЙС / FLIGHT

DP-967

КЛАСС / CLASS

Э

ВЫЛЕТ / DEPARTURE

24 ЯНВ. 19:40

РЕГ.№ / REG

012

ВРЕМЯ ОКОНЧАНИЯ ПОСАДКИ / BOARDING CLOSING

19:20

ВЫХОД / GATE

23A

МЕСТО / SEAT

6C

Посадочный талон необходимо показать бортпроводникам при входе в самолёт

ВАЖНАЯ ИНФОРМАЦИЯ / IMPORTANT NOTICE

Обязательно распечатайте этот посадочный талон и возмите его с собой в аэропорт. Вы можете не распечатывать посадочный талон и показать его на мобильном устройстве при вылете из городов: Москва (Внуково и Шереметьево), Санкт-Петербург, Сочи, Алания, Волгоград, Екатеринбург, Казань, Калининград, Кемерово, Киров, Красноярск, Минеральные воды, Нижнекамск, Новосибирск, Омск, Пермь, Самара, Саратов, Сургут, Тюмень, Уфа, Челябинск - только внутренние рейсы, а также из Минска и Стамбула.

РУЧНАЯ КЛДЬ И ЛИЧНЫЕ ВЕЩИ

Габариты всех вещей, перевозимых клиентом, не должны превышать 36 x 30 x 27 сантиметров. Вещи по количеству и весу не ограничиваются, но все вместе должны свободно помещаться в калибратор (измеритель) с логотипом авиакомпании, установленный в аэропорту. Крышка калибратора с вещами должна плотно закрываться без применения давления на неё.

[Подробнее →](#)

БАГАЖ

Если вы летите с багажом, сдайте его на стойке оформления багажа (Drop-Off) не позднее 40 минут до вылета. За 60 минут до вылета – Стамбул и Дубай. Правила перевозки багажа – [Подробнее →](#)

ВЫХОД НА ПОСАДКУ

Номер выхода на посадку уточните на информационном табло в аэропорту вылета.

ВНИМАНИЕ: Курение в самолёте строго запрещено. Пересадка на борту запрещена, за исключением выполнения требований экипажа. СЧАСТЛИВОГО ПОЛЁТА!

IMPORTANT INFORMATION ABOUT THE FLIGHT

Be sure to print out this boarding pass and take it with you to the airport. You can not print the boarding pass and show it on your mobile device when departing from the cities: Moscow (Vnukovo and Sheremetyevo), Volgograd, Yekaterinburg, Kazan, Kaliningrad, Kirov, Krasnoyarsk, Mineralnye Vody, Nizhnekamsk, Novosibirsk, Omsk, Orenburg, Perm, Samara, Saratov, Surgut, Tyumen, Ufa, Chelyabinsk, Saint Petersburg.

HAND LUGGAGE AND CARRIED ITEMS IN THE CABIN

Dimensions should not exceed 36 x 30 x 27 cm (the total size of all items). Hand luggage and carried items in the cabin are not limited in quantity and weight, but must be freely placed in a calibrator (hand luggage meter) with the Pobeda logo installed at the airport. The calibrator lid with hand luggage and carried items in the cabin must be tightly closed without applying pressure on it.

[Learn more →](#)

BAGGAGE

If you are traveling with baggage, check it in at the Drop-off no later than 40 minutes before departure. 60 minutes before departure – Istanbul and Dubai. Baggage transportation rules. [Learn more →](#)

BOARDING GATE

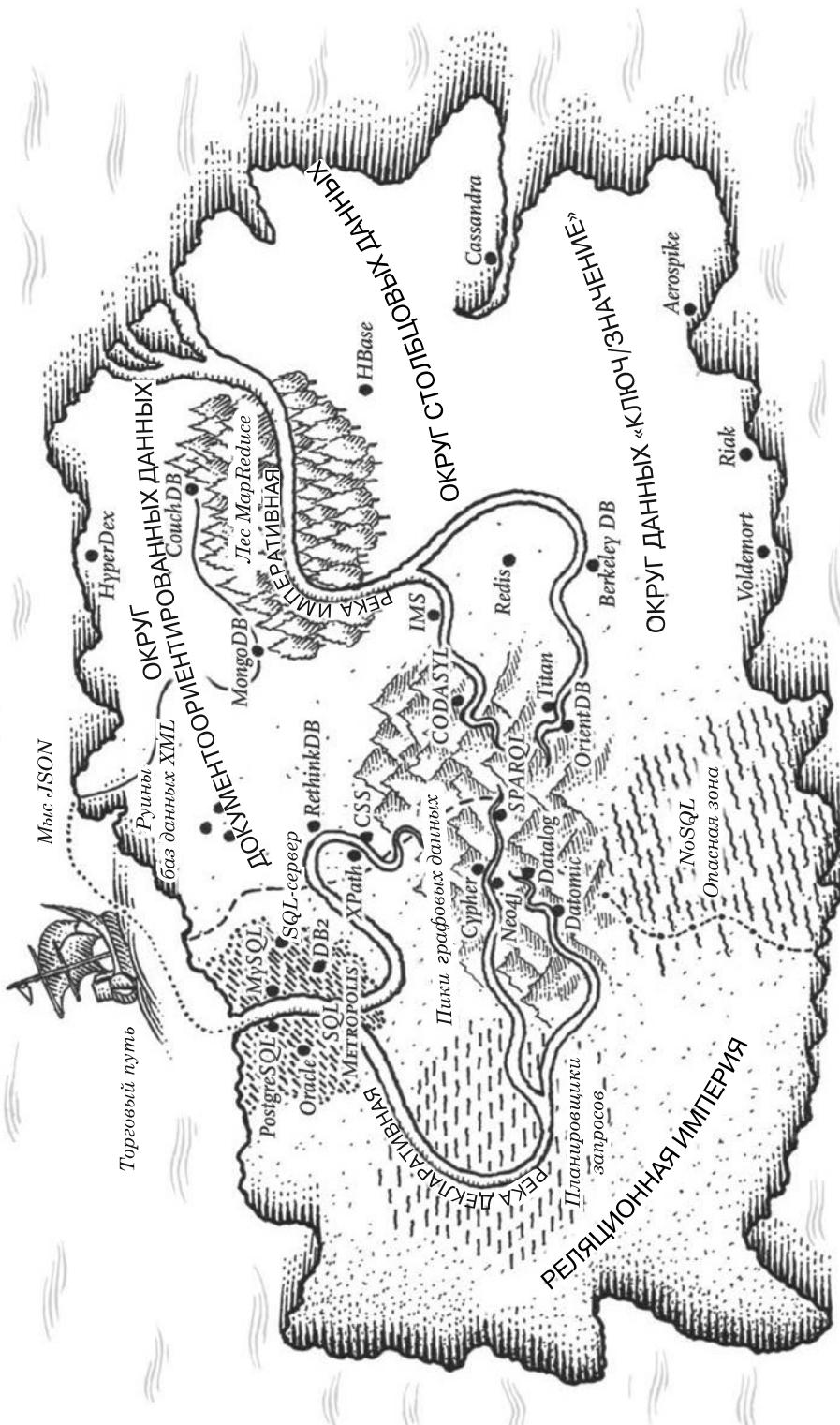
Check the boarding gate number on the information board at the airport of departure

ATTENTION: Smoking is strictly prohibited on the plane. Transfer on board is prohibited, except for the requirements of the crew.

WE WISH YOU A PLEASANT FLIGHT!

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/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>.



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 наслаждения 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 сети 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 больших 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 идеи 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 неуклонный промежуточный слой между объектами кода приложения и моделью таблиц, строк и столбцов БД. Этую расстыковку моделей иногда называют *рассогласование* (*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-стандарт 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 некоторыми 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.

<http://www.linkedin.com/in/williamhgates>



Bill Gates
Greater Seattle Area | Philanthropy

Summary
Co-chair of the Bill & Melinda Gates Foundation. Chairman, Microsoft Corporation. Voracious reader. Avid traveler. Active blogger.

Experience
Co-chair • Bill & Melinda Gates Foundation 2000 – Present
Co-founder, Chairman • Microsoft 1975 – Present

Education
Harvard University 1973 – 1975
Lakeside School, Seattle

Contact Info
Blog: thegatesnotes.com
Twitter: @BillGates

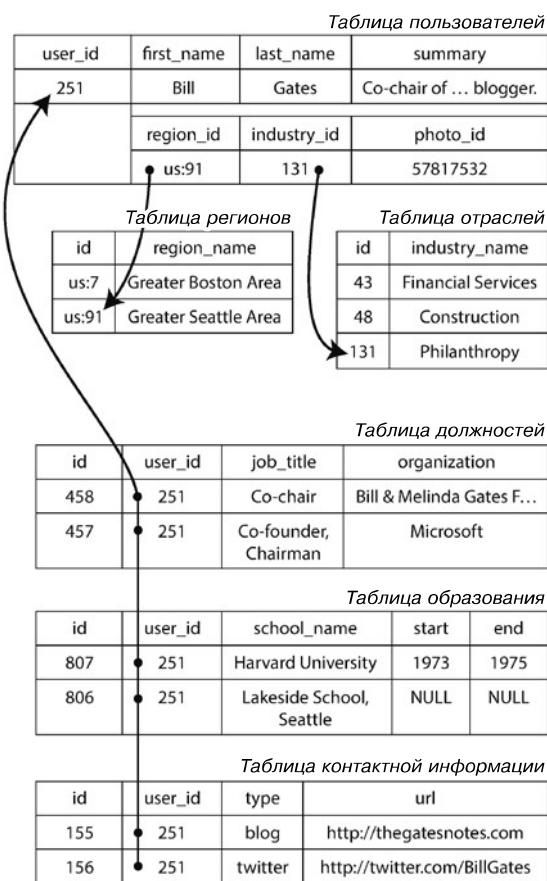


Рис. 2.1. Представление профиля LinkedIn с помощью реляционной схемы. Фото Билла Гейтса любезно предоставлено «Вики-складом», фотограф — Рикардо Стикерт, информагентство «Агенсия Бразил»

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. Этую 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}  
],  

```

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).

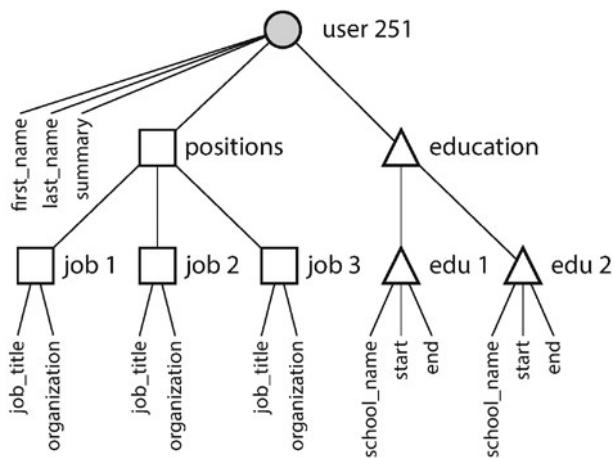


Рис. 2.2. Связи «один-ко-многим» формируют древовидную структуру

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 стандартизованных 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 стандартизованные 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 данных¹.



Администраторы и разработчики БД обожают спорить о нормализации и де-нормализации, 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 слаба².

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.