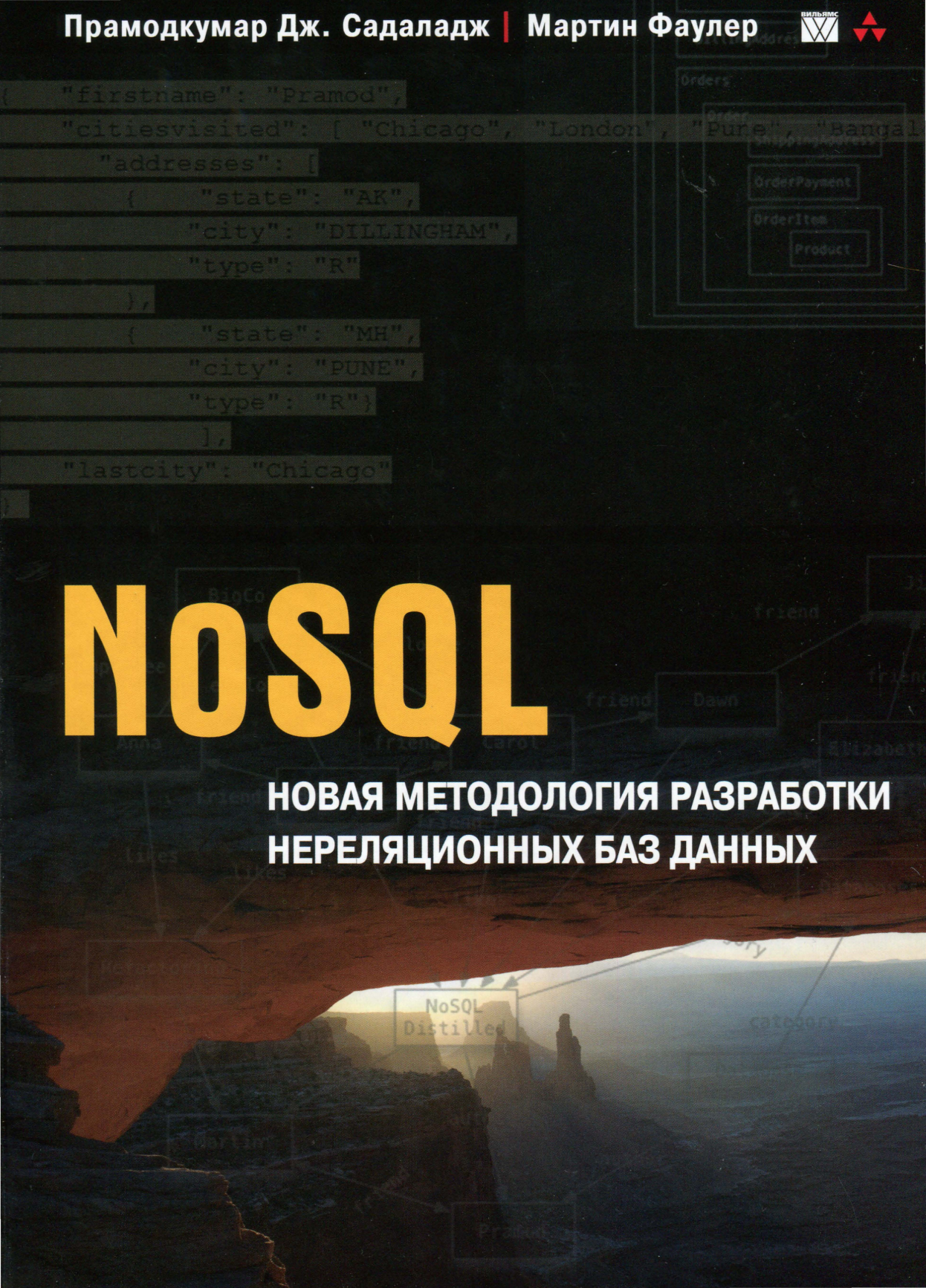
Прамодкумар Дж. Садаладж | Мартин Фаулер



NoSQL

НОВАЯ МЕТОДОЛОГИЯ РАЗРАБОТКИ НЕРЕЛЯЦИОННЫХ БАЗ ДАННЫХ

NoSQL DISTILLED

A Brief Guide to the Emerging World of Polyglot Persistence

Pramod J. Sadalage
Martin Fowler

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

NoSQL

НОВАЯ МЕТОДОЛОГИЯ РАЗРАБОТКИ НЕРЕЛЯЦИОННЫХ БАЗ ДАННЫХ

Прамодкумар Дж. Садаладж
Мартин Фаулер

Москва • Санкт-Петербург • Киев
2013

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского и редакция докт. физ.-мат. наук *Д.А. Клюшина*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресам:
info@williamspublishing.com, http://www.williamspublishing.com

Фаулер, Мартин, **Садаладж**, Прамодкумар Дж.

NoSQL: новая методология разработки нереляционных баз данных. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2013. — 192 с.: ил. — Парал. тит. англ.

Научно-популярное издание

Мартин Фаулер, Прамодкумар Дж. Садаладж

NoSQL: новая методология разработки нереляционных баз данных

Литературный редактор *И.А. Попова*
Верстка *Л.В. Чернокозинская*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

Подписано в печать 27.02.2013. Формат 70x100/16.
Гарнитура APCGaramond. Печать офсетная.
Усл. печ. л. 15,48. Уч.-изд. л. 10,1.
Тираж 1000 экз. Заказ № 3534.

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9-я линия, 12/28

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

Оглавление

Содержание

Посвящается моим учителям Гаджанану Чинчвадкару,
Даттатрейе Мхаскару и Арвинду Парчуру.
Вы вдохновляли меня больше всех, спасибо.

Прамод

Посвящается Синди

Мартин

Предисловие

Мы работаем в области обработки данных предприятий более двадцати лет и стали свидетелями многочисленных изменений в языках программирования, архитектурах, платформах и процессах. Тем не менее на протяжении всего этого времени одно оставалось неизменным — хранение данных в реляционных базах. Было несколько попыток изменить это положение, и некоторые из них достигли успеха в отдельных нишах рынка, но в целом хранение данных с точки зрения архитектуры всегда сводилось к использованию реляционных баз данных.

Стабильность такой ситуации обеспечивает много преимуществ. Данные организаций существуют намного дольше, чем программы, которые их обрабатывают (по крайней мере, так говорят многие люди, хотя нам доводилось видеть множество очень старых программ). Это неплохо, что существуют стабильные хранилища данных, понятные и доступные для многих платформ прикладного программирования.

Однако сейчас появился новый конкурент, носящий дерзкое имя NoSQL. Он появился в ответ на необходимость обрабатывать более крупные объемы данных, которая вынуждает нас сделать фундаментальный сдвиг в сторону построения более крупных аппаратных платформ, состоящих из кластеров обычных серверов. В связи с этим обострились старые проблемы, связанные с обеспечением эффективной работы прикладных программ в рамках реляционной модели данных.

Термин "NoSQL" выбран очень неудачно. Он относится ко многим новым нереляционным базам данных, таким как Cassandra, Mongo, Neo4J и Riak. Они содержат неструктурированные данные, работают на кластерах и обеспечивают компромисс между традиционной согласованностью и другими полезными свойствами. Сторонники баз данных NoSQL утверждают, что могут создавать намного более высококачественные, намного лучше масштабируемые и легче программируемые системы.

Можно ли это считать предзнаменованием конца эпохи реляционных баз данных или просто еще одним претендентом на трон? Наш ответ: "Ни тем ни другим". Реляционные базы данных — это мощный инструмент, который будет использоваться еще много десятилетий, но в результате происходящих изменений эти базы будут не единственными. С нашей точки зрения, мы вступаем в эру многовариантной персистентности (Polyglot Persistence), в которой предприятия и даже отдельные приложения будут использовать для управления данными несколько технологий. В результате архитекторы баз данных будут вынуждены осваивать эти технологии и оценивать их в соответствии со своими потребностями. Если бы мы так не думали, то не стали бы тратить время и силы на эту книгу.

Книга содержит достаточно много информации, чтобы ответить на вопрос, *стоят* ли базы данных NoSQL серьезного анализа для использования в будущих проектах. Каждый проект имеет свои отличительные особенности, поэтому невозможно нарисовать простое дерево решений для выбора правильного способа хранения данных.

Вместо этого мы попытались изложить принципы работы баз данных NoSQL, чтобы вы могли сделать самостоятельные выводы, не прибегая к сканированию веб. Мы намеренно написали небольшую книгу, чтобы вы могли достаточно быстро освоить эту информацию. Разумеется, в книге нет решений на все случаи жизни, но она должна сузить область поиска, которую вы должны исследовать, и помочь вам правильно сформулировать свои вопросы.

Чем интересны базы данных NoSQL

Есть две причины, по которым люди рассматривают возможность использовать базы данных NoSQL.

- **Эффективность разработки приложений.** Большинство усилий, связанных с разработкой приложений, затрачиваются на отображение данных из структур, хранящихся в памяти, в реляционные базы данных. База данных NoSQL может обеспечить модель данных, лучше удовлетворяющую потребности приложения, упростив тем самым это взаимодействие и уменьшив количество кода, который необходимо написать, отладить и развить.
- **Крупномасштабные данные.** Организации ценят возможность хранить более крупные объемы данных и быстрее их обрабатывать. Они считают слишком затратным использовать для этого реляционные базы данных. Основная причина заключается в том, что реляционные базы данных предназначены для работы на одном компьютере, в то время как большие объемы данных и программы для их обработки экономнее хранить на кластерах, состоящих из многочисленных небольших и дешевых компьютеров. Многие базы данных NoSQL разработаны специально для кластеров, поэтому они лучше вписываются в сценарии обработки больших объемов данных.

Краткое содержание книги

Книга состоит из двух частей. В первой части излагаются основные концепции, которые, по нашему мнению, необходимо знать, чтобы правильно оценивать возможность использования баз данных NoSQL в своих проектах и понимать, чем они отличаются от остальных. Во второй части мы сосредоточились на реализации систем с базами данных NoSQL.

Глава 1 начинается с объяснения быстрого роста популярности баз данных NoSQL — необходимость обрабатывать более крупные объемы данных в больших системах стимулировала переход от вертикального масштабирования к горизонтальному масштабированию на кластерах. Этим объясняется важная особенность многих баз данных NoSQL — явное хранение емких структур тесно связанных между собой данных, доступных как одно целое. В нашей книге мы называем такие структуры *агрегатами* (aggregate).

В главе 2 описывается, как агрегаты проявляются в трех основных моделях данных NoSQL: базы данных типа "ключ–значение" и документные базы данных ("Модели "ключ–значение" и документные модели данных"), а также семейство столбцов ("Семейства столбцов"). Агрегаты обеспечивают естественное взаимодействие различных приложений. Это одновременно ускоряет работу на кластерах и облегчает программам доступ к данным. В главе 3 рассматривается недостаток агрегатов — сложность выражения отношений ("Отношения") между сущностями в разных агрегатах. Это естественным образом приводит к графовым базам данных ("Графовые базы данных"), модели данных NoSQL, не соответствующей принципам, ориентированным на агрегаты. Мы также рассматриваем общую характеристику баз данных NoSQL — отказ от использования схем ("Неструктурированные базы данных"), который обеспечивает повышенную гибкость, но не настолько высокую, как можно было бы предположить.

Описав аспекты моделей данных в базах NoSQL, мы перейдем к их распределению: в главе 4 описывается, как база распределяет данные по кластерам. Эта процедура распадается на фрагментацию ("Фрагментация") и репликацию, которая может выполняться по схеме "ведущий–ведомый" (master-slave) ("Репликация ведущий–ведомый") или быть одноранговой (peer-to-peer) ("Одноранговая репликация"). Определив модели распределения, мы можем перейти к изучению согласованности. Благодаря ориентации на кластеры базы данных NoSQL обеспечивают более широкий выбор вариантов согласованности по сравнению с реляционными базами данных. В главе 5 описывается, как найти компромисс между изменениями согласованности при обновлении ("Согласованность при обновлении") и чтении ("Согласованность при чтении"), ролью кворумов ("Кворумы") и даже долговечностью ("Ослабление требования долговечности"). Если вы уже слышали что-либо о базе данных NoSQL, то почти наверняка слышали о теореме CAP; ее смысл и применение описывается в разделе "Теорема CAP".

Перечисленные выше главы посвящены в основном принципам распределения и сохранения согласованности данных, а в следующих двух главах описывается набор важных инструментов, выполняющих эту работу. В главе 6 описываются метки версий, отслеживающие изменения и устраняющие несогласованность. В главе 7 приводится краткий очерк организации параллельных вычислений, ориентированных на кластеры, а значит, и на системы NoSQL.

Освоив эти концепции, мы перейдем к вопросам их реализации, рассматривая конкретные примеры баз данных, относящихся к четырем ключевым категориям: в главе 8 в качестве базы данных типа "ключ–значение" используется база Riak, в главе 9 в качестве примера документной базы рассматривается база MongoDB, в главе 10 в качестве примера базы данных "семейство столбцов" описывается база Cassandra, а в главе 11 изучается графовая база данных Neo4J. Следует подчеркнуть, что это не исчерпывающее описание — за его пределами осталось много тем. Наш выбор примеров не следует рассматривать как рекомендации. Нашей целью было дать вам почувствовать разнообразие вопросов, существующих в этой области, и продемонстрировать применение описанных выше концепций в разных технологиях баз данных. Вы увидите, какие программы вам придется написать для этих систем, и получите представление об основных идеях, лежащих в их основе.

Принято считать, что благодаря отсутствию схемы в базах данных NoSQL можно легко изменять структуры данных на протяжении всего срока функционирования соответствующего приложения. Мы с этим не согласны — неструктурированная база данных имеет неявную схему, которая требует соблюдения принципов ее изменения при реализации, поэтому в главе 12 объясняется перенос данных как в системах со строгими схемами, так и в неструктурированных системах.

Благодаря всему сказанному становится ясно, что база данных NoSQL — не отдельная сущность и не может заменить реляционную базу данных. В главе 13 описывается будущая эра многовариантной персистентности, в которой будут сосуществовать разные способы хранения данных, даже в одном и том же приложении. Глава 14 расширяет горизонты книги и описывает другие технологии, которые не рассмотрены в книге, но также могут быть частью мира многовариантной персистентности.

Владея всей этой информацией, вы сможете делать осознанный выбор технологий хранения данных, поэтому заключительная глава ("Выбор базы данных") содержит несколько советов, касающихся этого выбора. С нашей точки зрения, существуют два главных фактора — определение эффективной программной модели, в которой модель хранения данных хорошо согласована с приложением, и обеспечение эффективного и надежного доступа к данным. Поскольку эра баз данных NoSQL только начинается, мы еще не имеем хорошо определенных процедур и должны согласовывать возможности со своими потребностями.

Это лишь краткий обзор — мы совершенно сознательно ограничили объем книги, выбрав лишь самую важную информацию. Если вы решили глубоко изучить эти технологии, то вам следует продолжить обучение, но мы надеемся, что эта книга станет хорошей отправной точкой на вашем пути.

Мы также хотели бы подчеркнуть, что базы данных NoSQL образуют очень изменчивую область компьютерной индустрии. Изменения в ней происходят ежегодно — появляются новые функциональные возможности и базы данных. Мы сделали основной акцент на концепциях, потому что их понимание является важным независимо от изменений соответствующих технологий. Мы убеждены, что большинство из написанного нами будет иметь значение еще долгое время, но абсолютно уверены, что проверку временем пройдет не все.

Для кого предназначена книга

Целевой аудиторией книги являются люди, так или иначе использующие базы данных NoSQL. Ее можно использовать как для создания нового проекта, так и для совершенствования существующего.

Мы стремились дать вам достаточно информации, чтобы вы могли осознанно решить, нужна ли технология NoSQL для удовлетворения ваших потребностей и какие инструменты следует изучить глубже. В качестве основных читателей мы

представляли себе архитекторов или технических руководителей, но думаем, что эта книга будет полезной всем, кто занимается управлением программным обеспечением и хочет получить представление о новой технологии. Мы считаем, что книга может стать хорошей отправной точкой для разработчиков, желающих овладеть новой технологией.

Мы не углублялись в детали программирования и развертывания конкретных баз данных — оставим эти вопросы для специальных книг. Мы стремились экономить объем, чтобы книга получилась краткой. Такую книгу удобно читать в самолете: она не стремится ответить на все вопросы, а дает читателям представление о том, какие вопросы он должен поставить.

Если вы уже работаете в области баз данных NoSQL, то эта книга, вероятно, ничего не прибавит к вашим знаниям. Тем не менее она будет полезной, поскольку поможет вам объяснить другим основные принципы технологии NoSQL. Очень важно научиться объяснять основные принципы технологии NoSQL, особенно если вы пытаетесь уговорить кого-то рассмотреть возможность использования баз данных NoSQL в проекте.

Что такое базы данных

В этой книге мы следуем основным принципам классификации баз данных NoSQL по их моделям данных. Ниже приведена таблица, содержащая четыре модели данных и базы данных, соответствующие этим моделям. Это не исчерпывающий список — он лишь напоминает о базах данных, не рассмотренных в книге. Пока мы писали книгу, полный список баз данных NoSQL хранился на веб-сайтах `http://nosql-database.org` и `http://nosql.mypopescu.com/kb/nosql`. В каждой категории мы пометили курсивом базу данных, рассмотренную в соответствующей главе.

Мы стремились выбрать репрезентативный инструмент для каждой категории баз данных. Говоря о конкретных примерах, мы имеем в виду всю категорию, даже если рассматриваемая база данных является уникальной и не допускает обобщений. Мы рассмотрели по одному примеру из категорий баз данных типа "ключ–значение", документных баз данных, семейств столбцов и графовых баз данных. Там, где это было возможно, мы упоминали другие продукты, которые могут удовлетворить конкретные потребности пользователей.

Эта классификация по моделям данных является полезной, но грубой. Границы между разными моделями данных, например, между базами данных типа "ключ–значение" и документыми ("Модели "ключ–значение" и документные модели данных"), часто размыты. Многие базы данных не укладываются в рамки одной категории; например, база данных OrientDB одновременно относится к категориям документных и графовых баз данных.

Пример базы данных	Модель данных
BerkeleyDB Key-Value ("Базы данных типа "ключ–значение")	Ключ–значение ("Базы данных типа "ключ–значение")
	LevelDB
	Memcached
	Project Voldemort
	Redis
	Riak
CouchDB Document ("Документные базы данных")	MongoDB
	OrientDB
	RavenDB
	Terrastore
Amazon SimpleDB Column-Family ("Семейства столбцов")	Cassandra
	HBase
	Hypertable
FlockDB Graph ("Графовые базы данных")	HyperGraphDB
	Infinite Graph
	Neo4J
	OrientDB

Благодарности

В первую очередь мы благодарим наших коллег из компании ThoughtWorks, многие из которых применяли базы данных NoSQL в своих проектах на протяжении последних лет. Их опыт был основным стимулом как для написания нашей книги, так и источником информации о ценности этой технологии. Положительный опыт, полученный при работе с базами NoSQL, стал основой нашего убеждения в том, что эта технология является фундаментальным сдвигом в области хранения данных.

Мы хотели бы поблагодарить разные группы людей, участвовавших в публичных дискуссиях, а также публиковавших статьи и блоги, посвященные использованию баз данных NoSQL. Основной прогресс в области разработки программного обеспечения остается скрытым, если люди не делятся своим опытом с коллегами. Особую благодарность выражаем компаниям Google и Amazon, опубликовавшим статьи о базах данных Bigtable и Dynamo. Эти статьи оказали большое стимулирующее влияние на развитие технологии NoSQL. Мы благодарим компании, финансирующие разработку баз данных NoSQL с открытым исходным кодом. Интересным отличием от предыдущих изменений в хранении данных является то, что технология NoSQL разрабатывалась как проект с открытым исходным кодом.

Отдельное спасибо компании ThoughtWorks за то, что она предоставила нам время для работы над книгой. Мы поступили на работу в компанию ThoughtWorks примерно в одно и то же время и работали в ней более десяти лет. Компания ThoughtWorks продолжает быть для нам радушным приютом, источником знаний и практического

опыта, а также гостеприимной средой для открытого обмена знаниями. Этим она сильно отличается от традиционных организаций, поставляющих системы баз данных.

Бетани Андерс–Бек (Bethany Anders-Beck), Илияс Бартолини (Ilias Bartolini), Тим Берглунд (Tim Berglund), Дункан Крейг (Duncan Craig), Поль Дюваль (Paul Duvall), Орен Эйни (Oren Eini), Перрин Фаулер (Perryn Fowler), Майкл Хангер (Michael Hunger), Эрик Кашич (Eric Kascic), Джошуа Кериевски (Joshua Kerievsky), Ананд Кришнасвами (Anand Krishnaswamy), Бобби Нортон (Bobby Norton), Аде Ошини (Ade Oshineye), Тьягу Паланисвами (Thiyagu Palanisamy), Прасанна Пендсе (Prasanna Pendse), Дан Притчетт (Dan Pritchett), Дэвид Райс (David Rice), Майк Робертс (Mike Roberts), Марко Родригес (Marko Rodriquez), Эндрю Слокум (Andrew Slocum), Тоби Трипп (Toby Tripp), Стив Виноски (Steve Vinoski), Дин Вамплер (Dean Wampler), Джим Уэббер (Jim Webber) и Уи Виттаваскул (Wee Witthawaskul) прочитали первые черновики книги и помогли нам своими советами.

Кроме того, Прамод хотел бы поблагодарить библиотеку Шамбурга (Schaumburg Library) за прекрасное обслуживание и уютное место для работы, прекрасных дочерей Архану и Арулу за понимание того, что папа должен идти в библиотеку и не может их взять с собой, а также любимую жену Рупали за безмерную поддержку и помощь.

Часть I

Основы

Глава 1

Почему NoSQL?

Почти все время нашей работы в сфере разработки программного обеспечения реляционные базы данных по умолчанию были основным способом хранения данных в серьезных проектах, особенно в области промышленных приложений. Если вы — архитектор, начинающий новый проект, то скорее всего выберете реляционную базу. (Кроме того, выбор базы данных часто диктуется наличием у компании доминирующего поставщика.) Иногда появлялись другие технологии баз данных, например объектные базы данных в 1990-х годах, но эти альтернативы не получили широкого распространения.

После долгого периода подавляющего преобладания появление сильного конкурента в виде баз данных NoSQL стало сюрпризом. В этой главе мы рассмотрим, почему реляционные базы стали настолько широко распространенными, и попробуем объяснить, почему мы считаем, что базы данных NoSQL ждет успех.

1.1. Значение реляционных баз данных

Реляционные базы данных стали настолько привычной частью обработки данных, что их можно принять просто как неизбежную данность. Тем более полезно понять, какие преимущества они имеют.

1.1.1. Персистентные данные

Вероятно, самым главным преимуществом баз данных является возможность долговременного хранения больших объемов данных. Большинство компьютерных архитектур подразумевают два вида памяти, быструю и изменчивую основную память, и более крупное и медленное резервное хранилище.

Основная память ограничена по объему и может потерять все данные при отключении электропитания или сбое в работе операционной системы. Следовательно, чтобы сохранить данные надолго, необходимо записывать их в резервное хранилище, в роли которого обычно выступает жесткий диск (но только на тот период времени, в течение которого диск физически может хранить данные).

Резервное хранилище может быть организовано разными способами. Для большинства высокопроизводительных приложений (например, для текстовых процессоров)

резервным хранилищем является файл, записанный в файловой системе. Однако для большинства промышленных приложений резервным хранилищем является база данных. Это обеспечивает более высокую гибкость при хранении крупных объемов данных по сравнению с файловой системой в том смысле, что база данных позволяет прикладным программам быстро и легко получать небольшие порции информации.

1.1.2. Параллельность

Промышленные приложения обычно позволяют многим клиентам одновременно просматривать и, возможно, модифицировать одни и те же данные. Большую часть времени пользователи работают с разными частями этих данных, но иногда они могут оперировать одними и теми же данными. В результате возникает необходимость координации этих взаимодействий, чтобы избежать ситуаций, подобных двум заказам на один и тот же гостиничный номер.

Как известно, параллельность трудно обеспечить в полной мере, предотвратив все возможные ошибки, которые встречаются даже у самых внимательных программистов. Поскольку в промышленных приложениях может быть много пользователей и систем, работающих одновременно, возникает большой простор для неприятностей. Реляционные базы данных помогают предотвратить эти неприятности, управляя всем доступом к данным посредством транзакций. Хотя это и не панацея (вы все еще обязаны исправлять ошибки транзакций, когда пытаетесь зарезервировать только что занятый гостиничный номер), механизм транзакций хорошо справляется со сложностью параллельной работы.

Транзакции также играют важную роль в обработке ошибок. Используя транзакции, можно внести изменение, и если возникнет ошибка при обработке этого изменения, выполнить откат, исправив ситуацию.

1.1.3. Интеграция

Промышленные приложения функционируют в сложном окружении, где требуется обеспечить взаимодействие многих приложений, написанных разными коллективами. Обеспечить такое взаимодействие приложений довольно сложно, потому что для этого необходимо раздвинуть организационные границы, установленные людьми. Разные приложения часто используют одни и те же данные, и изменения, внесенные одним приложением, должны быть известны другим.

Традиционно для этого используется *интеграция баз данных коллективного пользования* (shared database integration) [Hohpe, Woolf], при которой несколько приложений хранят свои данные в одной базе. Использование одной базы данных позволяет всем приложениям легко использовать данные другого приложения, а механизм управления параллельной работой базы данных обрабатывает запросы от нескольких приложений так, будто он обрабатывает запросы нескольких пользователей одного и того же приложения.

1.1.4. (Почти) стандартная модель

Реляционные базы данных получили признание благодаря тому, что они предоставляют важные преимущества, указанные выше, (почти) стандартным способом. В результате разработчики баз данных могут изучить основную реляционную модель и применять ее во многих проектах.

Несмотря на различия между разными реляционными базами данных, их основной механизм остается одним и тем же: разные диалекты языка SQL похожи друг на друга и транзакции обрабатываются практически одинаково.

1.2. Потеря соответствия

Реляционные базы данных имеют много преимуществ, но они не идеальны. Уже с первых лет они приносили много проблем.

Для разработчиков приложений основной проблемой была *потеря соответствия* (impedance mismatch): различие между реляционной моделью и структурами данных, находящимися в памяти. Реляционная модель данных представляет их в виде таблиц и строк, или точнее, отношений и кортежей. В реляционной модели кортежем называется множество пар имя–значение, а отношением — множество кортежей. (Реляционное определение кортежа немного отличается от их определения в математике и многих языках программирования, имеющих соответствующие типы данных, в которых кортежи представляют собой последовательности значений.) Все операции в языке SQL получают и возвращают отношения, что приводит к математически элегантной реляционной алгебре.

Использование отношений обеспечивает определенную элегантность и простоту, но одновременно накладывает некоторые ограничения. В частности, значения в реляционном кортеже должны быть простыми — они не должны содержать никаких структур, например вложенных записей или списков. Это ограничение не относится к структурам данных, находящихся в памяти, которые могут быть намного сложнее отношений. В результате, если вы захотите использовать сложную структуру данных, находящихся в памяти, то должны будете преобразовать реляционное представление для хранения на диске. В итоге возникает потеря соответствия — два разных представления, требующих трансляции (рис. 1.1).

Потеря соответствия — основной источник неприятностей для разработчиков приложений. В 1990-х годах многие считали, что это в конце концов приведет к замене реляционных баз данных базами, реплицирующими структуры данных, хранящихся в памяти, на диск. Это десятилетие было отмечено ростом объектно-ориентированных языков программирования, а за ними появились объектно-ориентированные базы данных. Ожидалось, что они будут доминировать в области разработки программного обеспечения в новом тысячелетии.

Однако в то время как объектно-ориентированные языки заняли лидирующие позиции в программировании, объектно-ориентированные базы данных исчезли во мраке. Реляционные базы данных ответили на вызов, усилив свою роль в качестве интегрирующего механизма, поддержанного практически стандартным языком для

манипулирования данными (SQL) и возрастающей профессиональной дифференциацией между разработчиками приложений и администраторами баз данных.

Появление множества библиотек для объектно-реляционного отображения, таких как Hibernate и iBATIS, реализующих широко известные шаблоны отображения [Fowler PoEAA], смягчило, но не устранило проблему потери соответствия. Библиотеки для объектно-реляционного отображения взяли на себя бо́льшую часть "грязной" работы, но сами по себе могут стать проблемой, если разработчики будут упрямо игнорировать потерю эффективности работы баз данных и обработки запросов.

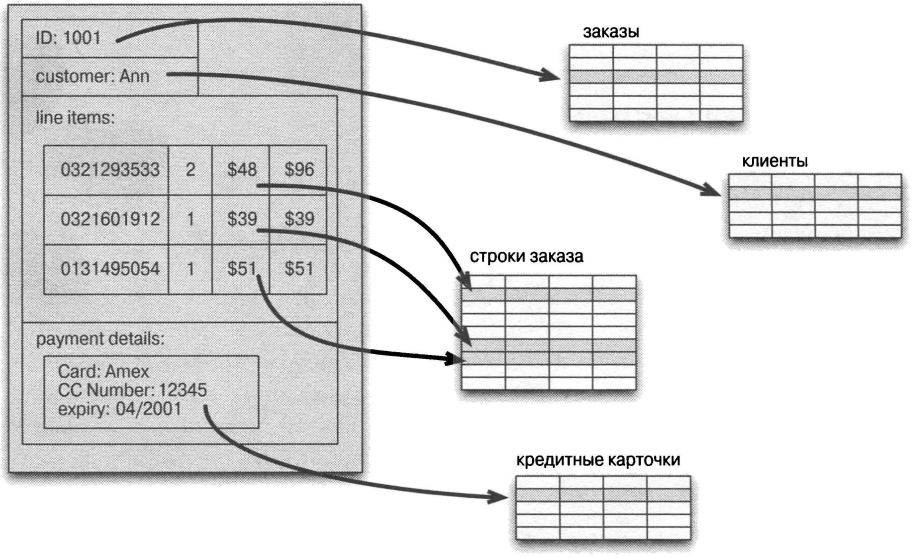


Рис. 1.1. Заказ, который в пользовательском интерфейсе выглядит как единая агрегированная структура, в реляционной базе данных разделяется на множество строк из многих таблиц

Реляционные базы данных продолжают доминировать в области промышленной обработки данных в 2000-х годах, но на протяжении первого десятилетия в стенах их неприступной крепости стали появляться трещины.

1.3. Интеграционные базы данных и базы данных приложения

Точные причины, по которым реляционные базы данных триумфально победили объектно-ориентированные базы данных, остаются предметом публичных споров между разработчиками определенного возраста. С нашей точки зрения, основным фактором была роль языка SQL в качестве интегрирующего механизма между приложениями. В этом сценарии база данных действует как *интеграционная база данных* (integration database) — с многочисленными приложениями, которые, как правило,

разработаны разными коллективами и хранят свои данные в общей базе данных. Это улучшает взаимодействие, потому что все приложения оперируют согласованным множеством сохраняемых данных.

Интеграция общих баз данных имеет свои недостатки. Структура, разработанная для интеграции многих приложений, рано или поздно становится сложнее — на самом деле, часто она оказывается намного сложнее, — чем требуется для отдельного приложения. Кроме того, если приложение захочет изменить хранилище своих данных, оно должно будет координировать это со всеми приложениями, использующими базу данных. Разные приложения имеют разные требования к структуре и производительности, поэтому индекс, необходимый для одного приложения, может стать проблемой для другого. То, что приложения обычно разрабатываются разными коллективами, означает, что база данных, как правило, не может доверить приложениям изменять свои данные с сохранением целостности базы данных и должна взять это на себя.

В качестве альтернативы можно применить *базу данных приложения* (application database), которая открывает доступ к данным только для одного приложения, разработанного одним коллективом. При работе с базой данных приложения только коллектив, использующий приложение, знает структуру базы данных. Это облегчает поддержку и развитие ее схемы. Поскольку команда, использующая приложение, управляет кодами как базы данных, так и приложения, ответственность за целостность базы данных можно возложить на код приложения.

Теперь ответственность за обеспечение взаимодействия перекладывается на интерфейсы приложения. Это позволяет использовать более хорошие протоколы взаимодействия и создать возможности для их изменения. На протяжении 2000-х годов мы видели явный сдвиг в сторону веб-сервисов [Daigneau], в которых приложения взаимодействуют посредством протокола HTTP. Веб-сервисы создают новую форму широко используемого механизма взаимодействия — альтернативу использованию языка SQL с общими базами данных. (Большая часть этой работы прошла под лозунгом "сервисно-ориентированной архитектуры" — термин, замечательный по своей бессмысленности.)

Интересный аспект сдвига в сторону веб-сервисов, используемых в качестве интеграционного механизма, заключался в том, что они повысили гибкость обмениваемых структур данных. Если вы обеспечиваете взаимодействие с помощью языка SQL, то данные должны быть структурированы как отношения. Однако при работе с сервисами вы можете использовать более содержательные структуры данных с вложенными записями и списками. Обычно они представляются в виде документов на языке XML, а с недавних пор — в формате JSON. Как правило, при удаленном общении стремятся уменьшить количество сеансов в ходе взаимодействия, поэтому полезно иметь возможность поместить побольше информации в один запрос или ответ.

Если вы собираетесь использовать сервисы для интеграции (чаще всего для этого применяют веб-сервисы с помощью текстового протокола HTTP), желаем успеха. Однако, если вы имеете дело с взаимодействиями, для которых важной является производительность, вам нужен двоичный протокол. Его следует использовать только в случае необходимости, поскольку с текстовыми протоколами работать легче — в качестве примера можно назвать Интернет.

Приняв решение использовать базу данных приложения, вы получаете свободу выбора базы данных. Поскольку между внутренней базой данных и сервисами, посредством которых вы общаетесь с внешним миром, есть несогласованность, внешнему миру совершенно безразлично, как вы храните свои данные. Это дает вам возможность использовать нереляционные варианты. Более того, существуют многочисленные особенности реляционных баз данных, такие как безопасность, которые для базы данных приложения особого значения не имеют, поскольку их можно обеспечить за счет внутреннего приложения.

Тем не менее, несмотря на эту свободу, нельзя сказать, что базы данных приложения оказали значительное влияние на использование альтернативных хранилищ данных. Большинство коллективов, занимавшихся разработкой баз данных приложения, придерживались реляционных баз данных. Помимо всего прочего, использование базы данных приложения приносит много преимуществ даже помимо гибкости данных (которая была основной причиной, по которой мы рекомендовали такие базы данных). Реляционные базы данных хорошо известны и обычно хорошо работают, точнее, достаточно хорошо. Возможно, со временем базы данных приложения действительно нарушат гегемонию реляционных баз данных, но причиной этого сдвига будет другой источник.

1.4. Атака кластеров

В начале нового тысячелетия в технологическом мире лопнул пузырь доткомов (dot-com bubble), надувавшийся в 1990-х годах. Это вызвало у многих людей вопросы об экономических перспективах Интернета, но в 2000-х годах некоторые из главных свойств сети веб приобрели очень большой масштаб.

Это увеличение масштаба происходило во многих направлениях. Веб-сайты стали внимательно следить за деятельностью клиентов и структурой. Появились крупные совокупности данных: связи, социальные сети, активность в блогах, картографические данные. Рост объема данных сопровождался ростом количества пользователей — крупнейшие сайты достигли громадных размеров и обслуживали огромное количество клиентов.

Для того чтобы справиться с возрастающим объемом данных и трафика, потребовались более крупные вычислительные ресурсы. Существовали две возможности масштабирования: вертикальное и горизонтальное. *Вертикальное масштабирование* (scaling up) подразумевает укрупнение компьютеров, увеличение количества процессоров, а также дисковой и оперативной памяти. Но более крупные машины становятся все более и более дорогими, не говоря уже о том, что существует физический предел их укрупнения. Альтернативой было использование большого количества небольших машин, объединенных в кластер. Кластер маленьких машин может использовать обычное аппаратное обеспечение и в результате удешевить масштабирование. Кроме того, кластер более надежен — отдельные машины могут выйти из строя, но весь кластер продолжит функционирование, несмотря на эти сбои, обеспечивая высокую надежность.

Когда произошел сдвиг в сторону кластеров, возникла новая проблема — реляционные базы данных не предназначены для работы на кластерах. Кластерные реляционные

базы данных, такие как Oracle RAC или Microsoft SQL Server, основаны на концепции общей дисковой подсистемы. Они используют кластерную файловую систему, выполняющую запись данных в легко доступную дисковую подсистему, но это значит, что дисковая подсистема по-прежнему является единственным источником уязвимости кластера. Реляционные базы данных также могут работать на разных серверах с разными наборами данных, эффективно выполняя фрагментацию базы данных (sharding) (см. раздел 4.2, "Фрагментация"). Хотя это позволяет разделить рабочую нагрузку, вся процедура фрагментации должна контролироваться приложением, которое должно следить за тем, какой сервер базы данных и за какой порцией данных обращается. Кроме того, утрачивается возможность управления запросами, целостностью данных, транзакциями и согласованностью между частями кластера. В связи с этим мы часто слышали от людей, выполнявших эти процедуры, что это было "неестественно".

Технические проблемы усугублялись стоимостью лицензий. Стоимость коммерческих реляционных баз обычно устанавливается при условии, что эти базы будут работать на одном сервере, поэтому переход на кластеры поднял цены и привел к бесполезным переговорам с отделами материально-технического снабжения.

Это несоответствие между реляционными базами данных и кластерами вынудило некоторые организации рассмотреть альтернативные способы хранения данных. В частности, большое влияние оказали две компании — Google и Amazon. Они вышли на передний фронт работы с горизонтально масштабированными кластерами; более того, они хранили огромные объемы данных. Это стимулировало остальные компании. Обе компании были успешными и быстро растущими высокотехнологичными предприятиями, обладающими средствами и возможностями. Им не приходило в голову, что они уничтожают свои реляционные базы данных. Когда прошло первое десятилетие нового века, обе компании опубликовали короткие, но очень важные документы о своих усилиях: BigTable от компании Google и Dynamo от компании Amazon.

Часто говорят, что масштаб работы компаний Amazon и Google намного превышает масштабы деятельности большинства компаний, поэтому решения, в которых они нуждались, могут оказаться неприемлемыми для организаций среднего размера. Несмотря на то что большинство проектов программного обеспечения не требуют такого масштабного применения, все больше и больше организаций начинают изучать возможности хранения и обработки более крупных объемов данных — и наталкиваются на те же проблемы. Таким образом, чем больше информации об опыте компаний Google и Amazon мы получаем, тем больше людей начинают предпринимать попытки создавать аналогичные базы данных, непосредственно предназначенные для кластеров. В то время как прежние угрозы доминирования реляционных баз превратились в фантомы, угроза со стороны кластеров оказалась серьезной.

1.5. Появление баз данных NoSQL

Ирония судьбы заключается в том, что термин "NoSQL" впервые появился в конце 1990-х годов в качестве названия реляционной базы данных с открытым исходным кодом [Strozzi NoSQL]. Согласно Карло Строцци (Carlo Strozzi), эта база данных хранит

свои таблицы в виде файлов в кодировке ASCII, каждый кортеж представляется строкой, состоящей из полей, разделенных знаками табуляции. Название объясняется тем фактом, что эта база данных не использует язык SQL в качестве языка запросов. Вместо этого эта база данных манипулирует сценариями оболочки, которые можно объединять в обычные конвейеры UNIX. Помимо совпадения имен, база данных NoSQL, разработанная Строцци, не имеет никакого отношения к базам данных, описанным в нашей книге.

Рождение термина "NoSQL" в современном смысле датируется 11 июня 2009 года. Он появился на конференции в Сан-Франциско, организованной Йоханом Оскарссоном (Johan Oskarsson), разработчиком программного обеспечения, живущим в Лондоне. Пример баз данных BigTable и Dynamo вдохновил создателей многих проектов на эксперименты с альтернативными хранилищами данных, и этой теме в то время было посвящено сразу несколько конференций. Йохан хотел побольше узнать об этих новых базах данных, прибыв на конференцию Hadoop в Сан-Франциско. Поскольку у него было мало времени, чтобы посетить все конференции, он решил организовать собственный семинар, на котором можно было собрать экспертов и представить их работы.

Йохану нужно было имя для своего семинара — что-нибудь вроде хеш-тега в Твиттере: короткое, запоминающееся и не вызывающее в поисковой машине Google слишком много ссылок. Он обратился за помощью в ретранслируемый интернет-чат #cassandra и получил несколько вариантов, из которых выбрал "NoSQL", предложенный Эриком Эвансом (Eric Evans), разработчиком из компании Rackspace, который не имеет никакого отношения к известному специалисту по предметно-ориентированному проектированию Эрику Эвансу. Несмотря на негативный оттенок и отсутствие прямой связи с описываемыми системами, это имя удовлетворило критерии хеш-тега. В то время они просто хотели как-то назвать свой семинар и не ожидали, что это имя станет названием целого технологического направления [Oskarsson].

Термин "NoSQL" прижился, но никогда не имел строгого определения. Исходное название семинара относилось к "распределенным нереляционным базам данных с открытым исходным кодом" [NoSQL Meetup]. Эти эпитеты относятся к базам данных Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB и MongoDB [NoSQL Debrief], но никогда не ограничивались этой семеркой. В настоящее время не существует ни общепринятого определения этого термина, ни авторитетного органа, который бы предложил такое определение, поэтому мы можем лишь обсуждать некоторые общие свойства баз данных, относящихся к категории NoSQL.

Для начала отметим очевидный факт: базы данных NoSQL не используют язык SQL. Некоторые из них имеют свой язык запросов, и их целесообразно было бы сделать похожим на SQL, чтобы их легче было изучить. К таким языкам относится CQL в базе данных Cassandra — "очень похожий на SQL (за исключением отличий)" [CQL]. Однако до сих пор не был реализован ни один язык, который бы достиг хотя бы той же степени гибкости, что и стандартный язык SQL. Интересно, что произойдет, если разработчики уже существующих баз NoSQL решат реализовать стандартную версию SQL; можно предсказать только одно последствие этого решения — множество объяснений.

Другое важное свойство этих баз данных заключается в том, что они представляют собой проекты с открытым исходным кодом. Несмотря на то что термин "NoSQL" часто применяется к системам с закрытым исходным кодом, существует мнение, что NoSQL — это феномен с открытым исходным кодом.

Большинство баз данных NoSQL создавались в ответ на необходимость работать на кластерах. Это особенно справедливо по отношению к разработчикам, принявшим участие в первом семинаре. Этот семинар повлиял на их модель данных и подход к обеспечению согласованности данных. Для обеспечения согласованности данных во всей базе данных реляционные базы данных используют транзакции ACID. Это изначально противоречит кластерной среде, поэтому базы данных предлагают спектр вариантов для обеспечения согласованности и распределения данных.

Однако не все базы данных NoSQL строго ориентируются на работу с кластерами. Графовые базы данных представляют собой базы данных NoSQL, использующие распределенную модель, похожую на реляционную базу данных, но предлагающие другую модель данных, которая лучше обрабатывает данные со сложными отношениями.

Базы данных NoSQL учитывают емкость веб-сайтов начала XXI века, поэтому обычно только системы, разработанные примерно в это время, называются NoSQL, тем самым исключая базы данных, созданные в прошлом веке, кроме базы BC (Before Codd).

Базы данных NoSQL работают без схемы, позволяя свободно добавлять поля в базу данных без предварительного изменения структуры. Это очень важно для баз данных с неоднородными данными и пользовательскими полями, для работы с которыми реляционные базы данных используют такие имена, как `customField6`, или таблицы, состоящие из пользовательских полей, которые неудобно обрабатывать и сложно понимать.

Все свойства, указанные выше, являются общими для баз данных NoSQL. Ни одно из них нельзя считать определяющим, так что скорее всего мы никогда не узнаем связное определение термина "NoSQL" (к сожалению). Однако этот набор характеристик стал нашим путеводителем при написании книги. Наш энтузиазм подогревался тем, что базы данных NoSQL открыли массу возможностей для хранения данных. Следовательно, эти возможности не должны ограничиваться тем, что обычно называется хранилищем NoSQL. Мы надеемся, что со временем другие возможности хранения данных станут более удобными, включая те из них, которые появились до баз данных NoSQL. Однако существует предел, до которого мы можем извлекать пользу из обсуждения нашей книги, поэтому мы решили отметить отсутствие формального определения.

Когда вы впервые слышите термин "NoSQL," сразу же возникает вопрос: "Это отрицание SQL?" Большинство людей, рассуждающих о технологии NoSQL, говорят, что на самом деле эта аббревиатура означает "Not Only SQL" ("Не только SQL"), но такая интерпретация порождает несколько проблем. Большинство людей пишут "NoSQL", в то время как фразе "Not Only SQL" соответствует аббревиатура "NOSQL". Кроме того, нет особого смысла называть базу данных NoSQL словом "не только", потому что, например, базы данных Oracle и Postgres тоже соответствуют такому определению, а это значит называть черное белым.

Для устранения этого противоречия мы предлагаем не интересоваться расшифровкой термина, а сосредоточиться на его смысле (это относится к большинству

аббревиатур). Таким образом, если база данных относится к категории NoSQL, значит, речь идет о нечетко определенном множестве баз данных с открытым кодом, в большинстве своем разработанных в начале XXI века и, как правило, не использующих язык запросов SQL.

Впрочем, интерпретация выражения "не только" имеет определенное значение, поскольку она описывает окружение, в котором многие люди видят свои будущие базы данных. На самом деле это именно то, что мы считаем самым важным в таком способе рассуждений: NoSQL — это скорее движение, а не технология. Мы не считаем, что реляционные базы данных исчезнут, поскольку они остаются самыми распространенными. Несмотря на то что мы написали книгу, посвященную базам данных NoSQL, мы по-прежнему рекомендуем использовать реляционные базы данных. Их понятность, надежность, функциональность и удобство поддержки будут неотразимыми аргументами при обсуждении большинства проектов.

Изменения заключаются в том, что теперь реляционные базы данных являются только одной из возможностей для хранения данных. Эта точка зрения часто называется *многовариантной персистентностью* (Polyglot Persistence), подразумевающей использование разных хранилищ данных в разных ситуациях. Вместо слепого выбора реляционных баз данных, потому что так принято, мы должны понимать природу данных, которые собираемся хранить, и знать, что мы хотим с ними делать. В результате большинство организаций в разных ситуациях будут использовать смешанные технологии хранения данных.

Для того чтобы заставить работать все это разнообразие, организации должны перейти от интеграционных баз данных к базам данных приложения. Мы предполагаем, что вы будете использовать базы данных NoSQL как базу данных приложения; как правило, мы не рассматриваем базы данных NoSQL как удачный выбор для интеграционной базы данных. Мы не считаем это недостатком, поскольку, даже если вы не используете технологию NoSQL, переход к инкапсуляции данных в сервисах — правильное направление.

В нашем очерке истории технологии NoSQL мы сосредоточились на обработке больших объемов данных на кластерах. Несмотря на то что это обстоятельство открывает новую эру в мире баз данных, это не единственная причина, по которой следует использовать базы данных NoSQL. Не менее важна старая проблема, связанная с потерей соответствия. Необходимость обработки больших объемов данных создала для людей возможность по-новому взглянуть на свои потребности, связанные с хранением данных, а некоторые коллективы разработчиков могут увидеть, как базы данных NoSQL помогают повысить производительность работы и упростить доступ к базам данных, если им необходимо выйти за пределы одной машины.

Итак, теперь, когда вы готовы прочитать остальную часть книги, запомните две основные причины для изучения технологии NoSQL. Первая причина — это необходимость обеспечить доступ к данным, объем которых и требования к производительности вынуждают использовать кластеры; вторая причина — повысить производительность разработки приложений с помощью более удобного способа обеспечения обмена данными.

1.6. Резюме

- Реляционные базы данных более двадцати лет были успешной технологией, обеспечивая персистентность, управление параллельностью и механизм интеграции.
- Разработчики приложений столкнулись с проблемой потери соответствия между реляционными моделями и структурами данных, хранящимися в памяти.
- Существует тенденция перехода от использования баз данных как средства интеграции к инкапсуляции баз данных в приложениях и интеграции посредством сервисов.
- Главным фактором, обусловившим изменение в способах хранения данных, стала необходимость обрабатывать большие объемы данных на кластерах. Реляционные базы данных не предназначены для эффективной работы на кластерах.
- Аббревиатура NoSQL — это случайный неологизм. У него нет четкого определения, и мы можем лишь указать его основные характеристики.
- Основные характеристии баз данных NoSQL
- Отказ от использования реляционной модели.
- Эффективная работа на кластерах.
- Открытый исходный код.
- Учет возможностей сети веб в XXI веке.
- Отсутствие структуры.
- Главным последствием появления технологии NoSQL является многовариантная персистентность (Polyglot Persistence).

Глава 2

Агрегированные модели данных

Модель данных — это модель, с помощью которой мы воспринимаем и обрабатываем свои данные. Для людей, работающих с базами данных, модель данных описывает способ взаимодействия с данными, находящимися в базе. Этим она отличается от модели хранилища, описывающей, как база данных хранит данные и манипулирует ими. В теории мы должны были бы игнорировать модель хранилища, но на практике нам необходимо иметь хотя бы приблизительное представление о ней, в основном для того, чтобы обеспечить приемлемую производительность.

В разговорной речи моделью данных часто называют модель конкретных данных в приложении. Разработчик может указать на диаграмму "сущность–связь" своей базы данных и назвать ее моделью данных, содержащей клиентов, заказы, товары и т.д. Однако в нашей книге термин "модель данных" относится к модели, в соответствии с которой база данных организует данные, — то, что формально можно было бы назвать метамоделью.

В последние несколько десятилетий доминирующей была реляционная модель данных, которая лучше всего представляется в виде набора таблиц, напоминающих страницы электронных таблиц. Каждая таблица состоит из строк, представляющих какую-то сущность. Мы описываем эту сущность с помощью столбцов, каждый из которых имеет отдельное значение. Столбец может относиться к другой строке той же или другой таблицы. Таким образом, возникают связи между сущностями. (Говоря о таблицах и строках, мы используем неформальную, но распространенную терминологию; более формальными терминами являются "отношения" и "кортежи".)

Одной из основных особенностей технологии NoSQL является отказ от реляционной модели. Каждое решение в рамках технологии NoSQL использует свою собственную модель. Эти модели разделяются на четыре категории: ключ–значение, документ, семейство столбцов и граф. Первые три модели имеют общее свойство, которое мы назовем *агрегатной ориентацией* (aggregate orientation). В этой главе объясняется, что мы понимаем под агрегатной ориентацией и что это значит для моделей данных.

2.1. Агрегаты

Реляционная модель получает информацию, которую мы хотим хранить, и разделяет ее на кортежи (строки). Кортеж — это ограниченная структура данных. Он хранит набор значений, поэтому не может содержать запись, список значений или другой кортеж. Эта простота образует основу реляционной модели и позволяет интерпретировать все операции как операции над кортежами и возвращение кортежей.

Агрегатная ориентация придерживается другого подхода. Она учитывает необходимость оперировать данными, имеющими более сложную структуру, чем набор кортежей. Ее можно описать в терминах сложной записи, которая может содержать списки и другие структуры записей. Как мы увидим, базы данных типа "ключ–значение", документ и семейство столбцов могут содержать сложные записи. Однако для этой сложной записи нет общепринятого термина; в книге мы называем ее *агрегатом* (aggregate).

Агрегат — это термин, пришедший из предметно-ориентированного проектирования [Evans]. В предметно-ориентированном проектировании *агрегатом* называют коллекцию связанных объектов, которая интерпретируется как единое целое. В частности, она представляет собой единицу для манипулирования данными и управления их согласованностью. Обычно мы модифицируем агрегаты с помощью атомарных операций и взаимодействуем с хранилищем данных посредством агрегатов. Это определение довольно точно описывает принципы работы баз данных типа "ключ–значение", документ и семейство значений. Агрегаты облегчают работу баз данных на кластерах, поскольку агрегат представляет собой естественную единицу репликации и фрагментации. Кроме того, агрегаты упрощают разработку прикладных программ, которые часто манипулируют данными с помощью агрегированных структур.

2.1.1. Пример отношений и агрегатов

Попробуем продемонстрировать сказанное на примере. Предположим, мы разрабатываем веб-сайт для электронной торговли; мы собираемся продавать товары непосредственно клиентами через веб и хотим хранить информацию о пользователях, каталогах наших товаров, заказы, адреса поставки и даты платежей. Этот сценарий можно использовать для моделирования данных с помощью реляционной модели, а также технологии NoSQL, а потом проанализировать их преимущества и недостатки. Разработку реляционной базы даных можно начать с модели данных, представленной на рис. 2.1.

Мы выполнили все правила реляционной модели, все нормализовано, так что никакие данные не встречаются в нескольких базах одновременно. Кроме того, мы обеспечили целостность ссылок. Реалистичная система заказов, конечно, выглядит сложнее, но в книге можно ограничиться ее упрощенным вариантом.

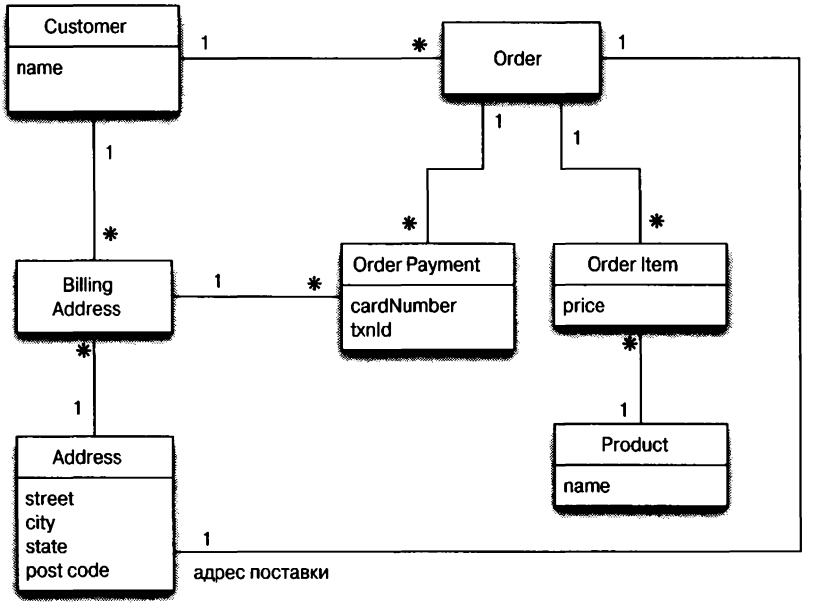


Рис. 2.1. Модель данных, ориентированная на реляционную базу данных (в обозначениях UML [Fowler UNL])

На рис. 2.2 представлены некие простые данные для этой модели.

Customer	
Id	Name
1	Martin

Orders		
Id	CustomerId	ShippingAddressId
99	1	77

Product	
Id	Name
27	NoSQL Distilled

BillingAddress		
Id	CustomerId	AddressId
55	1	77

OrderItem			
Id	OrderId	ProductId	Price
100	99	27	32.45

Address	
Id	City
77	Chicago

OrderPayment				
Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Рис. 2.2. Типичные данные для использования в модели RDBMS

Посмотрим, как будет выглядеть эта модель, если мы используем агрегатно-ориентированный подход (рис. 2.3).

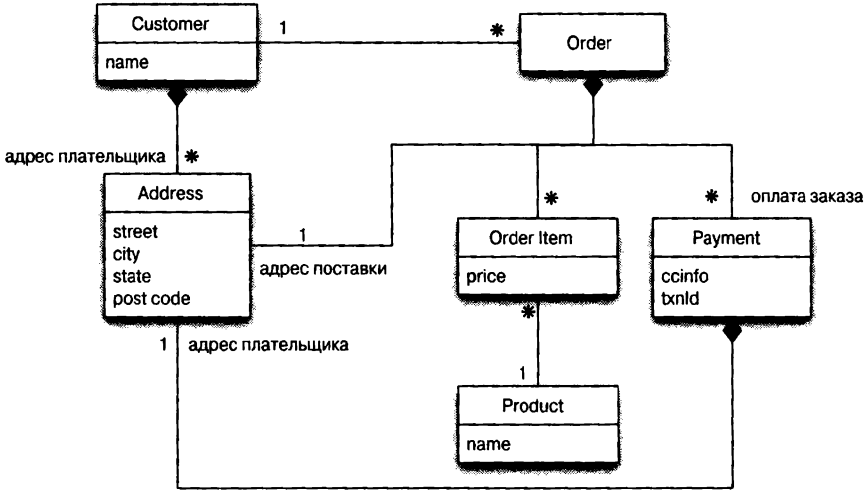


Рис. 2.3. Агрегатная модель данных

Итак, у нас есть простые данные, которые мы представим в формате JSON, который является основным способом представления данных в технологии NoSQL.

```
// Клиенты
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}

// Заказы
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
  }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
  "ccinfo":"1000-1000-1000-1000",
  "txnId":"abelif879rft",
  "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

В этой модели есть два основных агрегата: клиент и заказ. В языке UML черный ромб обозначает агрегацию. Агрегат клиента содержит список адресов плательщиков; агрегат заказа содержит список заказанных товаров, адреса поставки и данные о платежах. Запись о платеже сама содержит адрес клиента, выполняющего данный платеж.

Отдельная логическая запись, содержащая адрес, в этом примере появляется три раза, но вместо использования идентификатора она интерпретируется как значение и каждый раз копируется. Это соответствует ситуации, в которой вы не хотите изменять адреса доставки и плательщика. В реляционных базах данных мы должны были бы гарантировать, что строки адресов не будут изменяться, создавая вместо них новые строки. При использовании агрегатов мы можем копировать всю адресную структуру в агрегат.

Связь между клиентом и заказом не хранится в агрегатах — это связь между агрегатами. Аналогично связь, идущая от заказа, может идти к отдельной агрегированной структуре для товаров, но она не хранится в этой структуре. Мы показали название товара в качестве части заказа, — этот вид денормализации напоминает компромисс, принятый в реляционных базах данных, но по отношению к агрегатам он носит более общий характер, потому что мы хотим минимизировать количество агрегатов, к которым будем иметь доступ при работе с данными.

В данном примере важен не столько конкретный способ изображения границы агрегата, сколько тот факт, что мы должны думать о доступе к данным при разработке модели данных для приложения. Действительно, мы могли бы иначе изобразить границы агрегатов, поместив все заказы отдельного клиента в агрегат клиента (рис. 2.4).

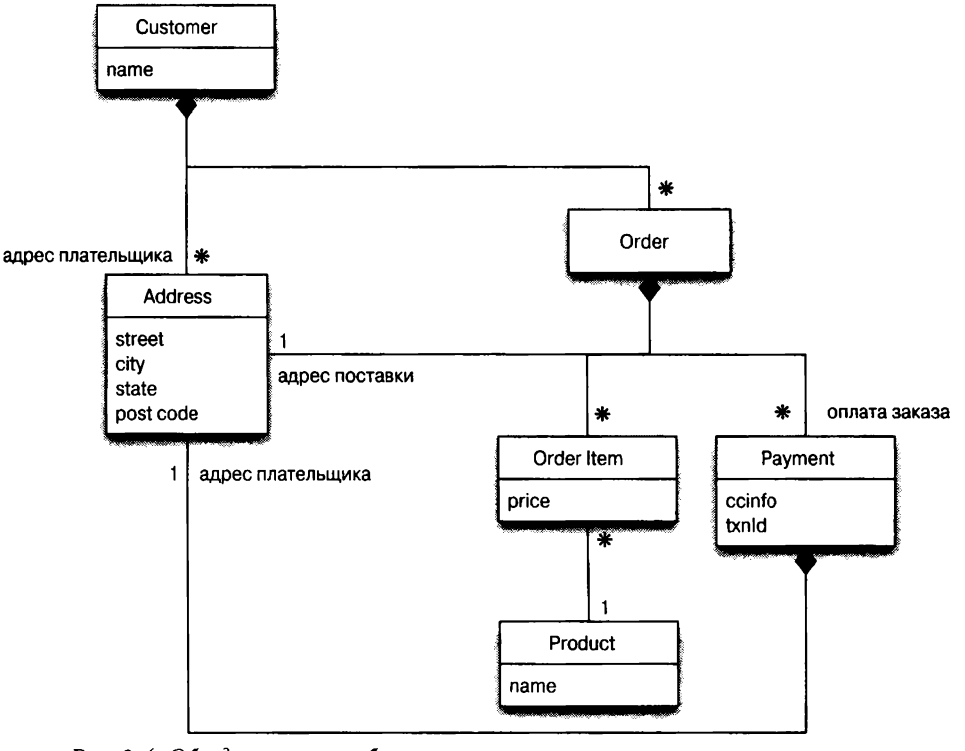


Рис. 2.4. Объединение всех объектов, соответствующих клиенту и его заказам

Используя описанную выше модель данных, записи `Customer` и `Order` можно переписать следующим образом:

```
// Клиенты
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
      "productId":27,
      "price": 32.45,
      "productName": "NoSQL Distilled"
    }
 ],
 "shippingAddress":[{"city":"Chicago"}]
   "orderPayment":[
     {
       "ccinfo":"1000-1000-1000-1000",
       "txnId":"abelif879rft",
       "billingAddress": {"city": "Chicago"}
     }],
   }]
  }
}
```

Как это часто бывает в моделировании, универсального способа для изображения границ агрегатов не существует. Это целиком зависит от ваших намерений относительно манипулирования данными. Если вы хотите получать доступ к записи о клиенте и ко всем его заказам одновременно, то, вероятно, предпочтете один агрегат. Однако, если вы хотите в каждый момент времени получать доступ к отдельному заказу, то должны предусмотреть отдельный агрегат для каждого заказа. Естественно, это очень сильно зависит от контекста; даже в рамках одной системы разные приложения могут иметь разные предпочтения. Этим, в частности, объясняется, почему так много людей предпочитают игнорировать агрегаты.

2.1.2. Последствия ориентации на агрегаты

Несмотря на то что реляционное отображение достаточно хорошо отражает элементы данных и отношения между ними, оно никак не учитывает понятие агрегата. При описании предметной области мы можем говорить, что заказ состоит из предметов заказа, адреса поставки и платежа. В реляционной модели это можно выразить в терминах отношений с внешним ключом. Однако отношения, представляющие

агрегацию, невозможно отличить от отношений, не представляющих агрегацию. В результате база данных не может использовать знание об агрегатной структуре при хранении и распределении данных.

В разных методах моделирования данных существуют возможности разметить агрегаты или составные структуры. Однако проблема заключается в том, что эти методы редко предоставляют какие-либо семантические конструкции, позволяющие отличить отношение агрегирования от других; а те методы, которые это делают, решают эту задачу по-разному. Работая с агрегатно-ориентированными базами данных, мы имеем более четкую семантику, позволяющую сосредоточиться на единице взаимодействия в хранилище данных. Тем не менее это не является логическим свойством данных. Все это относится к тому, как данные используются в приложениях, — т.е. к вопросу, который часто выходит за рамки моделирования данных.

Реляционные базы данных не имеют концепции агрегата в своей модели данных, поэтому мы называем их *безагрегатными* (aggregate-ignorant). В технологии NoSQL графовые базы данных также являются безагрегатными. Это свойство нельзя назвать недостатком. Часто трудно правильно изобразить границы агрегатов, особенно если одни и те же данные используются в разных контекстах. Заказ представляет собой удобный агрегат, если клиент создает и просматривает заказы, а розничный продавец обрабатывает их. Однако, если продавец захочет проанализировать свои продажи за последние несколько месяцев, агрегат заказов станет для него проблемой. Для того чтобы получить историю продаж, вы должны заглянуть в каждый агрегат в вашей базе данных. Итак, агрегатная структура может упростить одни операции с данными и усложнить другие. Безагрегатная модель позволяет легко просматривать данные разными способами, поэтому она является лучшим выбором, если у вас нет основной структуры для манипулирования данными.

Заключительный аргумент в пользу агрегатной ориентации заключается в том, что она очень облегчает работу на кластерах, которая, как вы помните, была основной причиной появления технологии NoSQL. При работе на кластерах вы должны минимизировать количество узлов, которые необходимо опросить, чтобы собрать данные. Включая агрегаты явным образом, мы предоставляем базе данных важную информацию о том, какие части данных обрабатываются вместе и, следовательно, должны храниться на одном и том же узле.

Агрегаты оказывают сильное влияние на транзакции. Реляционные базы данных позволяют манипулировать любой комбинацией строк из любой таблицы в рамках одной транзакции. Такие транзакции называются *ACID*: атомарные (atomic), согласованные (consistent), изолированные (isolated) и долговечные (durable). ACID — это надуманная аббревиатура; главным пунктом является атомарность: многие строки из разных таблиц обновляются в рамках одной операции. Эта операция завершается либо полным успехом, либо полной неудачей, причем параллельные операции изолируются друг от друга так, что они не могут выполнять частичные модификации.

Часто говорят, что базы данных NoSQL не поддерживают транзакции ACID и тем самым не обеспечивают согласованность. Это огульное упрощение. В целом это относится к тем агрегатно-ориентированным базам данных, у которых нет транзакций ACID, охватывающих несколько агрегатов. Вместо этого они поддерживают атомарные манипуляции с отдельными агрегатами по очереди. Это значит, что если

нам требуется обработать несколько агрегатов атомарным образом, то мы должны управлять ими из кода приложения.

На практике наши атомарные потребности можно удовлетворить в рамках отдельного агрегата; действительно, эта задача является частью общей проблемы, связанной с разделение данных по агрегатам. Следует также помнить, что графовые и другие безагрегатные базы данных обычно используют транзакции ACID аналогично реляционным базам данных. Кроме того, как будет показано в главе 5, согласованность данных представляет собой отдельную проблему, для решения которой не важно, поддерживает ли база данных транзакции ACID или нет.

2.2. Модель данных "ключ–значение" и документная модель

Ранее мы говорили о том, что базы данных типа "ключ–значение" и документные базы данных являются сильно агрегатно-ориентированными. Мы имели в виду, что эти базы данных в основном были сконструированы из агрегатов. Базы данных обоих типов состоят из множества агрегатов, каждый из которых имеет ключ или идентификатор, который используется для доступа к данным.

Эти две модели отличаются друг от друга тем, что в базе данных "ключ–значение" агрегат является непроницаемым для базы данных — просто большой черный ящик, состоящий из преимущественно бессмысленных битов. В противоположность этому документная база может видеть структуру агрегата. Преимущество непрозрачности заключается в том, что в агрегате можно хранить все что угодно. База данных может ограничивать общий размер агрегата, но в остальном мы имеем полную свободу. Документная база данных накладывает ограничения на то, что можно хранить в агрегате, определяя допустимые структуры и типы. Однако за это мы получаем бо́льшую гибкость доступа. В хранилище типа "ключ–значение" мы можем просматривать агрегат только с помощью его ключа. В документной базе данных мы можем посылать базе данных запросы, касающиеся полей в агрегате, извлекать части агрегата, а не весь агрегат целиком, причем база данных может создавать индексы с учетом содержимого агрегата. На практике разделительная линия между базами данных типа "ключ–значение" и документными базами данных довольна расплывчата. Люди часто записывают идентификаторы в документные базы данных, чтобы выполнять поиск в стиле "ключ–значение". Базы данных, классифицированные как базы типа "ключ–значение", могут предлагать новые структуры для данных, помимо непрозрачных агрегатов. Например, база данных Riak позволяет добавлять метаданные к агрегатам для индексирования и установления связей между агрегатами, а Redis позволяет разбивать агрегаты на списки или множества. Кроме того, можно обеспечить механизм запросов с помощью интегрированных средств поиска, как в базе данных Solr. Например, поисковый механизм базы данных Riak, аналогичный поисковому механизму базы Solr, выполняет поиск агрегатов, хранящихся в виде структур JSON или XML.

Несмотря на такое нечеткое разделение, эти две категории в целом отличаются друг от друга. Базы данных типа "ключ–значение", как правило, выполняют поиск агрегатов по ключу. В документных базах данных пользователь должен подать запрос, основанный на внутренней структуре документа; это может быть ключ, но, скорее всего, это будет нечто другое.

2.3. Хранилища типа "семейство столбцов"

Одной из ранних и популярных баз данных NoSQL была база BigTable компании Google [Chang etc.]. Ее имя вызывает в воображении табличную структуру, состоящую из отдельных столбцов и не имеющую схемы. Как вскоре будет показано, эту структуру не следует представлять в виде таблицы; скорее она представляет собой двухуровневый ассоциативный массив. Однако, как бы вы ни представляли себе эту структуру, эта модель оказала влияние на более поздние базы данных, такие как HBase и Cassandra.

Эти базы данных с таблицами в стиле BigTable часто называют хранилищами столбцов, но это имя относится совсем к другой сущности. Хранилища столбцов, существовавшие до появления технологии NoSQL, такие как C-Store [C-Store], прекрасно уживались с языком SQL и реляционной моделью. Они отличались лишь способом физического хранения данных. Большинство баз данных в качестве единицы хранения используют строки. Помимо всего прочего, это позволяет обеспечить высокую производительность записи. Однако существует много сценариев, в которых записи выполняются редко, но приходится часто считывать по несколько столбцов из многих строк одновременно. В этой ситуации лучше считать единицей хранения группы столбцов для всех строк. Именно поэтому такие базы данных называются хранилищами столбцов.

База данных BigTable и ее потомки основаны на концепции хранения групп столбцов (семейств столбцов), но, в отличие базы C-Store и ее аналогов, в них не используются реляционная модель и язык SQL. В данной книге мы называем этот класс баз данных семействами столбцов.

Вероятно, лучше всего представлять модель семейства столбцов как двухуровневую агрегатную структуру. Как и в хранилищах типа "ключ–значение", главный ключ часто описывается как идентификатор строки, отмечая интересующий нас агрегат. Отличительной особенностью структур типа "семейство столбцов" является то, что эта строка-агрегат сама состоит из ассоциативного массива более детализированных значений. Эти значения второго уровня называются столбцами. Помимо доступа к строкам как к единому целому, операции также допускают извлечение конкретного столбца, так что, для того чтобы получить имя клиента на рис. 2.5, мы могли бы написать команду наподобие `get('1234', 'name')`.

Базы данных типа "семейство столбцов" организуют свои столбцы в семейства. Каждый столбец должен быть частью одного семейства столбцов и быть единицей доступа. При этом предполагается, что данные в конкретном семействе столбцов обычно доступны одновременно.

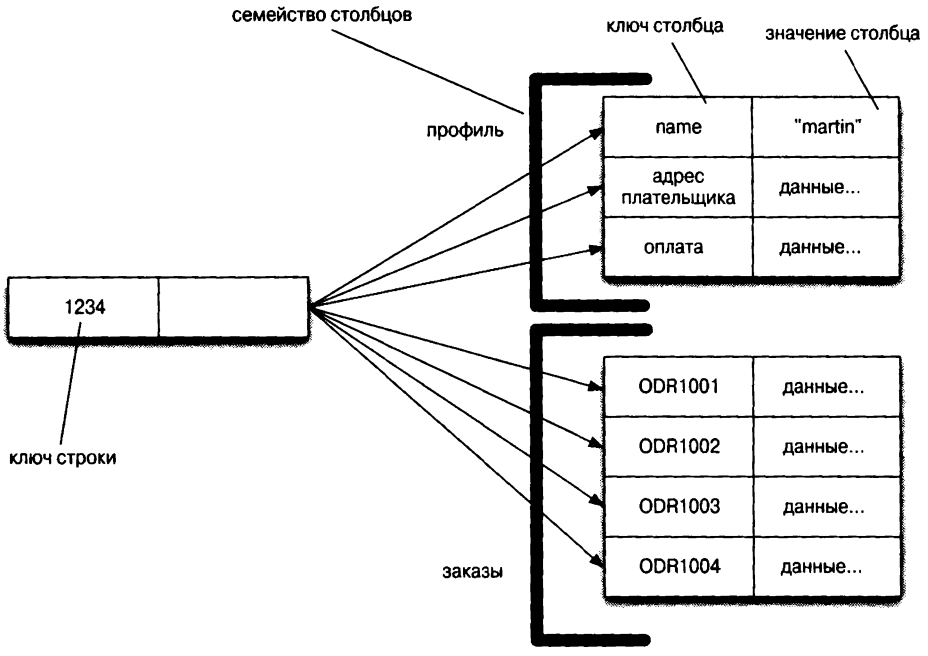


Рис. 2.5. Представление информации о пользователе в виде структуры "семейство столбцов"

Это открывает несколько возможностей для представления о том, как структурированы данные в базе.

- Ориентация по строкам. Каждая строка — это агрегат (например, клиент с идентификатором 1234), а семейства столбцов содержат фрагменты данных (профиль, история заказов) в этом агрегате.
- Ориентация по столбцам. Каждое семейство столбцов определяет тип записи (например, профили клиентов), причем каждой записи соответствуют строки. В таком случае строку можно интерпретировать как объединение записей из всех семейств столбцов.

Последний аспект отражает "столбцовую" природу баз данных типа "семейство столбцов". Поскольку базе данных известно о группировке данных, она может использовать эту информацию для хранения и обеспечения доступа. Несмотря на то что документная база данных объявляет определенную структуру данных, каждый документ по-прежнему рассматривается как отдельная единица. Базы данных типа "семейство столбцов" имеют двумерный характер.

Все сказанное относится к базам данных Google BigTable и HBase, но Cassandra немного отличается от них. Строка в базе данных Cassandra возникает только в семействе столбцов, но это семейство может содержать суперстолбцы — столбцы, содержащие вложенные столбцы. Суперстолбцы в базе Cassandra являются ближайшим аналогом классических семейств столбцов BigTable.

Представлять семейства столбцов в виде таблиц неправильно. Вы можете добавлять любой столбец в любую строку, а строки могут иметь самые разные ключи. В то время как новые столбцы добавляются в строки при обычном доступе к базе данных, определение нового семейства столбцов происходит намного реже и может вызвать остановку работы базы данных.

Пример, приведенный на рис. 2.5, иллюстрирует другой аспект баз данных типа "семейство столбцов", который может оказаться неизвестным людям, использующим реляционные базы данных: семейство столбцов `orders`. Поскольку столбцы можно добавлять свободно, список элементов можно легко моделировать, сделав каждый элемент отдельным столбцом. Если представлять себе семейство столбцов как таблицу, это может показаться очень странным, но если строку семейства столбцов интерпретировать как агрегат, все становится на свои места. В базе данных Cassandra строки бывают широкими и "худыми". *"Худые" строки* (skinny rows) содержат несколько столбцов, причем одни и те же столбцы используются в разных строках. В данном случае семейство столбцов определяет тип записи, каждая строка является записью, а каждый столбец — полем. *Широкая строка* (wide row) содержит много разных столбцов (возможно, тысячи). Широкое семейство столбцов моделирует список, в котором каждый столбец представляет собой элемент в этом списке.

Широкие семейства столбцов могут определять определенный порядок следования своих столбцов. В таком случае мы можем обращаться к заказам и диапазонам заказов по их порядковым ключам. Если заказы упорядочены по идентификаторам, это не представляет интереса, но ключ представляет собой конкатенацию даты и идентификатора (например, `20111027-1001`), и это оказывается полезным.

Несмотря на то что полезно различать широкие и "худые" семейства столбцов, нет никаких формальных причин, по которым семейство столбцов не может содержать как столбцы, похожие на поля, так и столбцы, напоминающие списки, хотя упорядочение при этом сильно усложняется.

2.4. Заключительные замечания об агрегатно-ориентированных базах данных

Мы рассмотрели достаточно много материала, чтобы сделать краткий обзор трех разных стилей агрегатно-ориентированных моделей и их отличий.

Для всех них характерно понятие агрегата, индексированного ключом, который можно использовать для поиска. Агрегат очень важен для работы на кластерах, поскольку база данных в этом случае будет гарантировать, что все данные одного агрегата хранятся на одном узле. Кроме того, агрегат является атомарной единицей модификации, обеспечивая полезный, хотя и ограниченный объем управления транзакциями.

Агрегаты бывают разными. Модели данных типа "ключ–значение" интерпретируют агрегаты как "черный ящик", т.е. искать можно только целые агрегаты, — вы не можете подать запрос на извлечение части агрегата.

В документной модели агрегат является прозрачным для базы данных. Это позволяет посылать запросы к фрагментам агрегата и осуществлять частичное извлечение

данных. Однако, поскольку документ не имеет схемы, база данных не может сильно влиять на структуру документа, чтобы оптимизировать хранение и извлечение частей агрегата.

Модели типа "семейство столбцов" разделяют агрегат на группы столбцов, интерпретируя их как единицы данных в агрегате-строке. Это накладывает на агрегат структурные ограничения, но позволяет базе данных использовать эту структуру для улучшения доступа.

2.5. Рекомендации по дальнейшему чтению

Читателям, желающим углубить свои знания об агрегатах, которые часто используются и при разработке реляционных баз данных, мы рекомендуем книгу [Evans]. Сообщество специалистов по предметно-ориентированному проектированию является лучшим источником информации об агрегатах, — свежая информация обычно появляется на веб-сайте `http://domaindrivendesign.org`.

2.6. Резюме

- Агрегат — это коллекция данных, с которой мы взаимодействуем как с отдельной единицей. Агрегаты образуют границы для операций ACID, применяемых в базе данных.
- Базы данных типа "ключ–значение", документные базы данных и семейства столбцов представляют собой агрегатно-ориентированные базы данных.
- Агрегаты упрощают управление хранением данных на кластерах.
- Агрегатно-ориентированные базы данных лучше всего работают, когда большинство операций над данными выполняются в одном и том же агрегате; безагрегатные базы данных лучше работают, когда операции выполняются над данными, которые относятся к многочисленным разным формациям.

Глава 3

Более подробно о моделях данных

В предыдущих главах описана основная особенность большинства баз данных NoSQL — использование агрегатов. Кроме того, показаны разные способы моделирования агрегатов разными агрегатно-ориентированными базами. Хотя агрегаты представляют собой ядро технологии NoSQL, существуют и другие концепции моделирования данных, к описанию которых мы переходим в этой главе.

3.1. Отношения

Агрегаты полезны тем, что они объединяют в одно целое данные, доступ к которым осуществляется одновременно. Однако существует много ситуаций, в которых доступ к связанным данным осуществляется по-другому. Рассмотрим связь между клиентом и всеми его заказами. Некоторые приложения требуют доступ к истории заказов каждый раз, когда они обращаются к данным о клиенте. В этом случае данные о клиенте и его истории заказов целесообразно объединить в один агрегат. Однако другие приложения желают обрабатывать заказы по отдельности и моделируют их как независимые агрегаты.

В этом случае агрегаты заказов и клиента целесообразно разъединить, сохранив между ними определенную связь, чтобы любая операция над заказом могла использовать данные о клиенте. Проще всего установить такую связь, включив идентификатор клиента в данные агрегата заказа. В этом случае, если вам потребуются данные из записи о клиенте, вы прочитаете заказ, узнаете идентификатор клиента и пошлете базе данных другой вызов, чтобы прочитать данные о клиенте. Это вполне работоспособное решение, которое прекрасно подходит для многих сценариев, но база данных не будет знать о связи между данными. Это может оказаться важным, поскольку иногда полезно, чтобы база данных знала о связях между данными.

В результате многие разработчики баз данных, даже хранилищ типа "ключ–значение", предпринимают усилия, чтобы сделать такие отношения видимыми. Документные хранилища открывают содержимое агрегатов для баз данных, чтобы те формировали индексы и запросы. Например, Riak, хранилище типа "ключ–значение", позволяет размещать информацию о связи в метаданных, поддерживая частичное извлечение и прослеживание связей.

Важным аспектом связей между агрегатами является их обработка модификаций. Агрегатно-ориентированные базы данных интерпретируют агрегат как единое целое при извлечении данных. Следовательно, атомарность поддерживается только в содержимом отдельного агрегата. Если вы обновляете несколько агрегатов одновременно, то должны самостоятельно реагировать на сбой при обращении к ним. Реляционные базы данных помогают пользователям тем, что позволяют модифицировать несколько записей одновременно в рамках одной транзакции, обеспечивая гарантии безопасности выполнения операций ACID при изменении нескольких строк.

Все это означает, что при обращении к нескольким агрегатам одновременно агрегатно-ориентированные базы данных становятся неудобными. Существует несколько способов смягчения этой проблемы, которые мы исследуем в следующих разделах, но фундаментальное неудобство устранить невозможно.

Следовательно, при работе с данными, между которыми установлено много связей, следует предпочесть реляционные базы данных, а не хранилище NoSQL. Несмотря на этот недостаток агрегатно-ориентированных баз данных, следует помнить, что не все реляционные базы данных великолепно справляются со сложными связями. И хотя в языке SQL вы можете формировать запросы, содержащие операции соединения, с ростом количества соединений ситуация быстро становится запутанной — как с точки зрения языка SQL, так и с точки зрения итоговой производительности.

А сейчас настал удобный момент представить вам другую категорию баз данных, которые часто относят к технологии NoSQL.

3.2. Графовые базы данных

Графовые базы данных — белые вороны в стае баз данных NoSQL. Причиной разработки большинства баз данных NoSQL стала необходимость работать на кластерах, которая привела к агрегатно-ориентированным моделям больших записей с простыми связями. Графовые базы данных появились как решение другой проблемы и поэтому имеют противоположную модель — маленькие записи со сложными связями, как показано на рис. 3.1.

В таком контексте этот граф — не диаграмма, а структура данных с узлами, соединенными ребрами.

На рис. 3.1 показана веб-информация с очень маленькими узлами и многочисленными связями между ними. Работая с этой структурой, мы можем задавать вопросы вроде "найти книгу в категории "Базы данных", написанную кем-то, чей друг мне нравится".

Графовые базы данных специально предназначены для хранения такой информации — но в более крупном масштабе, чем можно показать на диаграмме. Они идеально подходят для хранения любых данных, связанных со сложными отношениями, например, социальных сетей, товарных предпочтений или правил приема на работу.

Фундаментальная модель данных графовых баз очень простая: узлы, соединенные ребрами (которые называют также дугами). Помимо этой существенной характеристики, существует много вариаций в моделях данных — в частности, в том, какие

механизмы используются для хранения узлов и ребер. Например, база FlockDB, представляет собой простую совокупность узлов и ребер без какого-либо механизма для дополнительных атрибутов, Neo4J позволяет присоединять Java-объекты в качестве свойств узлов и ребер в неструктурированном виде (см. раздел 11.2, "Функциональные возможности"); а Infinite Graph хранит Java-объекты, являющиеся экземплярами подклассов таких встроенных типов, как узлы и ребра.

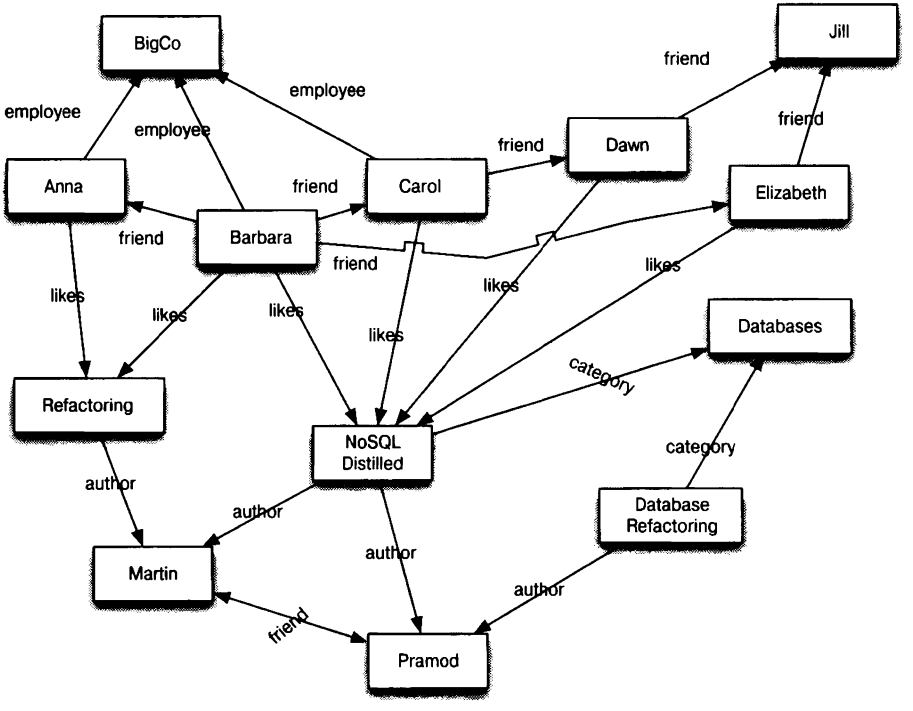


Рис. 3.1. Пример графовой структуры

Как только вы построите граф узлов и ребер, графовая база данных позволит вам послать запрос к сети, способной выполнять операции над запросами, относящимися к таким графам. В этом проявляется важное различие между графовыми и реляционными базами данных. Несмотря на то что реляционные базы данных могут реализовывать связи с помощью внешних ключей, операции соединения требуют навигации, которая может оказаться затратной. Следовательно, в моделях данных с большим количеством связей производительность упадет.

В графических базах данных обход узлов требует очень небольших затрат. В основном это объясняется тем, что графовые базы данных переносят бо́льшую часть работы, связанной с навигацией по связям, с момента запроса на момент вставки. Это естественно оправдывает себя в ситуациях, когда производительность запроса важнее скорости вставки.

Бо́льшую часть времени вы ищете данные, перемещаясь по ребрам сети с запросами вроде "назовите мне всех, кто любит Анну и Барбару". Однако вам необходима

отправная точка, поэтому некоторые узлы могут быть индексированы атрибутом, например идентификатором. Таким образом, вы можете начать с поиска идентификатора (например, найти людей с именами Анна и Барбара), а затем начать перемещение по ребрам. Как видим, графовые базы предназначены для ситуаций, в которых бо́льшую часть времени вы перемещаетесь по связям.

Акцент на связях резко отличает графовые базы данных от агрегатно-ориентированных. Это отличие имеет несколько последствий; такие базы данных чаще работают на одном сервере, а не распределены по кластерам. Транзакции ACID должны охватывать несколько узлов и ребер, чтобы обеспечивать согласованность данных. Единственное, что связывает их с агрегатно-ориентированными базами данных, — отрицание реляционной модели и повышенное внимание специалистов, объясняемое интересом к технологии NoSQL.

3.3. Неструктурированные базы данных

Все базы данных NoSQL являются неструктурированными. Когда вы хотите хранить данные в реляционной базе, сначала определяете схему базы данных, т.е. указываете, какие существуют таблицы и столбцы, и задаете типы данных, которые могут содержаться в этих столбцах. Прежде чем сохранить данные, вы должны иметь схему для этого.

В базах данных NoSQL хранение данных происходит много проще. Хранилище типа "ключ–значение" позволяет хранить данные по ключу. Документная база данных по существу делает то же самое, поскольку она не накладывает ограничений на структуру хранящихся документов. Семейство столбцов позволяет хранить любые данные в любом столбце. Графовые базы данных позволяют свободно добавлять новые ребра, а также новые свойства в узлы и ребра.

Сторонники неструктурированных баз данных подчеркивают их свободу и гибкость. Имея схему, вы должны заранее угадать, что вам потребуется хранить, а сделать это бывает трудно. Используя неструктурированные базы данных, вы можете легко изменять хранилище данных, по мере увеличения знаний о своем проекте. Обнаруживая новые сущности, вы можете легко их добавлять. Более того, если выяснится, что некую сущность вам больше не надо хранить, вы можете просто перестать это делать, не беспокоясь о потере старых данных. Работая с реляционной схемой, вы должны были бы обеспечить сохранность старых данных при удалении столбцов.

Помимо обеспечения удобных изменений, неструктурированные базы данных облегчают обработку *неоднородных данных*, т.е. данных, в которых все записи имеют разные наборы полей. Схема помещает все строки таблицы в "смирительную рубашку". Это может оказаться неудобным, если в разных строках хранятся разные данные. Вы либо останетесь с множеством столбцов, заполненных нулями (т.е. с разреженной матрицей), либо с бессмысленными столбцами вроде `custom column 4`. Неструктурированные базы данных позволяют избежать этого, позволяя каждой записи хранить все, что требуется, — ни больше ни меньше.

Отсутствие структуры выглядит привлекательно. Это позволяет избежать многих проблем, существующих в базах данных с фиксированной схемой, но порождает новые. Если вы только храните данные и выводите их на экран в виде простого списка, состоящего из строк `fieldName: value`, то неструктурированные базы данных — это то, что нужно. Но обычно мы делаем с базами не только это. Как правило, мы пишем программы для обработки данных, которые должны знать, что адрес заказчика, например, называется `billingAddress`, а не `addressForBilling`, а поле `quantify` будет хранить целое число 5, а не слово `five`.

Крайне важный, хотя и неудобный факт заключается в том, что когда мы пишем программу, получающую доступ к данным, она практически всегда подразумевает какую-то форму неявной схемы. За исключением случаев, когда мы пишем простой код наподобие

```
//псевдокод
foreach (Record r in records) {
  foreach (Field f in r.fields) {
    print (f.name, f.value)
  }
}
```

мы должны подразумевать определенные имена полей и некоторые осмысленные данные, а также делать предположения о типах данных, хранящихся в этих полях. Программа — не человек, она не может прочитать слово "qty" и сделать вывод, что оно означает "quantity", — по крайней мере, если вы не запрограммировали ее на это. Итак, даже в неструктурированных базах данных обычно существует неявная схема — набор предположений о структуре данных в коде, манипулирующем этими данными.

Наличие неявной схемы в коде приложения порождает несколько проблем. Неявная схема означает, что, для того чтобы понять, какие данные хранятся в базе, вы должны заглянуть в код приложения. Если этот код хорошо структурирован, вы сразу найдете место, по которому можно определить схему. Но это ничем не гарантируется; все зависит от того, насколько ясным является код. Более того, база данных никак не отражает наличие схемы — она просто не может использовать схему, чтобы помочь вам выбрать способ хранения данных и эффективно извлекать их. Она не может предотвратить несогласованное манипулирование данными в разных приложениях.

Существует несколько причин, по которым реляционные базы данных имеют, а большинство баз данных в прошлом имели фиксированную схему. Схема имеет большое значение, а отрицание схем в базах данных NoSQL выглядит довольно путающим.

По существу, неструктурированные базы данных переносят схему в код приложения, который к ней обращается. Это становится проблематичным, если к базе данных обращаются несколько приложений, разработанных разными людьми. Эти проблемы можно смягчить несколькими способами. Один из них — инкапсулировать все взаимодействия с базой данных в отдельном приложении и интегрировать его с другими приложениями через веб-сервисы.

Это соответствует современным предпочтениям многих людей, желающих обеспечить интеграцию с помощью веб-сервисов. Другой подход основан на четком разграничении разных областей агрегата, открытых для доступа разных приложений. Это могут быть разные разделы в документной базе данных или разные семейства столбцов в базе данных типа "семейство столбцов".

Несмотря на то что сторонники технологии NoSQL часто критикуют реляционные схемы за негибкость, эта критика не совсем справедлива. Реляционные схемы можно изменить в любой момент с помощью стандартных SQL-команд. По мере необходимости можно создать новые столбцы специальным образом, чтобы хранить в них неоднородные данные. Нам редко приходилось это делать, но это вполне возможно. Тем не менее неоднородность данных — хороший аргумент в пользу неструктурированных баз.

Неструктурированность оказывает большое влияние на изменения, происходящие со временем в структуре баз данных, особенно хранящих однородные данные. Несмотря на то что управление изменениями схемы реляционных баз данных практикуется реже, чем требовалось, это вполне возможно. Аналогично необходимо управлять изменениями, происходящими в способах хранения неструктурированных баз данных, чтобы пользователь мог легко извлечь как старые, так и новые данные. Более того, гибкость неструктурированных баз данных проявляется только внутри агрегата. Если вам требуется изменить границы агрегата, то перенос каждого бита окажется таким же сложным, как и в реляционных базах. Вопросы миграции баз данных мы обсудим позднее (глава 12, "Миграция схем").

3.4. Материализованные представления

Когда мы говорили об агрегатно-ориентированных моделях, мы подчеркивали их преимущества. Если вы хотите получить доступ к заказам, полезно собрать все данные для заказа в одном агрегате, который может храниться и предоставлять доступ как отдельная единица. Однако агрегатная ориентация имеет и недостаток: что произойдет, если товаровед захочет узнать, как часто конкретный товар заказывался на протяжении последних недель? В этом случае агрегатная ориентация мешает, вынуждая вас прочитывать каждый заказ в базе данных, чтобы получить ответ на вопрос. Вы можете облегчить задачу, построив индекс товаров, но все равно агрегатная структура останется неудобной.

Преимущество реляционных баз данных заключается в том, что отсутствие у них агрегатной структуры обеспечивает разнообразный доступ к данным. Более того, они обеспечивают удобный механизм, позволяющий искать данные не так, как они хранятся. Этот механизм называется *представлением.* Представление напоминает реляционную таблицу (оно является отношением), но определяется путем вычислений по базовым таблицам. Когда вы обращаетесь к представлению, база данных вычисляет данные в этом представлении — это удобная форма инкапсуляции.

Представления обеспечивают механизм сокрытия от клиента источника данных. Клиент не знает, были ли данные вычислены или просто взяты из базы данных. Однако вычисление некоторых представлений является затратным. Для решения этой проблемы были изобретены *материализованные представления* (materialized views), т.е. представления, вычисленные заранее и записанные в кеш-память диска. Материализованные представления эффективны при работе с часто считываемыми данными, но могут оказаться устаревшими

Несмотря на то что в базах данных NoSQL нет представлений, они могут иметь заранее вычисленные и кешированные запросы, к которым также применяется термин "материализованное представление". Этот аспект имеет гораздо большее значение для агрегатно-ориентированных баз данных, чем для реляционных баз, поскольку большинство приложений работают с запросами, которые плохо согласуются с агрегатной структурой. (Часто базы данных NoSQL создают материализованные представления с помощью подхода "отображение–свертка" (map-reduce), о котором мы поговорим в главе 7.)

Существуют две основные стратегии создания материализованного представления. Первая стратегия называется интенсивным обновлением, в рамках которой материализованное представление обновляется одновременно с обновлением соответствующих ему данных. В этом случае добавление заказа будет сопровождаться обновлением агрегатов истории покупок для каждого товара. Этот подход хорош, когда материализованное представление читается чаще, чем записывается, и должно быть актуальным. Здесь удобно применить подход, основанный на использовании баз данных приложения, поскольку он облегчает синхронную модификацию данных в базе и материализованном представлении.

Если вы не хотите мириться с дополнительными затратами при каждом обновлении базы данных, то можете через регулярные промежутки времени выполнять пакет заданий, связанных с обновлением материализованных представлений. Вы должны правильно задать допустимый уровень устаревания материализованных представлений, исходя из конкретных условий.

Материализованные представления можно создать за пределами базы данных, считывая данные, вычисляя представление и возвращая его в базу данных. Обычно базы данных самостоятельно создают материализованные представления. В этом случае вы должны задать необходимые вычисления, а база данных выполнит их, когда потребуется, в соответствии с заданными параметрами конфигурации. Это особенно удобно при интенсивном обновлении представлений в рамках постепенного отображения–свертки (подробнее об этом — в разделе 7.3.2, "Постепенное отображение–свертка").

Материализованные представления можно использовать в одном и том же агрегате. Документ заказа может содержать элемент, представляющий собой резюме заказа. Запрос к этому резюме не требует обращений ко всему документу заказа. В базах данных типа "семейство столбцов" для создания материализованных представлений используются разные семейства столбцов. Это позволяет обновлять материализованное представление в ходе одной и той же атомарной операции.

3.5. Моделирование доступа к данным

Как указывалось ранее, при моделировании агрегатов данных необходимо анализировать, как будут считываться данные, а также учитывать побочные эффекты, связанные с использованием агрегатов.

Начнем с модели, в которой все данные о клиенте хранятся в хранилище типа "ключ–значение" (рис. 3.2).

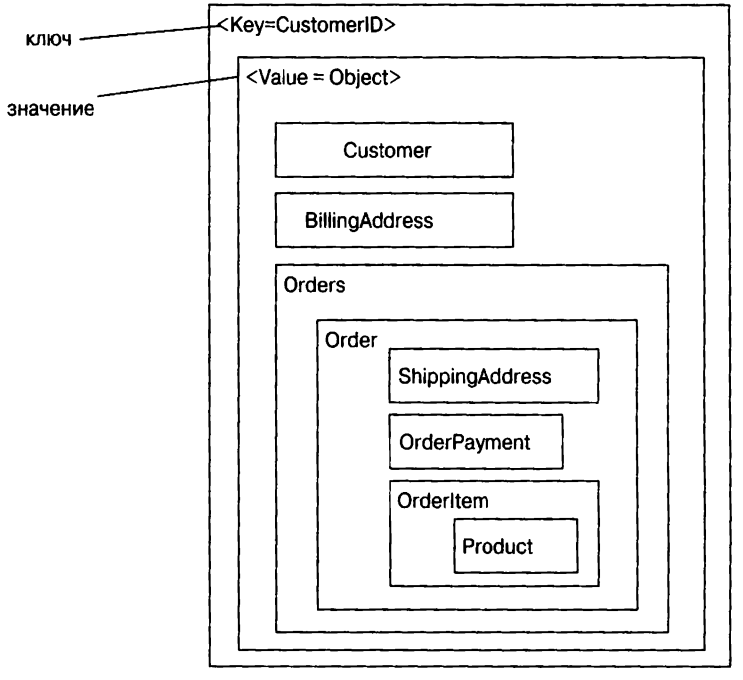


Рис. 3.2. Вложение всех объектов, относящихся к клиенту и его заказам

В этом сценарии приложение может считывать информацию о клиенте и все связанные с ним данные по ключу. Если необходимо считать заказы или товары, проданные по каждому заказу, считывается весь объект, а затем в результате анализа его клиентской части формируются результаты. Если нужны ссылки, можно перейти на документные хранилища, а затем направить запрос в документы или даже изменить данные в хранилище "ключ–значение" и разделить значение между объектами `Customer` и `Order`, а затем обеспечить взаимные ссылки этих объектов.

При работе со ссылками (рис. 3.3) заказы можно искать независимо от объекта `Customer`, а по ссылке `orderId` в объекте `Customer` можно найти все заказы клиента. Использование агрегатов позволяет оптимизировать чтение, но для этого необходимо поместить ссылку `orderId` в объект `Customer` для каждого нового объекта `Order`.

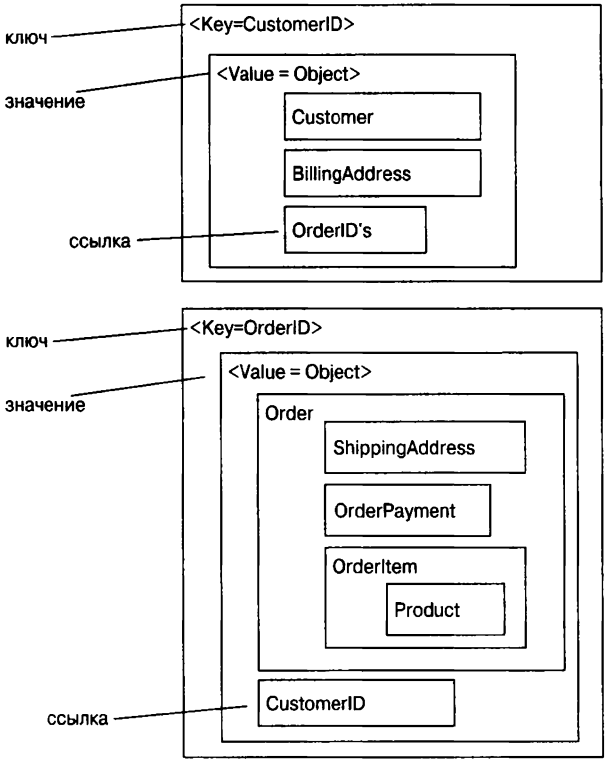


Рис. 3.3. Объект Customer хранится отдельно от объекта Order

```
# Customer object
{
"customerId": 1,
"customer": {
  "name": "Martin",
  "billingAddress": [{"city": "Chicago"}],
  "payment": [{"type": "debit","ccinfo": "1000-1000-1000-1000"}],
  "orders":[{"orderId":99}]
  }
}

# Order object
{
"customerId": 1,
"orderId": 99,
"order":{
  "orderDate":"Nov-20-2011",
  "orderItems":[{"productId":27, "price": 32.45}],
  "orderPayment":[{"ccinfo":"1000-1000-1000-1000",
     "txnId":"abelif879rft"}],
   "shippingAddress":{"city":"Chicago"}
  }
}
```

Агрегаты можно также использовать для получения аналитических данных; например, одновременно с обновлением агрегата можно заполнять информацию о том, какие объекты `Order` содержат заданный объект `Product`. Такая денормализация данных обеспечивает быстрый доступ к данным и лежит в основе процессов **Real Time BI** или **Real Time Analytics**, в которых предприятия не обязаны в конце каждого дня выполнять процедуры заполнения таблиц в хранилищах данных и генерировать аналитические отчеты; теперь они могут заполнять отчеты в момент размещения заказа клиентом.

```
{
"itemid":27,
"orders":{99,545,897,678}
}
{
"itemid":29,
"orders":{199,545,704,819}
}
```

В документных хранилищах запросы можно адресовать в документах, поэтому можно удалить ссылки на объекты `Order` из объекта `Customer`. Это позволяет не обновлять объект `Customer`, когда клиенты размещают новые заказы.

```
# Объект Customer
{
"customerId": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"payment": [
  {"type": "debit",
  "ccinfo": "1000-1000-1000-1000"}
  ]
}

# Объект Order
{
"orderId": 99,
"customerId": 1,
"orderDate":"Nov-20-2011",
"orderItems":[{"productId":27, "price": 32.45}],
"orderPayment":[{"ccinfo":"1000-1000-1000-1000",
   "txnId":"abelif879rft"}],
"shippingAddress":{"city":"Chicago"}
}
```

Поскольку документные хранилища данных позволяют посылать запросы к атрибутам в документе, становятся возможными запросы наподобие "найти все заказы, содержащие товар *Рефакторинг баз данных*", но решение о создании агрегата товаров и заказов следует принимать не из-за возможности таких запросов, а из-за возможности оптимизации чтения базы данных приложением.

При моделировании хранилищ типа "ключ–значение" мы получаем выгоду от упорядоченности столбцов. Это позволяет называть столбцы так, чтобы часто используемые столбцы считывались первыми. При использовании семейств столбцов для моделирования данных важно помнить, что это необходимо для оптимизации запросов, а не записи; общее правило таково: следует облегчить процедуру запроса и денормализовать данные при записи.

Существует много способов моделирования данных; можно хранить объекты `Customer` и `Order` в разных семействах, состоящих из *семейств столбцов* (рис. 3.4). Обратите внимание на то, что ссылка на все заказы, размещенные клиентом, хранится в семействе столбцов `Customer`. Аналогичные процедуры денормализации позволяют повысить эффективность выполнения запроса (чтения).

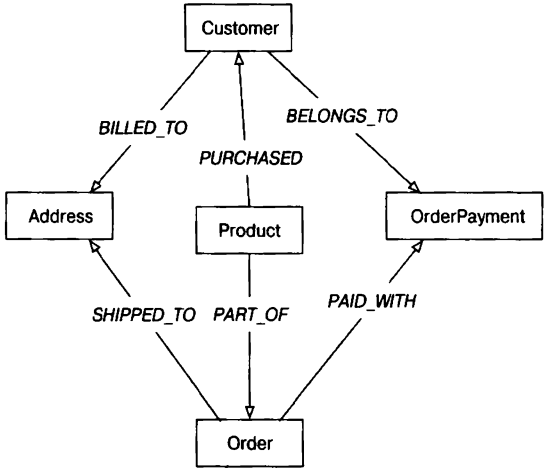


Рис. 3.4. Концептуальное представление о столбцовом хранении данных

При использовании графовой модели для моделирования тех же самых данных мы моделируем все объекты как узлы, а отношения между ними — как связи; эти связи имеют тип на направленность.

Каждый узел имеет независимые связи с другими узлами. Данные связи называются *PURCHASED*, *PAID_WITH* или *BELONGS_TO* (рис. 3.5); эти имена связей позволяют обойти граф. Допустим, вы хотите найти всех клиентов (`Customer`), купивших (*PURCHASED*) товар *Refactoring Database*. Для этого достаточно направить запрос к узлу товаров `Refactoring Databases` и найти все объекты `Customer` с входящей связью *PURCHASED*.

В графовых базовых данных этот тип обхода связей очень легко выполнить. Это особенно удобно, если данные нужны для рекомендации товаров пользователям или определения их покупательских предпочтений.

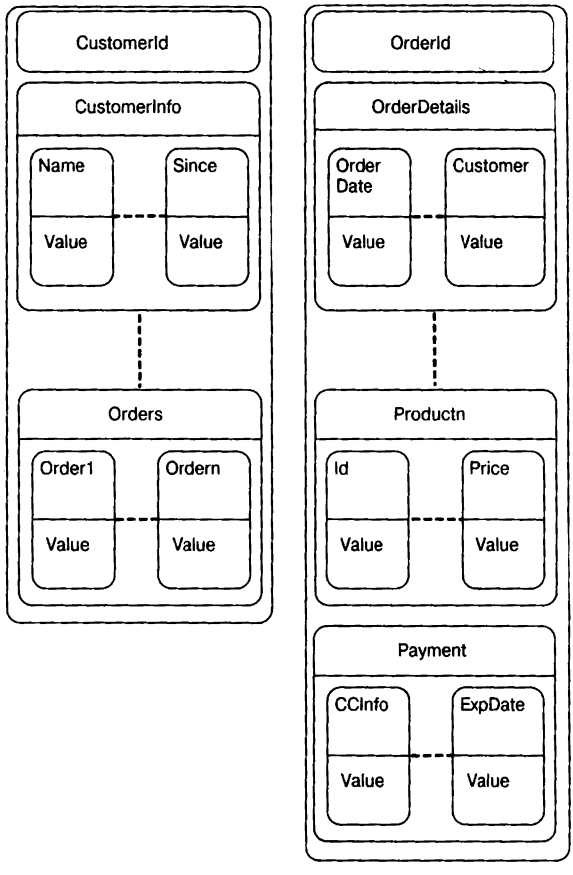


Рис. 3.5. Графовая модель данных для электронной торговли

3.6. Резюме

- Агрегатно-ориентированные базы данных создают межагрегатные связи, которые труднее обрабатывать, чем внутриагрегатные.
- Графовые базы данных организуют данные в виде графа, состоящего из узлов и ребер; они лучше всего работают с данными, имеющими сложную структуру связей.
- Неструктурированные базы данных позволяют легко добавлять поля в записи, но обычно существует неявная схема, подразумеваемая пользователями этих данных.
- Агрегатно-ориентированные базы данных часто вычисляют материализованные представления, чтобы представить пользователям данные, организованные не так, как в их исходных агрегатах. Для этого часто используются вычисления "отображения–свертка".

Глава 4

Модели распределения

Основным свойством технологии NoSQL, вызывающим интерес, является возможность функционирования баз данных на большом кластере. При возрастании объема данных вертикальное масштабирование становится слишком трудным и затратным — необходимо приобретать более крупный сервер для базы данных. Более привлекательным решением является горизонтальное масштабирование — размещение базы данных на кластере серверов. Агрегированная ориентация хорошо согласуется с горизонтальным масштабированием, поскольку агрегат является естественной единицей распределения.

В зависимости от выбранной модели распределения можно создать хранилище данных, предоставляющее возможность обрабатывать больший объем данных, обрабатывать более интенсивный трафик чтения или записи, а также избегать перегрузки и торможения сети. Во многих случаях эти преимущества оказываются очень важными, но за них приходится платить. Работа на кластерах повышает сложность базы, поэтому следует внимательно взвесить все аргументы за и против.

Не вдаваясь в подробности, можно сказать, что существуют два способа распределения данных: *репликация* (replication) и *фрагментация* (sharding). Репликация подразумевает копирование одних и тех же данных на нескольких узлах. Фрагментация означает размещение разных данных на разных узлах. Репликация и фрагментация являются ортогональными методами: можно использовать любую из двух или обе вместе. Репликация бывает двух видов: ведущий–ведомый и одноранговая. Мы рассмотрим эти методы в направлении от простого к сложному: сначала односерверную репликацию, затем репликацию "ведущий–ведомый", потом фрагментацию и наконец одноранговую репликацию.

4.1. Односерверная репликация

Проще всего не выполнять распределение вообще. Лучше запустить базу данных на отдельном компьютере, выполняющем все операции чтения и записи в хранилище данных. Мы предпочитаем это решение, потому что оно исключает все сложности, связанные с другими вариантами; пользователям намного проще управлять такими азами, а разработчикам намного проще их проектировать.

Несмотря на то что множество баз данных NoSQL предназначено для работы на кластере, в тех случаях, когда односерверная модель данных лучше подходит для

приложения, целесообразно использовать технологию NoSQL в сочетании именно с этой моделью. Очевидным выбором в этом случае являются графовые модели, так как они лучше всего работают в односерверной конфигурации. Если обработка данных в основном сводится к работе с агрегатами, то односерверное документное хранилище или хранилище типа "ключ–значение" может оказаться удобным, поскольку его легче разрабатывать.

В последующих разделах настоящей главы мы рассмотрим преимущества и сложности, связанные с более изощренными схемами распределения данных. Не поддавайтесь соблазну выбора сложных схем распределения. Если вы можете обойтись без распределения данных, всегда выбирайте односерверный подход.

4.2. Фрагментация

Довольно часто сложность хранилища данных объясняется тем, что разные пользователи обращаются к его наборам данных. В этих условиях можно поддержать горизонтальное масштабирование, разместив разные части данных на разных серверах. Этот метод называется *фрагментацией* (рис. 4.1).

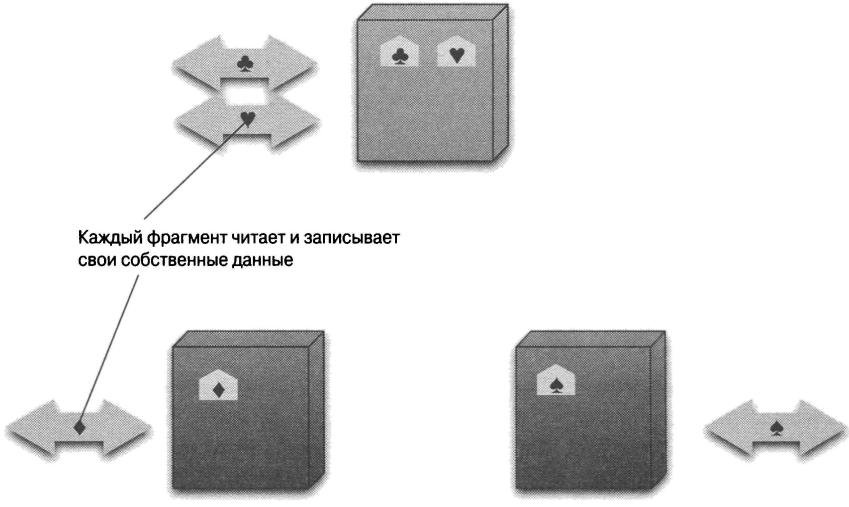


Рис. 4.1. Фрагментация разных данных по отдельным узлам, каждый из которых выполняет свои собственные процедуры чтения и записи

В идеальном случае разные пользователи будут обращаться к разным узлам. Каждый пользователь будет общаться с одним сервером и благодаря этому получать от него быстрые ответы. Например, если вы работаете с десятью серверами, каждый из них будет загружен только на 10%.

Разумеется, идеальный случай крайне редок. Для того чтобы приблизиться к идеалу, мы должны гарантировать, что данные, которые запрашиваются одновременно, размещаются вместе на одном и том же узле, чтобы ускорить доступ к ним.

Первая часть этого вопроса заключается в том, как сгруппировать данные так, чтобы один пользователь в основном получал данные с одного сервера. В этом нам поможет агрегатная ориентация. Главное свойство агрегатов заключается в том, что их разрабатывают для объединения данных, которые, как правило, запрашиваются одновременно, поэтому агрегаты являются естественной единицей распределения.

Когда приходится организовывать данные по узлам, несколько факторов могут повысить производительность работы базы данных. Если вы знаете, что большинство запросов к определенным агрегатам исходят из определенного физического адреса, то можно разместить данные поближе к этому адресу. Если вы получаете заказ от клиента, живущего в Бостоне, то логично поместить данные в хранилище данных на восточном побережье США.

Другой фактор — балансирование рабочей нагрузки. Это означает, что вы должны так организовать агрегаты, чтобы они были равномерно распределены по узлам, обеспечивая их равномерную рабочую нагрузку. Она может со временем изменяться, например, если некоторые данные запрашиваются в определенные дни недели, поэтому следует учитывать правила, диктуемые предметной областью.

В некоторых случаях, когда агрегаты могут читаться последовательно, их полезно разместить вместе. В работе Bigtable [Chang etc.] описано хранение строк в лексикографическим порядке и упорядочивание веб-адресов по обратным доменным именам (например, `com.martinfowler`). Таким образом, данные о многих страницах можно найти одновременно, что повышает производительность работы.

Исторически большинство людей выполняли фрагментацию в рамках логики приложения. Вы можете расположить данные обо всех клиентах с фамилиями от A до D в одном фрагменте, а от E до G — в другом. Это усложняет модель программирования, поскольку код приложения должен распределить запросы по разным фрагментам. Более того, изменение баланса между фрагментами означает изменение кода приложения и миграцию данных. Многие базы данных NoSQL предлагают *автоматическую фрагментацию* (auto-sharding), при которой базы данных берут на себя ответственность за размещение данных по фрагментам и предоставление доступа к соответствующему фрагменту. В этом случае использование фрагментации в приложении значительно упрощается.

Фрагментация является особенно ценной с точки зрения производительности, поскольку она может улучшить эффективность чтения и записи. С помощью репликации, особенно кеширования, можно значительно улучшить эффективность чтения, но это мало отражается на приложениях, выполняющих интенсивную запись. Фрагментация обеспечивает горизонтальное масштабирование операций записи.

Фрагментация сама по себе мало влияет на повышение отказоустойчивости баз данных. Несмотря на то что данные размещаются на разных узлах, сбой на узле делает данные фрагмента недоступными точно так же, как это произошло бы в односерверной базе данных. Отказоустойчивость системы зависит от безопасности фрагментов; однако нет ничего хорошего в базе данных, потерявшей часть записей. В односерверных базах данных довольно легко обеспечить работоспособность сервера, но кластеры обычно используют менее надежные компьютеры, на которых чаще происходят отказы. Таким образом, на практике фрагментация скорее снижает отказоустойчивость базы данных.

Несмотря на то что фрагментацию намного проще выполнить с помощью агрегатов, это нелегкий выбор. Некоторые базы данных изначально ориентированы на

фрагментацию. В этом случае они настраиваются на кластер с самого начала разработки и, конечно, в ходе производства. Другие базы данных используют фрагментацию как намеренный отказ от односерверной конфигурации. В таком случае лучше всего начать с односерверной базы данных, а фрагментацию использовать, только если проектная рабочая нагрузка окажется слишком высокой.

В любом случае переход от одного узла к фрагментации создает проблемы. Мы слышали много грустных историй о коллективах, испытывавших неприятности из-за того, что они слишком поздно перешли к фрагментации, когда ее реализация стала практически невозможной, потому что поддержка фрагментации поглотила бы все ресурсы базы данных при переносе данных на новые фрагменты. Урок заключается в том, что фрагментацию следует осуществлять до того, как в ней возникнет потребность, когда у вас еще достаточно ресурсов, чтобы обойтись без нее.

4.3. Репликация "ведущий–ведомый"

При распределении по схеме "ведущий–ведомый" происходит репликация данных по многим узлам. Один узел назначается *ведущим* (master), или главным. Этот ведущий узел является авторитетным источником данных и обычно несет ответственность за выполнение всех модификаций этих данных. Остальные узлы являются *ведомыми* (slaves), или вторичными. Процесс репликации синхронизирует ведомые узлы с ведущим (рис. 4.2).

Репликация "ведущий–ведомый" наиболее полезна для масштабирования, если в базе данных интенсивно выполняется операция чтения. В этом случае базу данных можно масштабировать горизонтально, чтобы выполнить больше запросов на чтение, добавив больше ведомых узлов и направив на них все запросы на чтение. Однако остаются ограничения, связанные с обработкой обновлений на ведущем узле и его возможностями переадресовывать эти обновления. Следовательно, это неудачный выбор для баз данных с интенсивным трафиком записи, хотя разгрузка трафика чтения немного облегчает выполнение рабочей нагрузки, связанной с записью.

Второе преимущество репликации "ведущий–ведомый" — *отказоустойчивость чтения*: если на ведущем узле произойдет отказ, ведомые узлы смогут по-прежнему обрабатывать запросы на чтение. Это полезно, если в базе данных интенсивно выполняются запросы на чтение. Сбой ведущего узла сделает запись невозможной, пока его работа не будет восстановлена или не будет подключен новый ведущий узел. Однако наличие реплик ведущего узла на ведомых узлах ускоряет процесс его восстановления после сбоя, поскольку в качестве ведущего узла может быть немедленно назначен один из ведомых.

Возможность назначать ведомый узел вместо отказавшего ведущего означает, что репликация "ведущий–ведомый" полезна и в тех случаях, когда горизонтальное масштабирование не требуется. Весь трафик чтения и записи можно направлять на ведущий узел, пока ведомый работает как оперативный резерв. В этом случае систему проще всего представлять как односерверное хранилище с оперативным резервом. Вы получаете выгоды односерверной конфигурации, но с повышенной отказоустойчивостью. Это особенно удобно, если вы хотите смягчить последствия сбоя сервера.

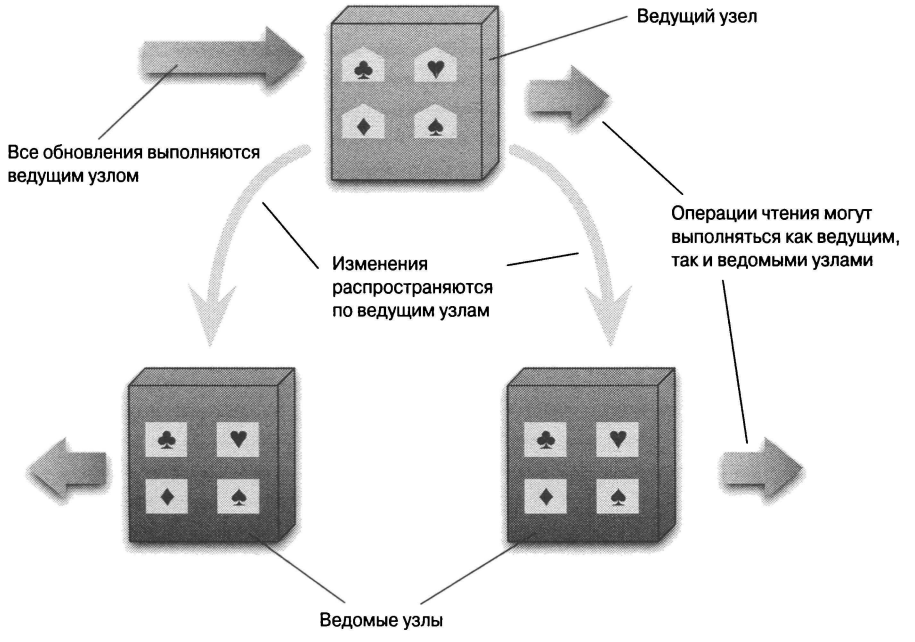


Рис. 4.2. Данные реплицируются от главного узла подчиненным. Запись выполняется службами главного узла; процедуры чтения могут выполняться службами как главного, так и подчиненных узлов

Ведущие узлы можно назначать вручную или автоматически. Ручное назначение обычно означает, что при конфигурации кластера вы настраиваете один узел как ведущий. При автоматическом назначении вы создаете кластер узлов, и они выбирают ведущий узел самостоятельно. Помимо более простой конфигурации, автоматическое назначение означает, что кластер может автоматически назначать новый ведущий узел при сбое старого, уменьшая время простоя.

Для того чтобы обеспечить отказоустойчивость чтения, необходимо гарантировать, что пути чтения и записи в приложении отличаются друг от друга, так что при сбое в ходе записи можно продолжать чтение. Это подразумевает, что операции чтения и записи должны выполняться по разным соединениям базы данных. Эта возможность редко поддерживается библиотеками, обеспечивающими взаимодействие с базами данных. В любом случае нет никакой гарантии, что вы получите отказоустойчивость чтения без тщательной проверки возможности чтения при отказе записи.

Репликация имеет не только привлекательные свойства, но и неизбежный недостаток — несогласованность. Существует опасность, что разные клиенты, читающие данные с ведомых узлов, получат разные значения из-за того, что обновления не успеют распространиться по всем ведомым узлам. В худшем случае клиент не сможет прочитать данные, которые были только что записаны. Это может произойти даже при использовании репликации "ведущий–ведомый" для оперативного резервирования, поскольку при отказе ведущего узла любое обновление, не имевшее резервной копии, будет потеряно. Мы обсудим это подробнее в главе 5, "Согласованность".

4.4. Одноранговая репликация

Репликация "ведущий–ведомый" обеспечивает масштабирование чтения, но не записи. Она обеспечивает отказоустойчивость по отношению к ведомым узлам, но не ведущего. По существу, ведущий узел остается узким местом и единственной точкой отказа. Одноранговая репликация (рис. 4.3) решает эту проблему, устраняя ведущий узел. Все реплики имеют одинаковый вес, все могут выполнять операции записи, и потеря любой из них не приводит к потере доступа к хранилищу данных.

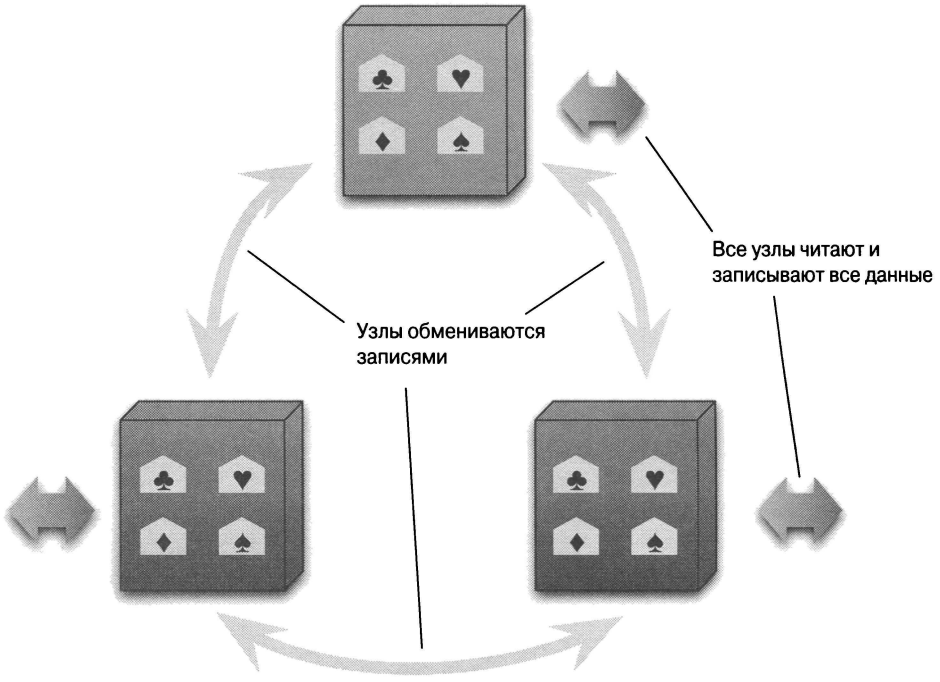


Рис. 4.3. При одноранговой репликации все узлы читают и записывают все данные

Все выглядит прекрасно. Работая с кластером по схеме одноранговой репликации, вы можете справляться со сбоями, не теряя данные. Более того, можно легко добавлять узлы, чтобы повысить производительность работы. Все замечательно, но есть сложности.

Самая большая сложность, как и раньше, — согласованность. Когда вы выполняете запись в два разных места, вы рискуете тем, что два человека попробуют обновить одну и ту же запись в один и тот же момент времени, — таким образом возникает конфликт "запись–запись". Несогласованность чтения тоже приводит к проблемам, но они являются преодолимыми. Несогласованность записи имеет необратимый характер.

Позднее мы поговорим подробнее о том, как справиться с несогласованностью записи, а пока укажем на ряд возможностей. С одной стороны, можно скоординировать реплики так, чтобы при записи данных не возник конфликт. Это обеспечит такую

же сильную гарантию, как и ведущий узел, несмотря на стоимость сетевого трафика, необходимого для координации записей. Нет никакой необходимости согласовывать операции записи на всех репликах. Достаточно ограничиться большинством, чтобы обеспечить выживаемость базы данных при потерях на меньшинстве реплик.

С другой стороны, можно попытаться преодолеть несогласованность операций записи. Существуют стратегии, позволяющие объединять несогласованные операции записи. В этом случае можно обеспечить полноценную производительность при записи на любой реплике.

Эти варианты завершают рассматриваемый спектр возможностей по обеспечению согласованности данных.

4.5. Сочетания фрагментации и репликации

Репликацию и фрагментацию можно сочетать. Если вы используете репликацию "ведущий–ведомый" и фрагментацию (рис. 4.4), это значит, что у вас есть несколько ведущих узлов, но каждая единица данных имеет только один ведущий узел. В зависимости от конфигурации вы можете назначить узел ведущим для одних данных и ведомым для других или совместить обязанности ведущего и ведомого на одном узле.

Использование одноранговой репликации и фрагментации — распространенная стратегия в базах данных типа "семейство столбцов". В этом сценарии на кластере могут существовать десятки и сотни узлов с фрагментированными между ними данными. Для начала удобно выбрать коэффициент репликации равным трем, чтобы каждый фрагмент существовал на трех узлах. Если на узле произойдет сбой, то фрагменты на этом узле будут созданы из фрагментов на других узлах (рис. 4.5).

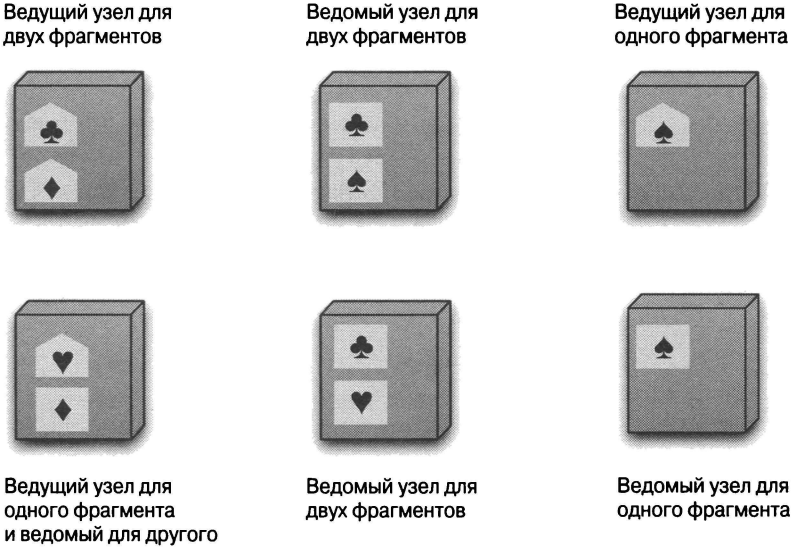


Рис. 4.4. Репликация "главный–подчиненный" с горизонтальным масштабированием

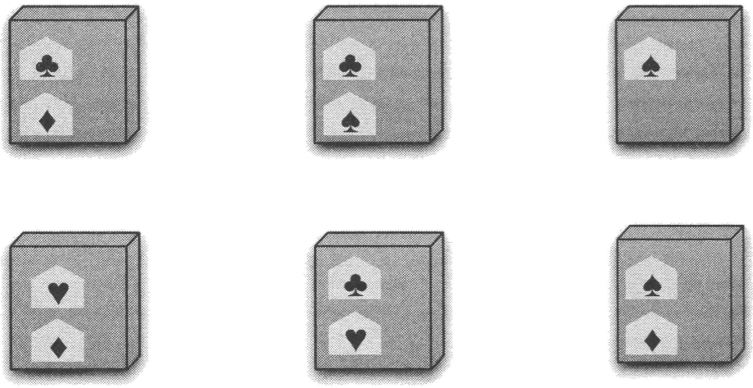


Рис. 4.5. Одноранговая репликация и горизонтальное масштабирование

4.6. Резюме

- Существуют два способа распределения данных.
  - При фрагментации разные данные распределяются по нескольким серверам, чтобы каждый сервер действовал как отдельный источник подмножества данных.
  - При репликации данные копируются между несколькими серверами, так что каждый бит данных можно найти в разных местах.

Система может использовать один из этих способов или оба одновременно.

- Существуют два вида репликации.
  - При репликации "ведущий–ведомый" один узел назначается авторитетным узлом, обрабатывающим записи, а ведомые узлы синхронизируют свою работу с ведущим и могут выполнять операции чтения.
  - При одноранговой репликации операции записи выполняются на любом узле; узлы координируют свою работу, чтобы синхронизировать свои копии данных. Репликация "ведущий–ведомый" уменьшает вероятность конфликта при обновлении базы данных, а одноранговая репликация предотвращает выполнение всех операций записи на одном сбойном узле.

Глава 5

Согласованность

Одним из крупнейших изменений при переходе от централизованной реляционной базы данных к базе данных NoSQL, ориентированной на кластер, является способ обеспечения согласованности. Реляционные базы данных пытаются обеспечить *строгую согласованность* (strong consistency), избегая всех возможных несогласованностей, которые мы вскоре обсудим. При обсуждении технологии NoSQL мы слышим фразы вроде "теорема CAP" и "итоговая согласованность", поэтому должны подумать о том, какой вид согласованности должны обеспечить в своей системе.

Согласованность данных проявляется в разных формах. Этим термином называют огромное множество ошибок, способных закрасться в вашу систему. Таким образом, необходимо обсудить разные виды возможной согласованности. Проанализировав эту проблему, мы можем ослабить требования согласованности (и связанной с ней долговечности).

5.1. Согласованность обновлений

Представьте себе процедуру обновления телефонного номера. Совершенно случайно Мартин и Прамод обнаружили, что на веб-сайте компании указан устаревший номер телефона. Как ни странно, они оба имеют право вносить обновления и одновременно решили обновить номер. Для того чтобы пример стал интересным, предположим, что это обновление они выполняют немного по-разному, потому что каждый из них использует свой формат. Эта проблема называется *конфликтом "запись–запись"* (write-write conflict): она возникает, когда два человека обновляют одни и те же данные в один и тот же момент времени.

Когда записи достигают сервера, тот их *сериализирует* — решает обработать одну, а потом другую. Допустим, он использует алфавитный порядок и извлекает сначала обновление Мартина, а затем — Прамода. Если контроля согласованности нет, то обновление Мартина будет выполнено, а затем немедленно перекрыто обновлением Прамода. В этом случае обновление Мартина называется *потерянным*. В данном случае потерянное обновление не составляет большой проблемы, но часто это не так. Это явление можно считать нарушением согласованности, потому что обновление Прамода использовало состояние до обновления Мартина, но применялось после него.

Подходы к обеспечению согласованности в контексте параллельной работы принято разделять на пессимистический и оптимистический. *Пессимистический* подход ориентирован на предотвращение возникших конфликтов, а *оптимистический* подход допускает возникновение конфликтов, но подразумевает их идентификацию и меры по их устранению. При конфликтах обновления большинство методов в рамках пессимистического подхода предусматривают блокировку записи, т.е. для внесения значения пользователь должен получить право на владение блокировкой, а система гарантирует, что в каждый момент времени право на владение блокировкой имеет только один клиент. Таким образом, Мартин и Прамод могли бы одновременно попытаться получить во владение блокировку записей, но только Мартин (будучи первым) получил бы это право. В этом случае Прамод мог бы увидеть результат записи Мартина и решить, следует ли делать свое обновление.

Оптимистический подход основан на *условном обновлении*, при котором любой клиент, выполняющий обновление, сначала проверяет, не было ли это значение изменено после последнего чтения. В данном случае обновление Мартина было бы принято, а обновление Прамода — нет. Эта ошибка позволила бы Прамоду еще раз проверить значение и решить, следует ли пытаться вносить дальнейшее обновление.

И пессимистический, и оптимистический подходы, описанные выше, основаны на согласованной сериализации обновлений. На одном сервере это очевидно — он должен выбрать одно обновление, а затем другое. Но при работе на нескольких серверах, как, например, в случае одноранговой репликации, два узла могут выполнять обновления в разном порядке, и в итоге на каждом узле будут записаны разные телефонные номера. Часто, когда люди говорят о параллельной работе в распределенных системах, они говорят о последовательной согласованности, которая гарантирует, что все узлы будут выполнять операции в одном и том же порядке.

Существует еще один оптимистический способ разрешения конфликта "запись–запись" — сохранить конфликтующие обновление и запись. Этот подход знаком многим программистам по системам управления версиями, в частности, по распределенным системам управления версиями, которые по своей природе часто получают конфликтующие записи. Следующий этап также является повторением решения, принятого в системах управления версиями: вы должны как-то объединить обновления. Можно показать оба значения пользователю и попросить его разобраться в них, — именно так происходит, когда вы обновляете телефонные номера в своей записной книжке или компьютере. В качестве альтернативы компьютер может самостоятельно выполнить объединение; если проблема заключается в формате телефонного номера, он может распознать ее и записать новый номер в стандартном формате. Любое автоматическое объединение при разрешении конфликта "запись–запись" носит узкоспециализированный характер и программируется в каждом конкретном случае поразному.

Когда люди впервые сталкиваются с такой проблемой, они часто предпочитают пессимистический подход, чтобы вообще предотвратить конфликты. Несмотря на то что в некоторых ситуациях это правильно, всегда существует компромисс. Параллельное программирование основано на базовом компромиссе между безопасностью (предотвращением ошибок, таких, как конфликты обновлений) и "живостью" (быстрой реакцией на запросы клиентов). Пессимистические подходы часто настолько сильно

снижают скорость реакции системы, что становятся бесполезными. Эта проблема усугубляется опасностью ошибок, так как пессимистическая параллельность часто приводит к возникновению взаимных блокировок, которые трудно предотвратить и отладить.

Репликация повышает вероятность конфликтов "запись–запись". Если на разных узлах хранятся разные копии, которые обновляются независимо друг от друга, то без специальных мер предосторожности конфликты возникнут обязательно. Если в качестве цели для всех записей определенных данных используется один узел, обеспечить согласованность много проще. Это решение использовалось во всех распределенных моделях, рассмотренных выше, кроме одноранговой репликации.

5.2. Согласованность чтения

Если хранилище данных обеспечивает согласованность обновлений, это не гарантирует, что читатели этих данных будут всегда получать на свои запросы согласованные ответы. Представьте, что у нас есть заказ на определенные товары с определенными расходами на доставку. Расходы на доставку рассчитываются на основе товаров, указанных в заказе. Если мы добавляем новую позицию, то должны выполнить вычисления заново и обновить запись о расходах на поставку. В реляционной базе данных расходы на поставку и товарные позиции хранятся в разных таблицах. Опасность несогласованности заключается в том, что Мартин добавляет позицию в свой заказ, а Прамод затем считывает эти позиции и расходы на доставку, а после этого Мартин обновляет запись о расходах на доставку. Этот вид ошибки называется *несогласованным чтением* или *конфликтом "чтение–запись"*.

На рис. 5.1 показана ситуация, в котоой Прамод выполнил чтение в середине процедуры записи, выполняемой Мартином.

Мы называем этот тип согласованности *логической согласованностью*. Она гарантирует, что разные элементы данных будут изменяться вместе. Для того чтобы избежать логической несогласованности при конфликте "чтение–запись", реляционные базы данных используют понятие транзакций. Позволяя Мартину упаковать обе записи в одну транзакцию, система гарантирует, что Прамод будет читать оба элемента данных либо до, либо после обновления.

Часто говорят, что базы данных NoSQL не поддерживают транзакции, а значит, не могут быть согласованными. Это высказывание, как правило, является ошибочным, потому что оно не учитывает много важных деталей. Во-первых, высказывания об отсутствии транзакций обычно относятся лишь к некоторым базам данных NoSQL, в частности агрегатно-ориентированным. В противоположность им графовые базы данных обычно поддерживают транзакции ACID почти так же, как реляционные базы данных.

Во-вторых, агрегатно-ориентированные базы данных поддерживают атомарные обновления, но только в рамках отдельного агрегата. Это значит, что мы получим логическую согласованность внутри агрегата, но не между агрегатами. В нашем примере мы

могли бы избежать несогласованности, если бы заказ, стоимость доставки и товарные позиции были частями одного и того же агрегата заказа.

Разумеется, не все данные можно записать в один и тот же агрегат, поэтому любые обновления, влияющие на несколько агрегатов, оставляют интервал времени, в течение которого клиенты могут выполнить несогласованное чтение. Продолжительность этого интервала называется *окном несогласованности* (inconsistency window). Система NoSQL может иметь довольно узкое окно несогласованности. Например, в документации компании Amazon сказано, что окно несогласованности для сервиса SimpleDB обычно не превышает секунды.

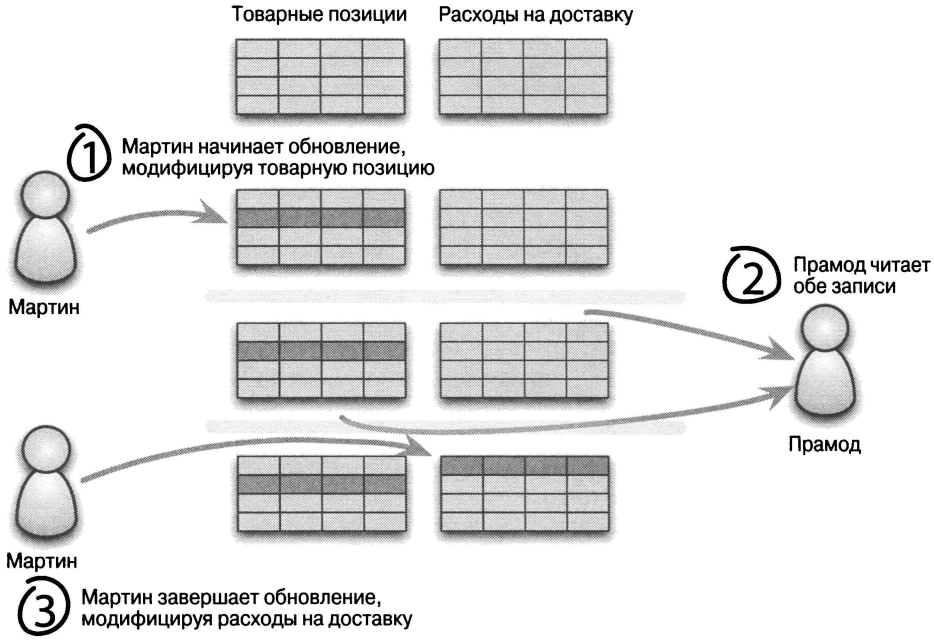


Рис. 5.1. Конфликт "чтение–запись" в логической несогласованности

Этот пример логической несогласованности чтения является классическим. Его можно встретить в любой книге по программированию баз данных. Однако с появлением репликации мы сталкиваемся с новым видом несогласованности. Представьте себе, что в гостинице остался только один свободный номер. Система резервирования номеров работает на нескольких узлах. Мартин и Синди — муж и жена, желающие занять этот номер, но они обсуждают это по телефону, потому что Мартин находится в Лондоне, а Синди — в Бостоне. Тем временем Прамод, находящийся в Мумбаи, приезжает и занимает последний номер. Он обновляет статус доступности реплицированного номера, но его обновление достигает Бостона быстрее, чем Лондона. Когда Мартин и Синди запускают свои браузеры, чтобы выяснить, доступен ли номер, Синди видит, что номер занят, а Мартин — что он свободен. Это новый вид несогласованного чтения, относящийся к *согласованности репликаций* (replication consistency), которая гарантирует, что при чтении с разных реплик один и тот же элемент данных имеет одно и то же значение (рис. 5.2).

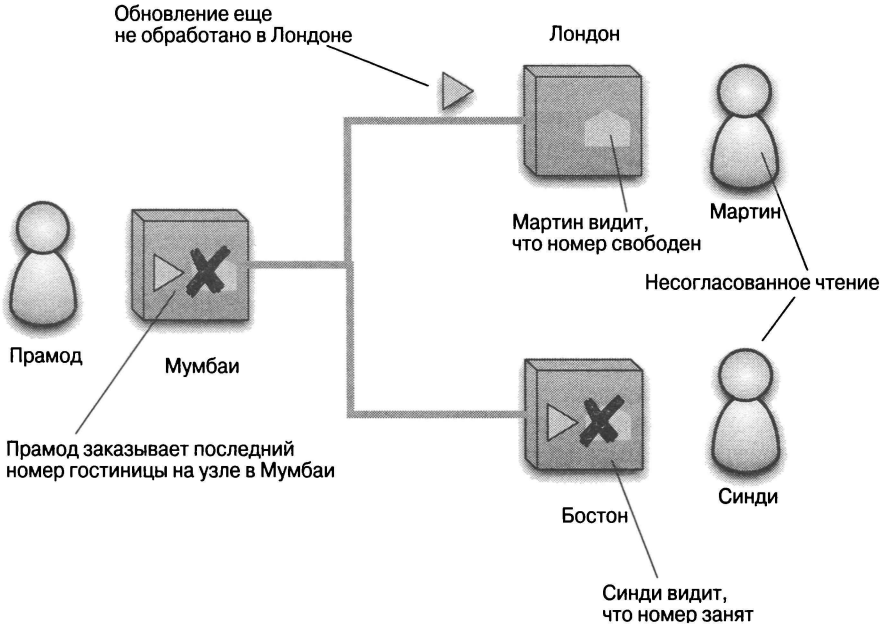


Рис. 5.2. Пример несогласованности репликаций

Разумеется, в конце концов, обновления будут полностью распределены по узлам, и Мартин увидит, что номер занят. По этой причине такая ситуация называется *итоговой согласованностью* (eventually consistent), или *согласованностью "в конечном счете"*. Это значит, что в любой момент времени узлы могут быть несогласованными, но, если нет новых обновлений, в конце концов все узлы будут обновлены и получат одно и то же значение. Прежние данные обычно называются устаревшими, что напоминает нам о том, что кеш — одна из форм репликации. По существу, это соответствует модели распределения "ведущий–ведомый".

Несмотря на то что согласованность репликаций не зависит от логической согласованности, ее отсутствие может усугубить логическую несогласованность из-за расширения окна несогласованности. Два разных обновления на ведущем узле могут быть выполнены непосредственно одно за другим, оставляя окно несогласованности на несколько миллисекунд. Однако задержка в работе сети означает, что на ведомых узлах окно несогласованности останется открытым намного дольше.

Гарантии согласованности не являются глобальными в масштабах приложения. Вы можете задать уровень согласованности в соответствии со своими индивидуальными запросами. Это позволит вам использовать слабую согласованность бо́льшую часть времени, когда это не важно, и требовать строгой согласованности, когда это имеет значение.

Наличие окна несогласованности означает, что разные люди увидят разные вещи в одно и то же время. Если Мартин и Синди ищут свободный номер в ходе трансатлантического телефонного разговора, может возникнуть недоразумение. Намного чаще пользователи действуют независимо друг от друга, и поэтому это не проблема. Но окно несогласованности может стать особенно проблематичным, если вы сталкиваетесь с

несогласованностью собственных обновлений. Рассмотрим в качестве примера публикацию комментариев в блоге. Немногие люди беспокоятся об окне несогласованности, которое может быть открыто на протяжении нескольких минут, пока они записывают свои свежие мысли. Часто системы поддерживают такие сайты на кластере и балансируют рабочую нагрузку за счет пересылки запросов на разные узлы. Здесь возникает опасность: вы можете разместить сообщение на одном узле, затем обновить браузер, но обновление произойдет на другом узле, который может еще не получить обновление ваших записей, — это выглядит так, будто ваша запись пропала.

В таких ситуациях можно смириться с окнами несогласованности, если продолжительность их открытого состояния не превышает разумных пределов, но требуется обеспечить *согласованность чтения–записи собственных записей* (read-your-writes consistency), которая означает, что, сделав обновление, пользователь гарантированно сможет его увидеть. Для этого можно заставить систему обеспечить сессионную согласованность: согласованность чтения–записи собственных записей в рамках одной пользовательской сессии. В противном случае система будет иметь лишь итоговую согласованность. Это значит, что пользователь может потерять согласованность, если сессия прервется по какой-то причине или к той же самой системе обратится пользователь с другого компьютера, но такие случаи относительно редки.

Существует много способов обеспечить сессионную согласованность. Обычно применяют простейший метод — *"липкую" сессию* (sticky session): сессию, привязанную к одному узлу (этот метод также называют *привязкой сессии* (session affinity)). "Липкая" сессия гарантирует, что из согласованности чтения–записи собственных записей на узле следует согласованность чтения–записи собственных записей и для сессий. Недостатком этого способа является то, что он создает препятствия для работы механизма балансировки рабочей нагрузки.

Другой способ обеспечения сессионной согласованности основан на использовании штампов версий (глава 6, "Штампы версий"). Он гарантирует, что каждое взаимодействие с хранилищем данных будет содержать штамп последней версии, видимый в ходе сессии. Узел сервера, со своей стороны, должен гарантировать, что он содержит обновления, содержащие этот штамп версии, прежде чем отвечать на запрос.

Обеспечение сессионной согласованности с помощью "липких" сессий и репликации "ведущий–ведомый" может оказаться неудобным, если вы хотите повысить производительность чтения на ведомых узлах, а записи должны по-прежнему выполнять на ведущем узле. Чтобы справиться с этой задачей, можно посылать записи на ведомые узлы, которые должны пересылать их на ведущий узел, поддерживая сессионную согласованность для своего клиента. Другой подход предусматривает временное переключение сессии на ведущий узел во время записи на достаточно долгое время и выполнять чтение с ведущего узла, пока ведомые узлы не получат обновление.

Мы говорим о согласованности репликаций в контексте хранилища данных, но она играет важную роль во всем проекте приложения. Даже в простых системах управления базой данных возникает много ситуаций, в которых данные представляются пользователю, а он анализирует их и обновляет. Не рекомендуется оставлять транзакцию открытой во время взаимодействия с пользователем, потому что, если пользователь попытается выполнить обновление, возникнет реальная опасность конфликтов, для устранения которых используются автономные блокировки [Fowler PoEAA].

5.3. Ослабление согласованности

Согласованность — это хорошо, но, к сожалению, иногда ею приходится жертвовать. Почти всегда можно разработать систему, предотвращающую несогласованность, но практически невозможно сделать это без ущерба для остальных характеристик системы. В результате часто приходится жертвовать согласованностью в пользу чего-то другого.

Несмотря на то что некоторые архитекторы считают это катастрофой, мы считаем, что это неизбежный компромисс, который возникает при проектировании любой системы. Более того, некоторые предметные области мало чувствительны к несогласованности, и это следует учитывать при выборе проектных решений.

Часто поступаться согласованностью приходится даже при разработке односерверных реляционных систем управления базами данных. Здесь основным инструментом обеспечения согласованности являются транзакции, которые могут обеспечить гарантии строгой согласованности. Однако системы с транзакциями обычно могут снижать уровни изолированности, разрешая запросам читать незафиксированные данные. На практике большинство приложений ослабляют требования к согласованности, понижая уровень изолированности с наивысшего (упорядоченного), чтобы обеспечить более высокую производительность. Мы в основном встречали людей, использующих чтение фиксированных данных, которое исключает некоторые конфликты чтения–записи, но разрешает другие.

Многие разработчики систем вообще отказываются от использования транзакций из-за того, что они слишком сильно снижают производительность. Это проявляется разными способами. В небольшом масштабе мы видели, насколько популярным был язык запросов MySQL, когда он не поддерживал транзакции. Многие веб-сайты предпочитали высокую скорость MySQL и прекрасно обходились без транзакций. На другом конце шкалы расположены некоторые очень крупные веб-сайты, такие как eBay [Pritchett], отказавшиеся от транзакций в пользу производительности. Это особенно сильно проявляется при фрагментации баз данных. Даже без этих ограничений многие разработчики приложений вынуждены взаимодействовать с удаленными системами, которые невозможно правильно включить в границы транзакций, поэтому обновление баз данных без использования транзакций стало довольно распространенным в промышленных приложениях.

5.3.1. Теорема CAP

В среде приверженцев NoSQL принято ссылаться на теорему CAP как на причину, по которой мы можем ослабить согласованность. Эта теорема была сфомулирована Эриком Брюером (Eric Brewer) в 2000 году [Brewer] и доказана Сетом Гильбертом (Seth Gilbert) и Нэнси Линч (Nancy Lynch) [Lynch and Gilbert] через несколько лет. (Возможно, вы слышали о ней как о гипотезе Брюера.)

Основное утверждение теоремы CAP гласит, что из трех свойств — согласованности данных, доступности и устойчивости к разделению — можно обеспечить не больше двух. Очевидно, это очень сильно зависит от того, как определить эти три свойства.

Обсуждение разных вариантов породило несколько дискуссий о реальных следствиях из теоремы CAP.

Согласованность данных в теореме CAP определяется практически так же, как мы ее определили выше. *Доступность* (availability) в контексте теоремы CAP имеет конкретный смысл — она означает, что если вы можете обращаться к узлу кластера, то он может читать и записывать данные. Это немного отличается от обычного смысла данного термина. Мы обсудим эту тему немного позднее. *Устойчивость к разделению* (partition tolerance) означает, что кластер может восстанавливать обмен данными после обрыва связей в кластере, который разделен на многочисленные фрагменты, не способные взаимодействовать друг с другом (эта ситуация называется "разделенным мозгом" (рис. 5.3)).

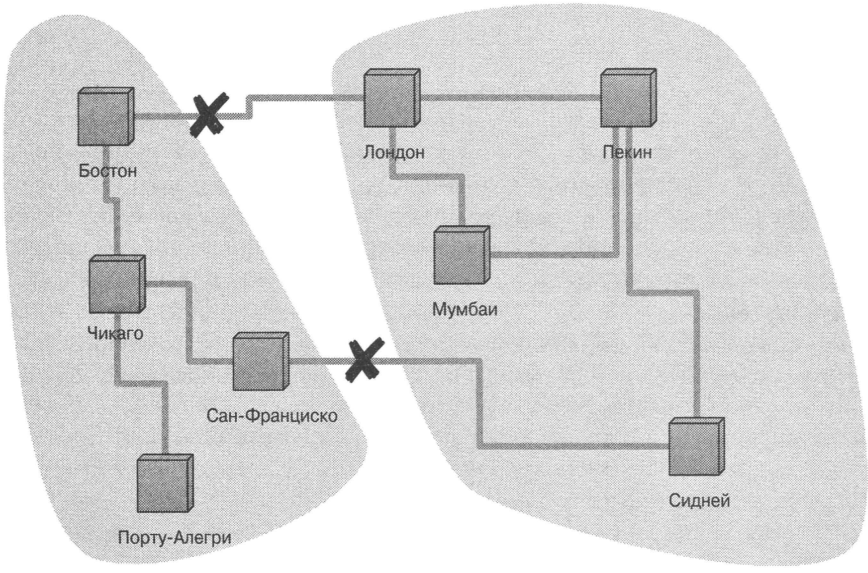


Рис. 5.3. При двух разрывах линий связи сеть разделяется на две группы

Очевидным примером системы со свойствами согласованности данных и доступности (система CA — Consistensy and Availability System), не устойчивой к разделению, является односерверная система. Один компьютер невозможно разделить, поэтому нет причин для беспокойства об устойчивости к разделению. В такой системе есть только один узел, и если он работает, то он доступен. Вполне разумно в ходе работы обеспечивать согласованность данных. В этих условиях существует большинство баз данных.

Теоретически возможно создать кластер CA. Однако это означало бы, что при разделении этого кластера все его узлы должны выключиться, чтобы ни один клиент не мог обратиться к узлу. Однако обычное определение доступности означает противоположное свойство. В этой ситуации особое определение свойства доступности в теореме CAP вводит людей в заблуждение. Теорема CAP определяет "доступность" следующим образом: "на каждый запрос, полученный работающим узлом в системе, должен быть дан ответ" [Lynch and Gilbert]. Итак, неработающий узел не влияет на доступность в смысле теоремы CAP.

Отсюда следует, что можно создать кластер CA, но при этом необходимо гарантировать, что его разделение будет происходить редко и полностью. Это можно сделать, по крайней мере, в центре данных, но, как правило, слишком дорогой ценой. Помните, что для выведения из работы всех узлов кластера при разделении вы должны вовремя идентифицировать это разделение, что само по себе сложно.

Итак, кластеры должны быть устойчивыми к разделению сети. Именно в этом пункте проявляется реальный смысл теоремы CAP. Несмотря на то что теорему CAP часто формулируют следующим образом: "Вы можете получить только два свойства из трех", на практике она означает, что в системе, которая подвержена разделению, например, в распределенной системе, следует искать компромисс между согласованностью и доступностью. Это не выбор из двух альтернатив; часто приходится соглашаться на невысокую согласованность, чтобы достичь определенной доступности. Полученная в результате система не будет ни хорошо согласованной, ни идеально доступной, но она будет представлять собой разумное сочетание этих свойств, удовлетворяющих ваши потребности.

Рассмотрим иллюстративный пример. Мартин и Прамод пытаются заказать номер в гостинице с помощью системы, использующей одноранговое распределение по двум узлам (Лондон — для Мартина и Мумбаи — для Прамода). Если вы хотите обеспечить согласованность, то, когда Мартин попытается заказать номер на лондонском узле, этот узел должен будет связаться с узлом в Мумбаи, прежде чем подтвердить прием заказа. По существу, оба узла должны согласовать сериализацию своих запросов, чтобы обеспечить согласованность. Однако, если сетевое соединение разорвется, то ни один узел не сможет заказать ни один номер в гостинице, т.е. исчезнет доступность.

Для того чтобы улучшить доступность, можно назначить один узел ведущим для конкретной гостиницы и сделать так, чтобы все заказы обрабатывались этим узлом. Если бы этот узел находился в Мумбаи, то он смог бы принять заказ и Прамод занял бы последний номер. Если бы мы использовали репликацию "ведущий–ведомый", то пользователи в Лондоне могли бы получить несогласованную информацию о заказах номеров, но не смогли бы сделать заказ, и, значит, возникла бы несогласованность обновлений. Однако пользователи могли бы подождать разрешения этой ситуации — так работает компромисс в данном случае.

Это улучшает ситуацию, но мы по-прежнему не можем заказать из Лондона номер в гостинице, ведущий узел которой расположен в Мумбаи, если связь с ним разорвана. В терминологии теоремы CAP это — отказ доступности, который проявляется в том, что Мартин может обращаться к узлу в Лондоне, но этот узел не может обновить данные. Для того чтобы повысить доступность, мы могли бы разрешить обеим системам принимать заказы, даже если сетевое соединение вышло из строя. Опасность заключается в том, что и Мартин, и Прамод забронируют последний номер. Однако, в зависимости от того, как работает гостиница, это может быть даже хорошо. Часто туристические компании допускают определенный избыток заказов, чтобы компенсировать неявку туристов. И наоборот, некоторые отели резервируют несколько номеров, даже если книга заказов полна, чтобы иметь возможность менять номера при возникновении проблем или принимать поздние заказы высокого статуса. Некоторые гостиницы в случае конфликта даже отменяют некоторые заказы, принося свои извинения. Причина такого решения заключается в том, что стоимость нескольких отказов меньше, чем стоимость потери заказов из-за сбоев сети.

Классическим примером разрешения несогласованных записей является корзина заказов, описанная в документе Dynamo [Amazon's Dynamo]. В этом случае вы всегда можете сделать запись в своей корзине заказов, даже если сбои сети приведут к тому, что у вас возникнет несколько корзин. Процесс оформления заказа может объединить две корзины в одну. Это практически всегда оказывается удачным решением — а если нет, то пользователь имеет возможность просмотреть корзину перед оформлением заказа.

Урок заключается в следующем. Несмотря на то что большинство разработчиков программного обеспечения считают согласованность обновлений совершенно обязательным, существуют ситуации, в которых прекрасно можно справиться с несогласованными ответами на запросы. Эти ситуации тесно связаны с предметной областью и требуют от разработчика хорошего знания предмета. Итак, эту проблему невозможно решить в узком кругу программистов — необходимо общаться с экспертами в предметной области. Если вы можете найти способ справиться с несогласованными ответами, то сможете повысить доступность и производительность. Например, для корзины заказов это означает, что покупатели всегда могут совершать покупки, и делать это быстро. Будучи патриотичными американцами, мы знаем, насколько важно выполнять наше предназначение в области розничной торговли.

Аналогичная логика применима к согласованности чтения. Если вы торгуете ценными бумагами на электронной бирже, то никогда не смиритесь с устаревшими данными. Однако если вы размещаете комментарии на веб-сайте новостей, то можете потерпеть старые новости несколько минут. В этих ситуациях необходимо знать, насколько высокой может быть толерантность к чтению устаревших данных и насколько широким может быть окно несогласованности. Часто оно выражается в терминах средней длины, наименьшей длины или определенного распределения длин. Разные данные обусловливают разную степень толерантности к их устареванию, а значит, нужны разные настройки в конфигурации ваших репликаций.

Сторонники технологии NoSQL часто говорят, что вместо свойств ACID реляционных транзакций системы NoSQL обеспечивают свойства BASE (доступность в большинстве случаев, неустойчивое состояние, итоговая согласованность — Basically Available, Soft state, Eventual consistency) [Brewer]. Мы считали себя обязанными упомянуть эту аббревиатуру, хотя не считаем ее очень полезной. Она еще более надуманная, чем ACID, потому что ни "доступность в большинстве случаев", ни "неустойчивое состояние" не имеют четкого определения. Следует подчеркнуть, что когда Брюер ввел понятие BASE, мы рассматривали компромиссы в диапазоне от ACID до BASE.

Мы включили в книгу обсуждение теоремы CAP, потому что она часто используется (и часто неправильно) при обсуждении компромиссов, касающихся согласованности в распределенных базах данных. Однако лучше размышлять не о компромиссе между согласованностью и доступностью, а о компромиссе между согласованностью и *временем ожидания* (latency). Подведем итог нашего обсуждения, сказав, что согласованность можно улучшить, включив в систему больше взаимодействующих узлов, но каждый дополнительный узел увеличивает время реакции. Доступность можно рассматривать как предельное время отклика, которое мы можем допустить; если время отклика становится слишком высоким, мы считаем данные недоступными, — это точно соответствует его определению в контексте концепции CAP.

5.4. Ослабление долговечности

До сих пор мы говорили о согласованности, которую большинство людей считают основным свойством транзакций ACID. Ключевым аспектом согласованности является сериализация запросов путем формирования атомарных изолированных единиц работы. Но у большинства людей упоминание об ослаблении долговечности вызывает усмешку — помимо всего прочего, зачем же нужно хранилище данных, если оно может терять обновления?

Оказывается, существуют ситуации, в которых можно пожертвовать определенной долговечностью в пользу более высокой производительности. Если база данных в основном работает в оперативной памяти, то обновления происходят в ее представлении в оперативной памяти, а изменения периодически сбрасываются на диск; это позволяет повысить скорость реакции на запросы. Взамен, если сервер выйдет из строя, любые изменения, внесенные после последней записи на диск, будут потеряны.

Примером этого компромисса является хранение состояния сеанса пользователя. Большой веб-сайт может иметь много пользователей и хранить временную информацию о действиях каждого пользователя в ходе последнего сеанса. Пользователь мог выполнять разные действия, влияющие на скорость реакции веб-сайта. Ключевым моментом является то, что потеря сеансовых данных не станет трагедией — она создаст небольшие неудобства, но все же меньшие, чем замедление работы веб-сайта. В этой ситуации логично применить недолговечные записи. Часто можно определять долговечность на основе позывных, так что более важные обновления можно принудительно записывать на диск.

Другим примером ослабления долговечности является сбор телеметрических данных с физических устройств. Возможно, лучше собирать данные побыстрее за счет потери последних обновлений при сбое сервера.

Другой класс компромиссов, связанных с долговечностью, образуют реплицированные данные. Снижение *долговечности репликации* (replication durability) возникает, когда узел обрабатывает обновление, но еще до того, как это обновление будет реплицировано на другие узлы, возникает сбой. Например, это может случиться, если вы используете модель распределения "ведущий–ведомый", в которой ведомый назначается новым ведущим узлом автоматически в случае отказа прежнего ведущего узла. Если ведущий узел выйдет из строя, все записи, не переданные репликам, будут потеряны. Как только ведущий узел вернется в строй, эти обновления вступят в конфликт с новыми обновлениями, которые были внесены после сбоя. Эта проблема связана с долговечностью, потому что вы считаете, что ваши обновления были обработаны сразу, как только достигли ведущего узла, но сбой на ведущем узле привел к их потере.

Если вы уверены, что ведущий узел быстро восстановит свою работу, то переключаться на ведомый узел не обязательно. В противном случае вы можете повысить долговечность репликации, сделав так, чтобы ведущий узел ожидал, пока некоторые реплики подтвердят получение обновления, и только после этого подтверждал их получение от клиента. Однако очевидно, что этот подход замедляет процесс обновления и выводит кластер из строя, если на ведомом узле произойдет сбой, — итак, мы снова

должны идти на компромисс, зависящий от того, насколько важной для нас является долговечность данных. В обычной ситуации полезно, чтобы индивидуальные вызовы содержали информацию о допустимом уровне долговечности.

5.5. Кворумы

Выбирая между согласованностью и долговечностью, не обязательно руководствоваться принципом "все или ничего". Чем больше узлов задействовано в вашем запросе, тем выше вероятность, что согласованность не возникнет. Естественным образом возникает вопрос: сколько узлов должно быть вовлечено в запрос, чтобы обеспечить строгую согласованность данных?

Представьте себе ситуацию, в которой данные реплицированы на трех узлах. Совершенно не нужно, чтобы все узлы подтверждали запись для обеспечения строгой согласованности; нужно, чтобы это сделали два из них, т.е. большинство. Если возникнут конфликтующие записи, то большинство получит только одна из них. Это явление называется *кворумом записи* (write quorum) и выражается несколько претенциозным неравенством `W > N/2`, означающим, что количество узлов, участвующих в записи (`W`), должно превышать половину количества узлов, задействованных в чтении (`N`). Количество реплик часто называется коэффициентом репликации.

Аналогично кворуму записи, существует кворум чтения: со сколькими узлами следует установить контакт, чтобы гарантировать, что вы получаете самое последнее изменение. Кворум чтения представляет собой немного более сложное понятие, потому что он зависит от того, сколько узлов должны подтвердить запись. Рассмотрим ситуацию, в которой коэффициент репликации равен 3. Если все записи должны подтвердить два узла (`W = 2`), то мы должны установить контакт по крайней мере с двумя узлами, чтобы гарантировать получение последних данных. Если же записи подтверждаются только одним узлом (`W = 1`), то мы должны связаться со всеми тремя узлами, чтобы гарантировать получение последних обновлений. В последнем случае у нас нет кворума записи, поэтому возникает конфликт обновлений, но, контактируя с достаточно большим количеством читателей, мы можем гарантированно обнаружить конфликт. Таким образом, мы можем получить строго согласованные результаты чтения, даже если у нас нет строгой согласованности записей.

Это отношение между количеством узлов, с которыми необходимо связаться при чтении (`R`), узлов, подтверждающих запись (`W`), и коэффициентом репликации (`N`) можно выразить в виде неравенства: строгую согласованность можно получить, если `R + W > N`.

Эти неравенства выведены для одноранговой модели распределения. В случае распределения "ведущий–ведомый", чтобы избежать конфликтов "запись–запись", достаточно записать данные на ведущий узел. Аналогично, для того чтобы избежать конфликтов "чтение–запись", достаточно выполнять чтение только с ведущего узла. В использованной системе обозначений легко перепутать количество узлов в кластере с коэффициентом репликации, но часто это совершенно разные числа. При фрагментации баз данных вы можете иметь 100 узлов в кластере при коэффициенте репликации, равном 3.

В большинстве случаев руководство считает, что для обеспечения хорошей отказоустойчивости достаточно коэффициента репликации, равного 3. Это позволяет сохранить кворумы чтения и записи при сбое одного узла. При автоматической балансировке процесс создания третьей реплики может потребовать недопустимо много времени, поэтому существует небольшая вероятность потери второй реплики до замены узла.

Количество узлов, задействованных в операции, может зависеть от вида операции. При записи одни типы обновлений могут требовать наличия кворума, а другие нет. Это зависит от того, насколько высокую степень согласованности и доступности мы требуем от системы. Аналогично, если необходимо обеспечить быстрое чтение и допускается определенное устаревание данных, количество узлов может быть небольшим.

Часто требуется учесть оба фактора. Если требуется быстрое, строго согласованное чтение, то можно потребовать подтверждение записей от всех узлов, разрешая чтение только с одного узла (`N = 3, W = 3, R = 1`). Это может замедлить процесс записи, поскольку при этом необходимо установить контакт со всеми тремя узлами, а вы не можете допустить потери ни одного узла. Однако в некоторых ситуациях возможен компромисс.

Все дело в том, что у вас есть широкий выбор решений, которые допускают разные сочетания преимуществ и недостатков. Некоторые специалисты по технологии NoSQL пишут о простом компромиссе между согласованностью и доступностью; мы надеемся, что теперь вы понимаете, что этот компромисс на самом деле более гибкий и сложный.

5.6. Рекомендации по дальнейшему чтению

В Интернете существует множество интересных блогов и статей о согласованности данных в распределенных системах, но самым полезным источником мы считаем работу [Tanenbaum and Van Steen]. Это прекрасная работа об основах распределенных систем, которая позволит вам углубить свои знания.

Когда мы заканчивали работу над книгой, вышел специальный номер журнала *IEEE Computer* [IEEE Computer Feb 2012], посвященный возрастающему влиянию теоремы CAP. Это очень полезный источник информации по данной теме.

5.7. Резюме

- Конфликты "запись–запись" возникают, когда два клиента пытаются записать одни и те же данные в одно и то же время. Конфликты "чтение–запись" возникают, когда один клиент читает несогласованные данные посреди процесса записи, выполняемого другим клиентом.
- Пессимистические подходы блокируют запись данных, чтобы предотвратить конфликты. Оптимистические подходы обнаруживают конфликты и устраняют их.
- Конфликты чтения–записи в распределенных системах возникают, когда некоторые узлы получают обновленные данные, а другие нет. Итоговая

согласованность означает, что определенная часть системы станет согласованной, как только все записи будут распространены по всем узлам.

- Клиенты обычно стремятся к согласованности чтения–записи. Это значит, что клиент может записывать, а затем немедленно читать новое значение. Это может оказаться трудным, если чтение и запись выполняются на разных узлах.
- Чтобы обеспечить хорошую согласованность данных, в ходе выполнения операций над данными необходимо использовать много узлов, но это увеличивает время реакции. В результате часто возникает необходимость достичь компромисс между согласованностью и временем реакции.
- Теорема CAP утверждает, что при разделении сети вы можете достичь компромисса между доступностью и согласованностью данных.
- Долговечность также можно согласовывать со временем реакции, особенно если необходимо обеспечить восстановление реплицированных данных после сбоев.
- Для того чтобы обеспечить строгую согласованность при репликации, не обязательно устанавливать контакт со всеми репликантами; достаточно обеспечить большой кворум.

Глава 6

Штампы версий

Многие критики технологии NoSQL подчеркивают отсутствие поддержки транзакций. Транзакции — это полезный инструмент, помогающий программистам обеспечивать согласованность данных. Одной из причин, по которым сторонники технологии NoSQL не слишком беспокоятся об отсутствии транзакций, является то, что агрегатно-ориентированные базы данных NoSQL поддерживают атомарные обновления в агрегате. Агрегаты проектируются так, чтобы их данные образовывали естественную единицу обновления. Вместе с тем следует признать, что при выборе баз данных следует учитывать потребность в транзакциях.

В частности, важно помнить, что транзакции имеют ограничения. Даже в транзакционной системе мы сталкиваемся с обновлениями, которые требуют вмешательства человека и обычно не могут быть выполнены в транзакциях, поскольку в этом случае транзакция слишком долго оставалась бы открытой. Эту проблему можно решить с помощью *штампов версий* (version stamps), которые могут быть полезными и в других ситуациях, особенно за пределами односерверной модели распределения данных.

6.1. Коммерческие и системные транзакции

Необходимость поддерживать согласованность обновлений без транзакций является общим свойством систем, даже если они были созданы на основе транзакционных баз данных. Когда пользователи размышляют о транзакциях, они обычно имеют в виду *коммерческие транзакции* (business transactions). Коммерческая транзакция может представлять собой просмотр каталога, выбор бутылки виски по приемлемой цене, заполнение информации о кредитной карточке и подтверждение заказа. Все это обычно не должно происходить в рамках системных транзакций, предоставляемых базой данных, потому что в этом случае пришлось бы блокировать элементы базы данных на время, пока пользователь будет искать свою кредитную карточку и, не найдя, пойдет обедать с коллегами.

Обычно приложения начинают системную транзакцию только в конце сеанса взаимодействия с пользователем, чтобы блокировка возникала на короткое время. Однако проблема заключается в том, что вычисления и решения могут приниматься на основе уже измененных данных. В прейскуранте могла измениться цена виски, кто-то мог изменить адрес клиента, а компания могла изменить стоимость доставки.

Для решения таких проблем были разработаны многочисленные методы *автономной параллельности* (offline concurrency) [Fowler PoEAA], которые оказались полезными и для технологии NoSQL. Особенно полезным оказался подход под названием *оптимистическая автономная блокировка* (Optimistic Offline Lock) [Fowler PoEAA] — форма условного обновления, при которой операция клиента считывает любую информацию, от которой зависит коммерческая транзакция, и проверяет, не была ли она изменена после того, как пользователь ее прочитал и вывел на экран. Для этого удобно сделать так, чтобы записи в базе данных содержали нечто вроде *штампа версии*: поля, которое изменяется при каждом обновлении данных, хранящихся в записи. Считывая данные, вы получаете сообщение о штампе версии, так что при записи данных можно проверить, изменилась ли версия.

Этот метод применяется при обновлении ресурсов в протоколе HTTP [HTTP]. Для этого используется объектная метка `etag`. Как только вы получите ресурс, сервер изменяет метку `etag` в заголовке. Данная метка является скрытой строкой, обозначающей версию ресурса. Если впоследствии вы измените этот ресурс, то сможете использовать условное обновление, предъявив метку `etag`, полученную от последней выполненной операции GET. Если ресурс на сервере изменился, то метки `etag` не будут совпадать и сервер откажется выполнять обновление, вернув сообщение 412 (Precondition Failed).

В некоторых базах данных реализован аналогичный механизм условного обновления, позволяющий гарантировать, что изменения не будут основываться на устаревших данных. Вы можете сделать это самостоятельно, но в таком случае будете вынуждены прекратить все другие потоки выполнения, использующие данный ресурс, пока не прочитаете и не обновите данные. (Иногда эту операцию называют *"сравнить и установить"* (compare-and-set — CAS), по аналогии с операциями CAS, выполняемыми процессорами. Разница заключается в том, что процессор CAS сравнивает значение перед тем, как его установить, а механизм условного обновления в базах данных сравнивает штампы версий этого значения.)

Существует несколько разных способов для создания собственного штампа версий. Для этого можно использовать счетчик, который постоянно изменяется при обновлении ресурса. Счетчик полезен тем, что с его помощью легко выяснить, какая из версий является более новой. С другой стороны, в этом случае сервер должен генерировать значение счетчика и можно использовать только один ведущий узел, чтобы счетчики не дублировались.

Другой подход подразумевает создание идентификатора GUID — большого случайного числа, которое гарантированно является уникальным. Это может быть комбинация дат, информация об аппаратном обеспечении и другая случайная информация. Преимущество идентификаторов GUID состоит в том, что их может генерировать кто угодно и никто не может продублировать; недостаток заключается в том, что эти идентификаторы слишком большие и не позволяют выяснить, какая из версий является более новой.

Третий подход основан на хешировании содержимого ресурса. При достаточно большом размере ключа хеширования содержимое можно сделать уникальным, аналогично идентификатору GUID, и легко сгенерировать; преимущество этого подхода заключается в его детерминированности — любой узел генерирует одинаковое

хешированное содержание одного и того же источника данных. Однако, как и идентификаторы GUID, эту информацию невозможно сравнивать, а ее представление может оказаться слишком длинным.

Четвертый подход — использование меток времени последнего обновления. Как и счетчики, эти метки имеют относительно небольшой размер и допускают непосредственное сравнение, в то же время не ограничивая базу данных единственным ведущим узлом. Метки времени могут генерировать несколько компьютеров, но для правильной работы их необходимо синхронизировать. Один узел с неправильными часами может вызвать повреждение данных. Это опасно и в ситуациях, когда метки времени являются слишком точными, в этом случае могут возникнуть дубликаты. Нельзя использовать метки времени с точностью до миллисекунд, если в одну миллисекунду в вашей базе происходит много обновлений.

Описанные выше подходы можно сочетать друг с другом, создавая разные схемы для создания смешанных штампов. Например, в базе данных CouchDB используется сочетание счетчика и хеширования содержимого. Это позволяет во многих случаях сравнивать штампы даже при использовании одноранговой репликации. Если два одноранговых узла одновременно выполнят обновление, то сочетание счетчика и хеширования разного содержимого легко локализует конфликт.

Кроме устранения конфликтов, связанных с обновлениями, штампы версий полезны для обеспечения сеансовой согласованности.

6.2. Штампы версий на нескольких узлах

Типичный штамп версий хорошо работает, если вы имеете один авторитетный источник данных, например один сервер или репликацию "ведущий–ведомый". В этом случае штамп времени контролируется ведущим узлом. Все ведомые узлы просто следят за штампами. Однако эту систему следует адаптировать для одноранговой модели распределения, в которой нет единственного места для формирования штампа версий.

Если вы ищете на двух узлах одни и те же данные, то можете получить разные ответы. В этом случае ваша реакция будет зависеть от причины таких отличий. Возможно, обновление достигло одного узла, не дошло до второго. В этом случае можно принять позднейшую версию данных (если вы можете ее определить). Возможно, произошли несогласованные обновления. В этом случае вы должны решить, как с ними поступать. В подобной ситуации простой идентификатор GUID или метка `etag` не помогут, поскольку они не предоставляют информацию об отношениях между обновлениями.

Простейшей формой штампа версии является счетчик. Каждый раз, когда узел обновляет данные, счетчик увеличивается на единицу, а его значение заносится в штамп времени. Если у вас есть зеленая и синяя реплики единственного ведущего узла, то, получив ответы от синего узла с штампом версии, равным 4, и от зеленого с штампом версии, равным 6, вы знаете, что зеленый узел выдает более свежую информацию.

При работе с несколькими ведущими узлами ситуация усложняется. Один способ, основанный на распределенных системах управления версиями, предусматривает, что

все узлы должны содержать историю штампов версий. В этом случае вы можете обнаружить, что синий узел содержит более старую информацию по сравнению с зеленым. В этом случае клиенты или серверные узлы обязаны хранить истории штампов версий и указывать ее при поиске данных. Этот способ позволяет выявлять несогласованность данных, которая может возникнуть, если вы получаете два штампа версии и ни один из них не указан в истории другого. Несмотря на то что такие истории хранятся в системах управления версиями, они не используются в базах данных NoSQL.

Простой, но проблематичный подход основан на использовании меток времени. Основная проблема здесь заключается в том, что обычно трудно синхронизировать все узлы, особенно если обновления происходят часто. Как только часы на узле теряют синхронизацию, возникают разнообразные неприятности. Кроме того, метки времени не позволяют обнаруживать конфликты "запись–запись", поэтому хорошо работают только при одном ведущем узле, а в таком случае лучше использовать счетчик.

Наиболее распространенный подход, используемый в одноранговых системах NoSQL, основан на особой форме штампа версии, которая называется *вектором штампов* (vector stamp). По существу, вектор штампов — это набор счетчиков, по одному на каждый узел. Вектор штампов для трех узлов (синий, зеленый, черный) может выглядеть примерно так: `[blue: 43, green: 54, black: 12]`. Каждый раз, когда на узле происходит внутреннее обновление, его счетчик изменяется, поэтому обновление на зеленом узле может изменить вектор на `[blue: 43, green: 55, black: 12]`. При обмене данными два узла синхронизируют свои векторные штампы. Существует несколько способов синхронизации таких штампов. Мы выбрали в качестве общего названия термин "вектор штампов", хотя встречали и "вектор часов", и "вектор версий" — это особые формы вектора штампов, отличающиеся способами синхронизации.

С помощью такой схемы можно определить, какой из векторов штампов является более новым, поскольку в таком векторе все счетчики будут больше или равны соответствующим счетчикам в старом векторе штампов. Например, вектор `[blue: 1, green: 2, black: 5]` является более новым, чем `[blue: 1, green: 1, black 5]`, потому что в нем один счетчик больше соответствующего счетчика в другом векторе. Если в обоих векторах есть счетчики, которые больше соответствующего счетчика в другом векторе, например `[blue: 1, green: 2, black: 5]` и `[blue: 2, green: 1, black: 5]`, то возникает конфликт "запись–запись".

В векторе могут быть пропущенные значения. В этом случае пропущенное значение интерпретируется как 0. Итак, вектор `[blue: 6, black: 2]` следует интерпретировать как `[blue: 6, green: 0, black: 2]`. Это позволяет легко добавлять новые узлы без повреждения существующих векторов штампов.

Векторы штампов — отличное средство, позволяющее локализовать несогласованность данных, но не устраняет их. Разрешение любого конфликта зависит от предметной области, в которой вы работаете. Это часть компромисса между согласованностью и скоростью реакции. Вы либо должны смириться с фактом, что разделение сети может сделать вашу систему недоступной, либо выявить и устранить несогласованности данных.

6.3. Резюме

- Штампы версий позволяют выявить конфликты при параллельной работе. Когда вы читаете данные, а затем обновляете их, вы можете проверять вектор штампов, чтобы убедиться, что никто не обновил данные между вашим чтением и записью.
- Штампы версий можно реализовать с помощью счетчиков, идентификаторов GUID, хеширования содержимого, меток времени или сочетания этих методов.
- В распределенных системах вектор штампов версий позволяет обнаружить, когда на разных узлах возникают конфликтующие обновления.

Глава 7

Отображение–свертка

Появление агрегатно-ориентированных баз данных в большой степени обусловлено ростом количества кластеров. Работа на кластере подразумевает поиск оптимального способа хранения, отличающегося от способа их хранения на одном компьютере. Кластеры не просто изменили правила хранения данных — они также изменили правила вычислений. Если вы храните много данных на кластере, то их эффективная обработка требует другой организации. При централизованном хранении данных существуют два основных способа обработки данных: либо на самом сервере базы данных, либо на клиентском компьютере. Обработка на клиентском компьютере обеспечивает более высокую гибкость при выборе программного окружения, облегчая создание и модификацию программ. Однако при этом приходится перетаскивать данные с сервера. Если нужно обработать много данных, целесообразно перенести их обработку на сервер, пожертвовав удобством программирования и увеличив нагрузку на сервер.

Кластер сразу дает массу выгод — много компьютеров для распределения вычислений между ними. Тем не менее и в этом случае необходимо минимизировать объем данных, передаваемых по сети, стараясь выполнять бо́льшую часть обработки данных на том узле, который в них нуждается. Шаблон проектирования *Map-Reduce* (отображение–свертка), представляющий собой разновидность шаблона проектирования *Scatter-Gather* (*рассылка–сборка*) [Hohpe and Woolf]), — это способ организации обработки данных, позволяющий выполнить вычисления на многих компьютерах кластера и обеспечить хранение и обработку большей части данных на одном и том же компьютере. Впервые он был с успехом реализован в каркасе MapReduce компании Google [Dean and Ghemawat]. Его популярная реализация с открытым исходным кодом является частью проекта Hadoop, хотя некоторые базы данных содержат свои собственные реализации этого шаблона. Как и у большинства шаблонов проектирования, детали реализаций этого шаблона отличаются друг от друга, поэтому мы сосредоточимся на его основной идее. Название *Map-Reduce* означает, что в основе шаблона лежат операции отображения и свертки над коллекциями в функциональных языках программирования.

7.1. Основы шаблона Map-Reduce

Для объяснения основной идеи начнем с хорошо изученного примера — клиентов и заказов. Допустим, в качестве агрегата выбраны заказы, причем в каждом заказе есть товарные позиции. Каждая товарная позиция имеет идентификатор товара, его количество и стоимость. Этот агрегат кажется вполне логичным, поскольку люди, как правило, хотят видеть весь заказ сразу, а не по частям. У нас много заказов, поэтому мы фрагментировали базу данных по нескольким компьютерам.

Однако специалисты, анализирующие продажи, хотят видеть товар и прибыль, которую он принес за последнюю неделю. Этот отчет не соответствует имеющейся агрегатной структуре — в этом и заключается недостаток агрегатов. Для того чтобы получить отчет о прибылях от продажи товаров, вы должны обратиться к каждому компьютеру кластера и проверить много записей на каждом компьютере.

Именно эта ситуация требует применения отображения–свертки. На первом этапе отображения–свертки выполняется отображение — функция, получающая в качестве аргумента отдельный агрегат и порождающая набор пар "ключ–значение". В данном случае аргументом является заказ, а результатами — пары "ключ–значение", соответствующие товарным позициям. Каждая пара будет содержать идентификатор в качестве ключа и вложенное отображение с количеством и ценами в качестве значений (рис. 7.1).

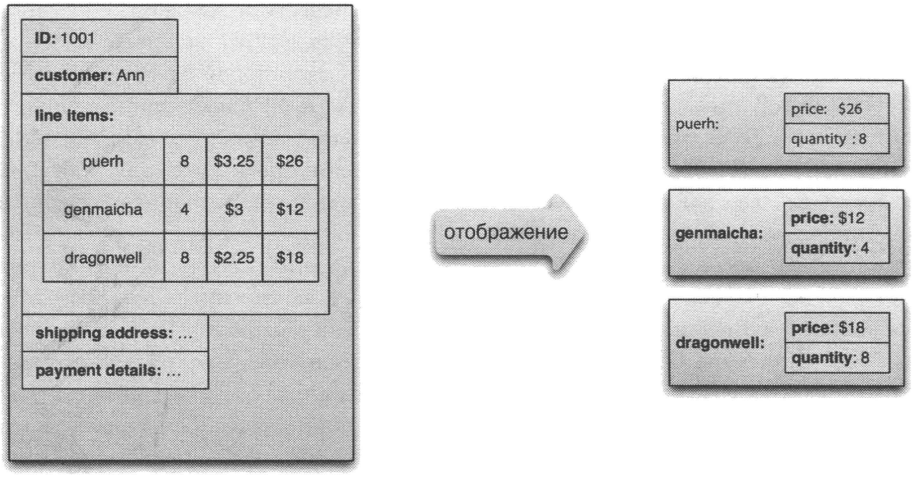


Рис. 7.1. Функция отображения считывает записи из базы данных и порождает пары "ключ–значение"

Каждое применение функции отображения не зависит от остальных. Это позволяет безопасно выполнять эти функции параллельно, так что каркас отображения–сверток может создать эффективные задания на отображение на каждом узле и свободно передавать заказ этим заданиям. Это обеспечивает высокую степень параллелизма и локальности доступа. В нашем случае мы просто извлекаем значение из заказа, но функция отображения может быть более сложной — это зависит от важности данных в агрегате.

Операция отображения действует только на одну запись; функция свертки получает несколько результатов, возвращенных операцией отображения и имеющих одинаковые ключи, и объединяет их значения. Итак, функция отображения может выдать 1000 товарных позиций из заказов книги "Рефакторинг баз данных"; а функция свертки — объединить их в одну запись, содержащую количество и прибыль. В то время как функция отображения может действовать на данные, принадлежащие только одному агрегату, функция свертки может использовать все значения, выданные для одного ключа (рис. 7.2).

Рис. 7.2. Функция свертки получает несколько пар "ключ–значение" с одним и тем же ключом и объединяет их в одно целое

Каркас отображения–свертки организует задачи отображения так, чтобы их выполняли соответствующие узлы, а данные передавались функции свертки. Чтобы упростить создание функции свертки, каркас собирает все значения для отдельной пары и вызывает функцию свертки один раз с заданным ключом и коллекцией всех значений, соответствующих этому ключу. Итак, для того чтобы выполнить задание отображения–свертки, необходимо написать только две функции.

7.2. Разделение и объединение

Простейшее задание отображения–свертки может иметь одну функцию свертки. Результаты работы всех операций отображения, выполненных на разных узлах, объединяются вместе и посылаются на свертку. Несмотря на что такая схема вполне работоспособна, можно повысить ее параллелизм и уменьшить объем передаваемых данных (рис. 7.3).

Сначала повысим степень параллелизма, разделив результаты работы отображений. Каждая функция свертки применяется к результатам с одним ключом. Это — ограничение, означающее, что вы не можете выполнить свертку с несколькими ключами. Одновременно это является преимуществом, потому что оно позволяет параллельно выполнять несколько сверток. Для того чтобы воспользоваться этим преимуществом, результаты отображений на каждом из обрабатывающих узлов разделяются по ключам. Обычно в один раздел группируются несколько ключей. Затем каркас получает данные от всех узлов одного раздела, объединяет их в одну группу combines и посылает

функции свертки. Несколько функций свертки могут обрабатывать разделы параллельно, объединяя окончательные результаты в одно целое. (Этот шаг называется *"тасованием"* ("shuffling"), а разделы иногда называют *"корзинами"* ("bucket") или *"областями"* ("regions").)

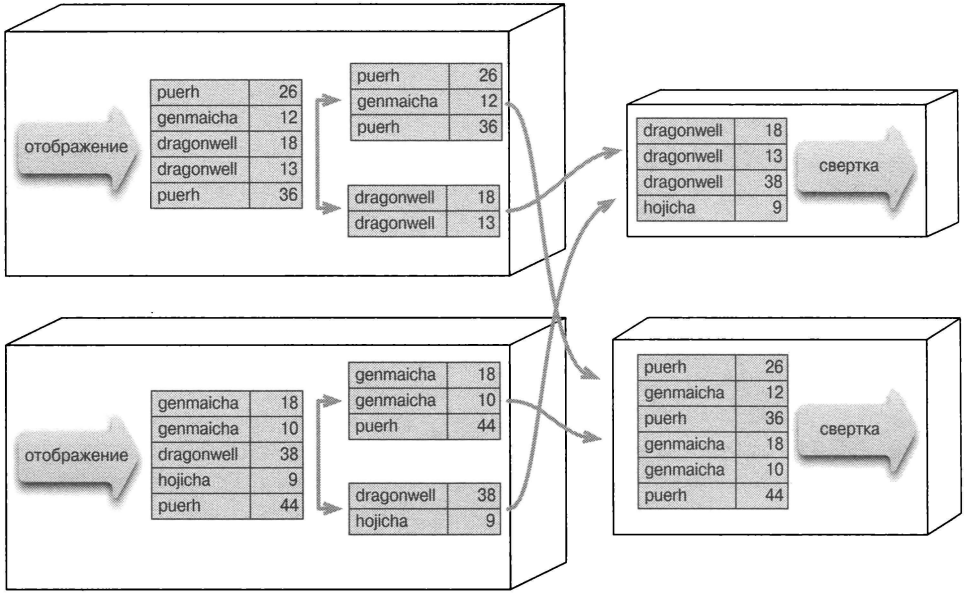


Рис. 7.3. Разделение позволяет свернуть функции, чтобы параллельно обрабатывать разные ключи

Следующая задача заключается в уменьшении объема данных, передаваемых между узлами, а также между функциями отображения и свертки. Большинство из этих данных повторяются и состоят из многих пар "ключ–значение" с одним и тем же ключом. Функция объединения выполняет усечение этих данных, объединяя все данные с одним и тем же ключом в одно значение (рис. 7.4). Функция объединения по существу является функцией свертки, — действительно, во многих случаях функцию объединения часто можно использовать для получения окончательной свертки. Для этого функция свертки должна иметь специальный вид: ее результат должен совпадать с аргументом. Такая функция называется *сверткой, допускающей объединение* (combinable reducer).

Но не все свертки допускают объединение. Рассмотрим функцию, подсчитывающую количество уникальных клиентов для конкретного товара. Функция отображения для такой операции должна возвращать товар и клиента. Затем свертка может объединить их и подсчитать, сколько раз каждый клиент заказывал конкретный товар, выдавая в качестве результата товар и счетчик (рис. 7.5). Но этот результат работы свертки отличается от входной информации, поэтому такая функция не может быть сверткой, допускающей объединение.

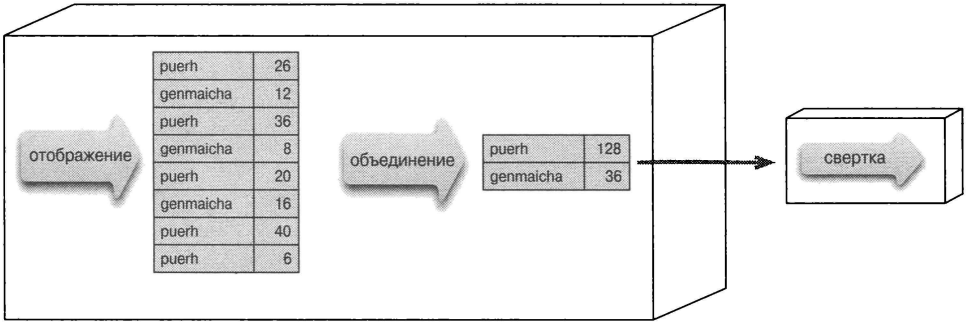


Рис. 7.4. Объединение сворачивает данные перед пересылкой по сети

В этой ситуации можно применить функцию объединения: она просто исключит дубликаты среди пар "ключ–значение", но будет отличаться от окончательной свертки.

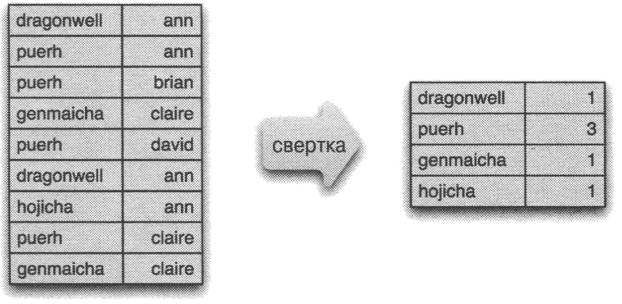


Рис. 7.5. Функция свертки, подсчитывающая количество уникальных клиентов, заказавших определенный сорт чая но не допускающая объединение

Имея свертки, допускающие объединение, каркас отображения–свертки может безопасно работать не только параллельно (чтобы свернуть разделы), но и последовательно, чтобы свернуть один раздел несколько раз в разных местах. Объединять данные можно не только перед передачей по сети, но и до завершения отображения. Это обеспечивает дополнительную гибкость каркаса отображения–свертки. Некоторые каркасы отображения–свертки требуют, чтобы все свертки допускали объединение, чтобы гибкость была максимальной. Если в одном из таких каркасов потребуется свертка, не допускающая объединения, необходимо выделить обработку данных в отдельный конвейер отображения–сверток.

7.3. Составные вычисления в схеме "отображение–свертка"

Подход "отображение–свертка" позволяет сохранить гибкость вычислений в относительно простой структуре параллельных вычислений на кластере. Поскольку параллельность и гибкость противоречат друг другу, существуют ограничения, накладывае-

мые на вычисления. Отображение можно применять только к отдельному агрегату, а свертку — только к отдельному ключу. Это значит, что следует подумать о структуре программ, работающих в таких условиях.

Одно из простых ограничений заключается в том, что вычисления должны состоять из вычислений, согласующихся с операцией свертки. Ярким примером является вычисление средних. Рассмотрим заказы, которые мы искали; допустим, мы знаем средний объем заказа для каждого товара. Важное свойство средних заключается в том, что они не компонуются, т.е. если взять две группы заказов, то нельзя объединить их отдельные средние значения. Вместо этого необходимо определить общий объем, подсчитать заказы по каждой группе, объединить их, а затем вычислить среднее значение по общей сумме и общему количеству (рис. 7.6).

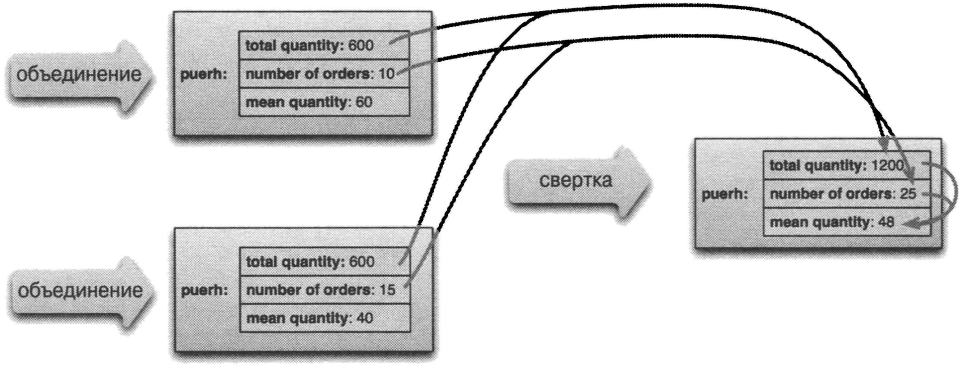


Рис. 7.6. При вычислении средних значений сумму и количество можно объединить в ходе свертки, но среднее значение необходимо вычислять на основе общей суммы и общего количества

Стремление к правильной свертке влияет на способ подсчета. Для того чтобы подсчитать количество товара, функция отображения создает поля счетчиков со значением 1, чтобы их можно было просуммировать (рис. 7.7).

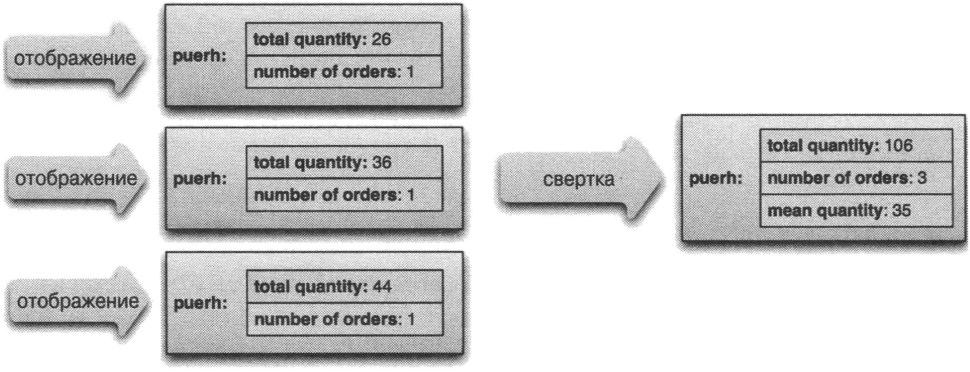


Рис. 7.7. При создании счетчика каждое отображение заносит в поле число 1, чтобы можно было подсчитать общее количество

7.3.1. Пример двухэтапной схемы "отображение–свертка"

При выполнении более сложных вычислений по схеме "отображение–свертка" полезно разделить их на этапы, используя подход *"каналы и фильтры"* (pipes-and-filters), в котором результат одного этапа становится входом для второго этапа, как на конвейерах в операционной системе UNIX.

Рассмотрим пример, в котором мы хотим сравнить объемы продаж товаров в каждом месяце 2011-го и предыдущего года. Для этого разобьем вычисления на два этапа. На первом этапе создадим записи, в которых будут храниться суммарные данные для конкретного товара, проданного в конкретном месяце. На втором этапе эти данные будут использованы как входная информация и будет вычислен результат для конкретного товара, который будет сравниваться с аналогичным результатом для того же самого товара в том же самом месяце предыдущего года (рис. 7.8).

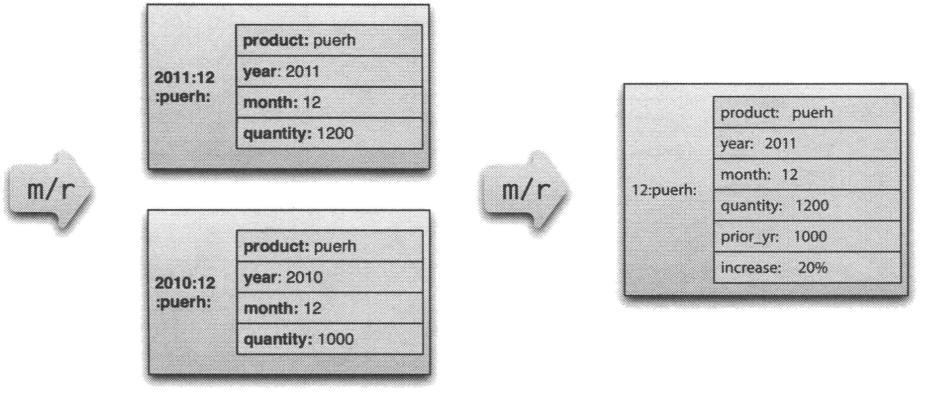


Рис. 7.8. Вычисление разбивается на два этапа отображения–свертки, которые будут проиллюстрированы на трех следующих рисунках

На первом этапе (рис. 7.9) выполняется чтение исходных записей о заказах и выдается ряд пар "ключ–значение" для продаж отдельного товара по месяцам.

Этот этап похож на схему "отображение–свертка", описанную ранее. Единственное новшество заключается в том, что здесь используется составной ключ, позволяющий свернуть записи по значениям нескольких полей.

Процесс двухэтапного отображения (рис. 7.10) обрабатывает эти результаты по годам. Запись, соответствующая 2011 году, заполняется текущим значением, а запись, соответствующая 2010 году, заполняется значением, полученным для прошлого года. Записи для прошлых лет (например, 2009 года) не влияют на результаты отображения.

Свертка в данном случае сводится к слиянию записей (рис. 7.11), в котором суммирование значений для двух разных лет позволяет свернуть результат в одно значение (соответственно вычисления, основанные на свернутых данных, учитывают один показатель).

Разделив отчет на несколько этапов отображения–свертки, мы упростили процедуру его создания. Как и во многих примерах трансформации, если каркас позволяет разделить процесс вычислений на этапы, то обычно проще реализовать поэтапную схему, состоящую из маленьких шагов, чем пытаться объединять всю логику работы в один этап.

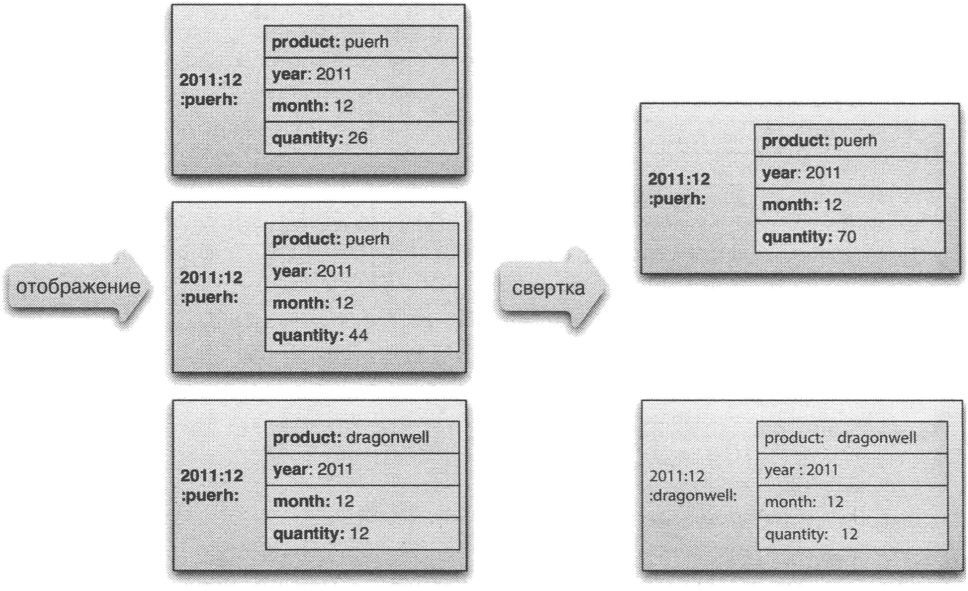


Рис. 7.9. Создание записей для ежемесячных продаж товара

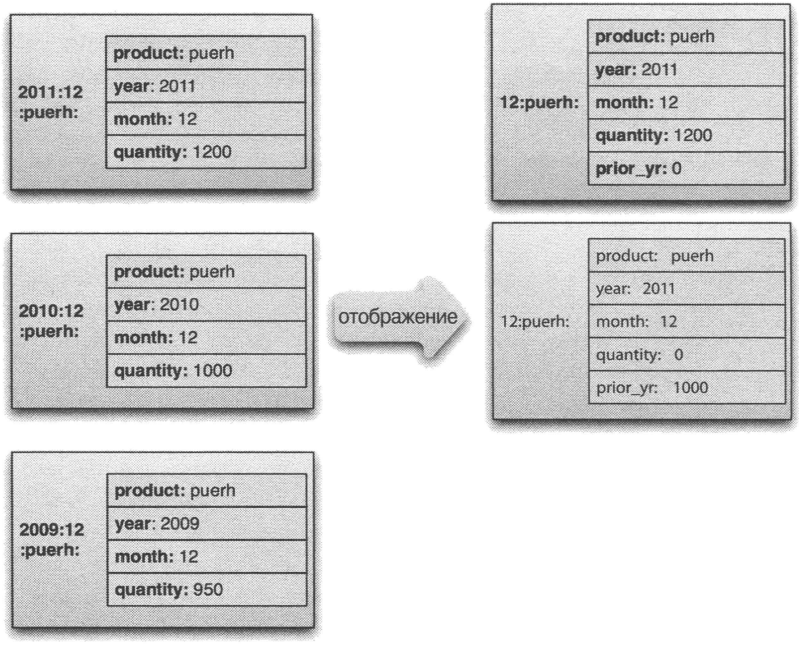


Рис. 7.10. На втором этапе отображения создается база записей для сравнения данных по годам

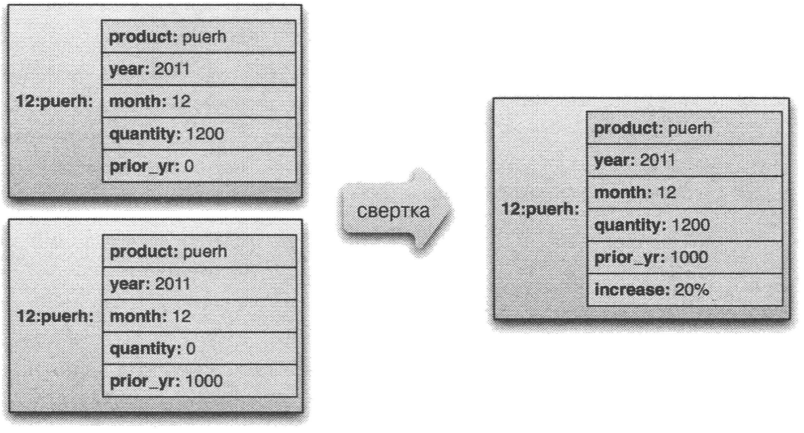


Рис. 7.11. На этапе свертки происходит объединение неполных записей

Другой подход заключается в том, что промежуточные результаты также могут оказаться полезными для повторного использования. Эта возможность важна тем, что она экономит время как для программирования, так и для выполнения программы. Промежуточные записи можно отправить в хранилище, формируя материализованное представление (см. раздел 3.4, "Материализованные представления").

Результаты отображения–свертки особенно полезно сохранить на ранних этапах, поскольку на этих этапах обычно выполняется бо́льшая часть операций, связанных с доступом к данным, поэтому, создав их один раз и используя в качестве основы для дальнейших операций, можно сэкономить большой объем работы. Однако, как и при любом повторном использовании, важно настроить эти данные на реальные запросы. Теоретические конструкции повторного использования редко учитывают это требование. Таким образом, важно учитывать влияние разнообразных запросов на вычисления в материализованных представлениях.

Шаблон *Map-Reduce* можно реализовать в любом языке программирования. Однако ограничения, связанные с требованиями хорошего стиля, требуют применения языков, специально разработанных для выполнения отображений–сверток. Язык Apache Pig [Pig] — ответвление проекта Hadoop — был специально разработан для упрощения процедуры разработки программ, выполняющих отображения–свертки. Определенно, много проще работать с каркасом Hadoop, чем с соответствующими библиотеками на языке Java. Другими словами, если вы хотите написать программу отображений–сверток на языке наподобие SQL, то используйте систему Hive [Hive] — другое ответвление проекта Hadoop.

Шаблон *Map-Reduce* необходимо знать даже независимо от контекста баз данных NoSQL. Исходная система отображения–свертки, разработанная компанией Google, оперировала файлами, хранящимися в распределенной файловой системе. Именно этот подход был принят в открытом проекте Hadoop. Несмотря на то что он предназначался для учета ограничений, связанных со структурированными вычислениями в рамках многоэтапных отображений–сверток, результатом стал метод вычислений, который естественным образом приспособлен для кластеров. При работе с большими

объемами данных необходимо ориентироваться на кластеры. Для таких вычислений хорошо подходят агрегатно-ориентированные базы данных. Мы считаем, что в ближайшие годы все больше организаций примут кластерно-ориентированный подход, а шаблон *Map-Reduce* будет использоваться еще шире.

7.3.2. Постепенное отображение-свертка

Пример, рассмотренный выше, иллюстрировал полные вычисления по схеме "отображение–свертка". Мы начинали с исходных данных и получили окончательный результат. Многие вычисления по схеме "отображение–свертка" требуют продолжительного времени, даже на кластерном оборудовании, причем в ходе их выполнения поступают новые данные, вынуждающие повторять вычисления заново, чтобы результаты не устаревали. Начинать вычисления каждый раз с самого начала может оказаться неприемлемо долгим процессом, поэтому часто вычисления по схеме "отображение–свертка" структурируют так, чтобы учесть постепенные обновления и свести к минимуму повторные вычисления.

Этапы отображения легко сделать постепенными — они повторяются только при изменении входных данных. Поскольку отображения изолированы друг от друга, последовательные обновления осуществляются просто.

Более сложную проблему представляет этап свертки, поскольку он объединяет результаты многих отображений и любое изменение одного из них вынуждает повторить свертку. Количество повторных вычислений можно уменьшить за счет параллельного выполнения свертки. При разделении данных для свертки необходимо помнить, что любой неизменный раздел не требует повторной свертки. Аналогично, объединение не нужно повторять, если соответствующие исходные данные не изменились.

Если свертка допускает объединение, то возникает еще больше возможностей для уменьшения объема вычислений. Если изменения являются аддитивными, т.е. мы можем только добавлять новые записи, но не изменять или удалять старые, то можно просто выполнить свертку существующих результатов и добавить новые. Если изменения носили деструктивный характер, т.е. представляли собой обновление и удаление, то повторного вычисления можно частично избежать, разбив свертку на этапы и повторяя вычисления только для этапов, исходные данные для которых изменились. По существу, это означает применение шаблона проектирования *Dependency Network* [Fowler DSL].

Каркасы отображения–свертки управляют многими из упомянутых аспектов, поэтому важно понимать, как конкретный каркас поддерживает постепенные операции.

7.4. Рекомендации для дальнейшего чтения

Если вы собираетесь применять вычисления по схеме "отображение–свертка", то сначала прочитайте документацию о конкретной базе данных, с которой будете работать. Каждая база данных имеет свой собственный подход, словарь и особенности. Все это необходимо знать. Кроме того, необходимо понимать, как структурировать

отображение–свертку, чтобы достичь максимальной технологичности и производительности. Мы не указываем конкретные книги по этой теме, но советуем начать с книг, посвященных проекту Hadoop. Хотя проект Hadoop не является базой данных, он представляет собой инструмент, интенсивно использующих парадигму "отображение–свертка", поэтому может оказаться полезным в других контекстах (в зависимости от того, насколько проект Hadoop отличается от ваших систем).

7.5. Резюме

- *Map-Reduce* — это шаблон проектирования, позволяющий распараллелить вычисления на кластере.
- Отображение считывает данные из агрегата и порождает соответствующие пары "ключ–значение". Отображение читают только по одной записи в каждый момент времени, поэтому их можно распараллелить и запустить на узле, хранящем эти записи.
- Свертки получают от отображений несколько значений, соответствующих одному и тому же узлу, и суммируют их, создавая единый результат. Каждая свертка работает с результатом, соответствующим одному ключу, поэтому их можно распараллелить по ключам.
- Свертки, у которых входная и выходная информация имеют одинаковый вид, можно объединить в конвейеры. Это повышает степень параллелизма и уменьшает объем передаваемых данных.
- Операции отображения–свертки можно объединять в конвейеры, в которых результат каждой свертки является входной информацией для следующего отображения.
- Если результат отображения свертки интенсивно используется, то его целесообразно хранить в виде материализованного представления.
- Материализованные представления можно обновлять с помощью постепенных операций отображения и свертки, которые вычисляют только изменения в представлении, а не вычисляют заново все данные.

Часть II

Реализация

Глава 8

Базы данных типа "ключ–значение"

Хранилище типа "ключ–значение" — это простая хеш-таблица, которая используется в основном тогда, когда весь доступ к базе данных осуществляется по первичному ключу. Эту таблицу можно представить как традиционную систему управления реляционной базой данных (Relational Data Base Management System — RDBMS) с двумя столбцами, например `ID` и NAME, где столбец `ID` является первичным ключом, а столбец NAME содержит значение. В системах RDBMS столбец NAME может хранить только данные типа `String`. Приложение может вычислить `ID` и VALUE и сохранить эту пару. Если ключ `ID` уже существует, то текущее значение замещается, в противном случае создается новая запись. Сравним терминологию, принятую в системах Oracle и Riak.

Oracle	Riak
база данных	кластер
таблица	сегмент
строка	ключ-значение
идентификатор строки	ключ

8.1. Что такое хранилище типа "ключ–значение"

Хранилище типа "ключ–значение" — простейшее хранилище данных NoSQL с точки зрения интерфейса прикладного программирования. Клиент может либо получить значение по ключу, либо записать значение по ключу, либо удалить ключ из хранилища данных. Значение — это двоичный объект данных, который записан в хранилище без детализации его внутренней структуры; что именно хранится в этом объекте, определяет приложение. Поскольку хранилища типа "ключ–значение" всегда используют доступ по первичному ключу, они обычно имеют высокую производительность и легко масштабируются.

К популярным базам данных типа "ключ–значение" относятся Riak [Riak], Redis (которую часто называют сервером Data Structure) [Redis], Memcached DB и ее версии [Memcached], Berkeley DB [Berkeley DB], HamsterDB (особенно для использования

в качестве встроенного хранилища) [HamsterDB], Amazon DynamoDB [Amazon's Dynamo] (закрытый исходный код) и Project Voldemort [Project Voldemort] (реализация базы Amazon DynamoDB с открытым кодом).

В некоторых хранилищах типа "ключ–значение", например в базе данных Redis, хранящиеся агрегаты не обязаны быть объектами предметной области — в их качестве могут использоваться любые структуры данных. База данных Redis поддерживает хранение объектов типа `lists`, `sets`, `hashes` и предусматривает операции вычисления диапазона, разности, объединения и пересечения. Эти свойства позволяют использовать базу данных Redis разнообразнее, чем стандартное хранилище типа "ключ–значение".

Мы перечислили далеко не все хранилища типа "ключ–значение", причем со временем появляются все новые базы данных. Исходя из удобства изложения, мы выбрали в качестве объекта изучения базу данных Riak. Она позволяет Riak хранить ключи в сегментах (buckets), представляющих собой некое подобие пространств имен, используемых для сегментирования ключей.

Если бы мы захотели хранить данные сеанса пользователя, информацию о его корзине товаров и предпочтениях в базе данных Riak, то могли бы просто записать их в один сегмент с одним ключом для всех перечисленных объектов. В рамках такого сценария мы получили бы один объект, содержащий все данные, и записали бы его в отдельный сегмент (рис. 8.1).

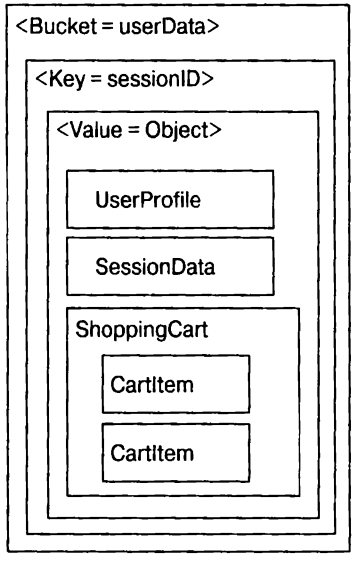


Рис. 8.1. Хранение всех данных в отдельном сегменте

Недостатком хранения всех объектов (агрегатов) в одном сегменте является тот факт, что агрегаты могут иметь разные типы, которые могут вызвать конфликты ключей. В качестве альтернативы к ключу можно было бы добавить имя объекта, например `288790b8a421_userProfile`, чтобы при необходимости можно было извлечь отдельный объект (рис. 8.2).

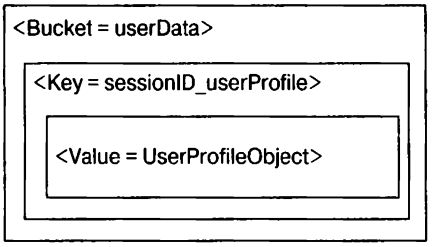


Рис. 8.2. Изменение структуры ключа для сегментирования данных в отдельном сегменте

Можно было бы также создать сегменты для хранения специфических данных. В базе данных Riak такие сегменты называются *предметными* (domain buckets) и допускают сериализацию и десериализацию с помощью клиентского драйвера.

```
Bucket bucket = client.fetchBucket(bucketName).execute();
DomainBucket<UserProfile> profileBucket =
DomainBucket.builder(bucket, UserProfile.class).build();
```

Использование предметных сегментов для хранения разных объектов (таких, как `UserProfile` и `ShoppingCart`) позволяет разделить данные между разными сегментами и считывать только необходимые объекты без изменения структуры ключа.

Хранилища типа "ключ–значение", такие как Redis, также поддерживают хранение произвольных структур данных, например множеств, хеш-таблиц, строк и т.д. Это свойство можно использовать для хранения списков объектов, таких как `states` или `addressTypes`, а также массивов данных о посещениях пользователя.

8.2. Функциональные возможности хранилищ типа "ключ–значение"

При работе с любыми хранилищами данных NoSQL необходимо понимать, чем они отличаются от стандартных баз данных RDBMS. Это позволит понять, каких свойств не достает и как изменить архитектуру приложения, чтобы лучше использовать особенности хранилищ типа "ключ–значение". Мы обсудим общие свойства всех хранилищ данных NoSQL: согласованность данных, транзакции, свойства запросов, структуру данных и масштабирование.

8.2.1. Согласованность данных

Согласованность данных относится только к операциям над отдельным ключом, поскольку эти операции могут получать, записывать или удалять данные по отдельному ключу. Можно реализовать оптимистические стратегии записи, но это было бы очень дорого, поскольку в этом случае хранилище не распознает изменение значения.

В распределенных хранилищах типа "ключ–значение", таких как Riak, реализована модель *итоговой согласованности данных*. Поскольку значение может быть уже реплицировано на других узлах, база данных Riak может устранить конфликты обновлений двумя способами: либо отдать предпочтение новой записи, удалив старую, либо вернуть оба значения клиенту, чтобы он сам устранил конфликт.

В базе данных эти возможности можно настроить при создании сегмента. Сегменты — это просто способ разделить ключи по пространствам имен, чтобы уменьшить вероятность конфликта между ними, — например, все ключи клиента могут храниться в сегменте `customer`. При создании сегмента можно использовать значения настроек согласованности данных, заданные по умолчанию, например, что запись считается правильной, только если она согласуется со всеми узлами, на которых хранятся ее данные.

```
Bucket bucket = connection
    .createBucket(bucketName)
    .withRetrier(attempts(3))
    .allowSiblings(siblingsAllowed)
    .nVal(numberOfReplicasOfTheData)
    .w(numberOfNodesToRespondToWrite)
    .r(numberOfNodesToRespondToRead)
    .execute();
```

Если требуется, чтобы данные во всех узлах были согласованными, можно в качестве флага `numberOfNodesToRespondToWrite` в атрибуте `w` взять флаг из атрибута `nVal`. Разумеется, это уменьшит производительность записи на кластере. Для того чтобы устранить конфликты записи или чтения, можно изменить флаг `allowSiblings` при создании сегмента и отдать предпочтение последней записи, не создавая конкурентов.

8.2.2. Транзакции

Разные хранилища типа "ключ–значение" используют разные спецификации транзакций. Вообще говоря, гарантий для записей не существует. Многие хранилища данных реализуют транзакции по-разному. База данных Riak использует концепцию кворума (см. раздел 5.5, "Кворумы"), реализованную с помощью значения `W` — коэффициента репликации, — при записи вызова интерфейса прикладного программирования.

Допустим, у нас есть кластер Riak с коэффициентом репликации, равным 5, и мы установили значение `W` равным 3. Запись считается корректной тогда и только тогда, когда она записана и признана успешной по крайней мере на трех узлах. Это обеспечивает толерантность к записи в базе данных Riak; в нашем случае при `N` равном 5 и `W` равном 3 кластер может допустить сбой операции записи на `N - W = 2` узлах, хотя некоторые данные на этих узлах невозможно будет прочитать.

8.2.3. Функциональные возможности запросов

Все хранилища типа "ключ–значение" могут выполнять запросы по ключу — вот и все. Если в запросе используется атрибут столбца значений, базу данных использовать невозможно: приложение должно считать значение, чтобы понять, удовлетворяет ли атрибут заданным условиям.

Запрос по ключу также имеет интересный побочный эффект. Что, если ключ неизвестен, особенно при нестандартных запросах во время отладки? Большинство хранилищ данных не раскрывают список всех первичных ключей; но даже если они сделают это, извлечение списка ключей и запрос по значению окажется слишком сложной процедурой. Некоторые базы данных типа "ключ–значение" решают эту проблему, предоставляя возможность поиска внутри значения, например, механизм **Riak Search**, позволяющий запрашивать данные так, будто вы используете индексы Lucene.

При использовании хранилищ типа "ключ–значение" большое внимание уделяется выбору структуры ключа. Можно ли генерировать ключ с помощью какого-нибудь алгоритма? Может ли пользователь предоставлять ключ (идентификатор пользователя, его электронный адрес и т.д.)? Следует ли выводить ключ из меток времени или других данных, не хранящихся в базе?

Благодаря этим характеристикам запросов базы типа "ключ–значение" часто используют для хранения данных о пользовательских сессиях (при этом в качестве ключа используется идентификатор сессии), корзин покупателей, профилей пользователей и т.п. С помощью свойства `expiry_secs` можно задать интервал времени, по истечении которого ключ станет недействительным, это особенно удобно для работы с сессионными объектами и корзинами покупателей.

```
Bucket bucket = getBucket(bucketName);
IRiakObject riakObject = bucket.store(key, value).execute();
```

При записи сегмента Riak с помощью функции `store` объект записывается по указанному ключу. Аналогично можно извлечь значение по указанному ключу с помощью функции `fetch`.

```
Bucket bucket = getBucket(bucketName);
IRiakObject riakObject = bucket.fetch(key).execute();
byte[] bytes = riakObject.getValue();
String value = new String(bytes);
```

База данных Riak имеет HTTP-ориентированный интерфейс, поэтому все операции можно выполнять с помощью веб-браузера или команды `curl` из командной строки. Вот как записываются эти данные в базу Riak:

```
{
"lastVisit":1324669989288,
"user":{
  "customerId":"91cfdf5bcb7c",
  "name":"buyer",
  "countryCode":"US",
  "tzOffset":0
  }
}
```

С помощью команды curl можно применить запрос POST к этим данным, сохранив данные в сегменте session вместе с ключом a7e618d9db25 (этот ключ следует передать вместе с данными):

```
curl -v -X POST -d '
{ "lastVisit":1324669989288,
  "user":{"customerId":"91cfdf5bcb7c",
  "name":"buyer",
  "countryCode":"US",
  "tzOffset":0}
}'
-H "Content-Type: application/json"
http://localhost:8098/buckets/session/keys/a7e618d9db25
```

Данные по ключу a7e618d9db25 можно извлечь с помощью команды curl:

```
curl -i http://localhost:8098/buckets/session/keys/a7e618d9db25
```

8.2.4. Структура данных

Для баз данных типа "ключ–значение" безразлично, что хранится в разделе значения в паре "ключ–значение". Значением может быть двоичный объект, текст, документ в формате JSON или XML и т.д. Чтобы указать конкретный тип в базе данных Riak, можно использовать заголовок Content-Type в запросе POST.

8.2.5. Масштабирование

Многие базы данных типа "ключ–значение" допускают масштабирование с помощью фрагментации (см. раздел 4.2 "Фрагментация"). При фрагментации значение ключа определяет, на каком узле хранится ключ. Допустим, мы выполняем фрагментацию по первому символу ключа; если ключ равен f4b19d79587d, то он начинается с буквы f и будет послан на узел, который отличается от узла, хранящего ключ ad9c7a396542. Такая настройка фрагментации может повысить производительность за счет добавления в кластер новых узлов.

Помимо этого, фрагментация создает несколько проблем. Если на узле, хранящем ключ, начинающийся на букву f, произойдет сбой, данные на этом узле станут недоступными, а новые данные с ключами, начинающимися с буквы f, будет невозможно записать.

Хранилища данных, такие как Riak, позволяют контролировать аспекты теоремы CAP (см. раздел 5.3.1 "Теорема CAP"): N (количество узлов для хранения реплик "ключ–значение"), R (количество узлов, которые должны подтвердить успешное извлечение данных, чтобы их чтение считалось корректным) и W (количество узлов, на которые необходимо сделать запись, чтобы она считалась успешной).

Допустим, у нас есть кластер Riak, состоящий из пяти узлов. Задав N равным 3, мы требуем, чтобы данные реплицировались по крайней мере на трех узлах; задав R

равным 2, мы требуем, чтобы какие-нибудь два узла ответили на запрос GET, чтобы он считался успешным; а задав W равным 2, мы гарантируем, что запрос PUT будет записан на двух узлах, прежде чем запись будет признана успешной.

Эти параметры позволяют осуществить тонкую настройку реакции на сбой узлов при чтении и записи. Исходя из своих требований, мы можем изменить эти значения, чтобы повысить доступность чтения или записи. Вообще говоря, выбор значения W зависит от требуемой степени согласованности; эти значения можно задавать по умолчанию во время создания сегмента.

8.3. Примеры использования

Обсудим некоторые ситуации, в которых хранилища типа "ключ–значение" зарекомендовали себя с лучшей стороны.

8.3.1. Хранение информации о сессии

Каждая веб-сессия является уникальной и имеет уникальный идентификатор sessionid. Приложения, записывающие идентификатор sessionid на диск или в базу данных RDBMS, могут извлечь немалую пользу из хранилищ типа "ключ–значение", поскольку вся информация о сессии может быть записана с помощью одного запроса PUT и получена с помощью одного запроса GET. Операция, состоящая из одного запроса, выполняется очень быстро, поскольку вся информация о сессии хранится в одном объекте. Во многих веб-приложениях используется хранилище Memcached. Если требуется обеспечить высокую доступность, можно использовать базу данных Riak.

8.3.2. Профили пользователей, предпочтения

Почти каждый пользователь имеет уникальный атрибут userId, username или какой-то другой идентификатор, а также предпочтения, например, язык, цвет, часовой пояс, выбранные товары и т.д. Все это можно поместить в один объект и получать предпочтения пользователя с помощью одной операции GET. Аналогично можно хранить профили товаров.

8.3.3. Корзины заказа

Коммерческие веб-сайты используют корзины заказа, связанные с пользователем. Если требуется, чтобы корзина заказа была доступна постоянно, независимо от браузеров, компьютеров и сессий, всю информацию о покупках можно поместить в объект value с ключом userid. Для таких ситуаций лучше всего подходит кластер Riak.

8.4. Когда хранилища типа "ключ–значение" использовать не следует

Существуют ситуации, в которых хранилища типа "ключ–значение" не являются оптимальным выбором.

8.4.1. Отношения между данными

Если между разными наборами данных необходимо установить отношения или поддерживать корреляцию между разными наборами ключей, хранилища типа "ключ–значение" являются неудачным выбором, даже несмотря на то, что некоторые из них обеспечивают возможности перехода по ссылкам.

8.4.2. Транзакции, состоящие из многих операций

Если при сохранении нескольких ключей при записи одного из них произошел сбой и вы хотите вернуться в исходное положение или выполнить откат остальных операций, хранилища типа "ключ–значение" не смогут вам помочь.

8.4.3. Запрос по данным

Хранилища типа "ключ–значение" плохо справляются с поиском ключей по соответствующим значениям. У них нет механизма для проверки значения на стороне базы данных, за некоторыми исключениями, например механизма Riak Search или механизмов индексирования Lucene [Lucene] и Solr [Solr].

8.4.4. Операции с множествами

Поскольку операции в каждый момент времени ограничены одним ключом, невозможно работать с несколькими ключами одновременно. Если требуется обработать несколько ключей сразу, это придется делать на клиентской стороне.

Глава 9

Документные базы данных

Основной концепцией в документных базах данных является документ. База данных хранит и извлекает документы в форматах XML, JSON, BSON и т.д.. Эти документы представляют собой самоописываемые иерархические древовидные структуры данных, которые могут состоять из ассоциативных массивов, коллекций и скалярных значений. Документы хранятся примерно одинаково, но не обязательно должны быть одинаковыми. Документные базы данных хранят документы в качестве значений в хранилищах типа "ключ–значение"; документные базы данных можно интерпретировать как хранилища типа "ключ–значение", в которых значение допускает проверку. Сравним терминологию, принятую в базах данных Oracle и MongoDB.

Oracle	MongoDB
экземпляр	экземпляр MongoDB
схема	база данных
таблица	коллекция
строка	документ
rowid	_id
join	DBRef

Поле `_id` — это специальное поле, которое можно найти в любом документе в базе данных Mongo. То же самое относится к псевдостолбцу `ROWID` в базе данных Oracle. В базе MongoDB поле `_id` может присваиваться пользователем, поскольку оно является уникальным.

9.1. Что такое документная база данных

```
{ "firstname": "Martin",
  "likes": [ "Biking",
  "Photography" ],
  "lastcity": "Boston",
  "lastVisited":
}
```

Представленный выше документ можно интерпретировать как строку в традиционной базе данных RDBMS. Рассмотрим другой документ.

```
{
  "firstname": "Pramod",
  "citiesvisited": [ "Chicago", "London", "Pune", "Bangalore" ],
  "addresses": [
    { "state": "AK",
      "city": "DILLINGHAM",
      "type": "R"
    },
    { "state": "MH",
      "city": "PUNE",
      "type": "R" }
    ],
   "lastcity": "Chicago"
}
```

Проанализировав эти документы, мы видим, что они похожи, но отличаются именами атрибутов. В документных базах данных это разрешается. Схема данных в разных документах может изменяться, но все эти документы принадлежат одной и той же коллекции — в отличие от базы данных RDBMS, в которой каждая строка в таблице должна иметь одну и ту же схему. Мы можем представить список `citiesvisited` как массив, а список `addresses` — как список документов, вложенных в главный документ.

Вложение дочерних документов как подобъектов в документах обеспечивает легкий доступ и более высокую производительность. Если посмотреть на эти документы, то вы увидите, что некоторые атрибуты у них одинаковые, например `firstname` или `city`. В то же время во втором документе есть атрибуты, которых нет в первом, например `addresses`, в то время как атрибут `likes` есть в первом документе, но его нет во втором.

Это представление данных не совпадает с представлением данных в базах RDBMS, в которых должен быть определен каждый столбец, и если он не содержит данных, то должен быть помечен как пустой или иметь значение `null`. В документах не бывает пустых атрибутов; если атрибут не найден, мы предполагаем, что он не был задан или не является релевантным для данного документа. Документы позволяют создавать новые атрибуты без определения или изменения существующих документов.

Среди популярных документных баз данных следует назвать MongoDB [MongoDB], CouchDB [CouchDB], Terrastore [Terrastore], OrientDB [OrientDB], RavenDB [RavenDB] и, конечно, хорошо известную и часто подвергаемую критике базу Lotus Notes [Notes Storage Facility], которая использует документное хранилище.

9.2. Функциональные возможности

Несмотря на большое количество специализированных документных баз данных, в качестве их представителя, имеющего типичные функциональные возможности, мы рассмотрим базу MongoDB. Следует помнить, что каждая документная база имеет свои функциональные возможности, которых может не быть у остальных.

Сначала попробуем разобраться в том, как работает документная база MongoDB. Каждый экземпляр MongoDB имеет несколько *баз данных*, и каждая база данных может иметь несколько *коллекций*. Если сравнить это с базами RDBMS, то мы увидим, что экземпляр RDBMS соответствует экземпляру MongoDB, схемы в базах RDBMS соответствуют базам данных MongoDB, а таблицы RDBMS — это коллекции в MongoDB. Когда мы сохраняем документ, мы должны выбрать, какой базе данных и коллекции он должен принадлежать, — например, выполнить операцию `database.collection.insert(document)`, которая обычно записывается как `db.coll.insert(document)`.

9.2.1. Согласованность данных

Согласованность в базах данных MongoDB настраивается с помощью *наборов реплик* (replica sets). При этом записи должны быть реплицированы на всех ведомых узлах или на заданном количестве ведомых узлов. Каждая запись должна задавать количество серверов, на которых должна тиражироваться запись, прежде чем ее возвращение будет считаться успешным.

Такая команда, как `db.runCommand({ getlasterror : 1 , w : "majority" })`, сообщает базе данных, насколько строгим должна быть согласованность данных. Например, если у вас один сервер и вы задали атрибут `w` равным `majority`, запись будет возвращена немедленно, поскольку существует только один узел. Если в набор реплик входят три узла, а атрибут `w` равен `majority`, то запись будет выполнена как минимум на двух узлах, прежде чем будет признана успешной. Значение атрибута `w` можно повысить, чтобы усилить согласованность, но при этом снизится производительность записи, поскольку теперь записи должны выполняться на большинстве узлов. Наборы реплик позволяют также повысить производительность чтения, допуская чтение с ведомых узлов путем установки параметра `slaveOk`; этот параметр можно установить как для соединения, так и для базы данных, коллекции или отдельной операции.

```
Mongo mongo = new Mongo("localhost:27017");
mongo.slaveOk();
```

В этом фрагменте мы устанавливаем параметр `slaveOk` для операции, поэтому можем решать, какие операции могут работать с данными, полученными с ведомого узла:

```
DBCollection collection = getOrderCollection();
BasicDBObject query = new BasicDBObject();
query.put("name", "Martin");
DBCursor cursor = collection.find(query).slaveOk();
```

Аналогично параметрам, регламентирующим согласованность чтения, по желанию можно изменять настройки, влияющие на строгость согласованности записей. По умолчанию запись считается успешной, как только база данных получит ее; это условие можно изменить и ждать, пока записи будут синхронизированы с диском или размножены на два или более узла. Для этого предназначен параметр `WriteConcern`. Для того чтобы определенные записи были записаны на ведущий узел и несколько ведомых узлов, следует установить параметр `WriteConcern` равным `REPLICAS_SAFE`. Ниже приведен код, в котором параметр `WriteConcern` задается для всех записей в коллекции.

```
DBCollection shopping = database.getCollection("shopping");
shopping.setWriteConcern(REPLICAS_SAFE);
```

Параметр `WriteConcern` можно задавать для отдельных операций, задав его в команде сохранения.

```
WriteResult result = shopping.insert(order, REPLICAS_SAFE);
```

Существует компромисс, который следует тщательно рассмотреть с учетом требований приложения и бизнеса: целесообразно ли установить параметр `slaveOk` для чтения и какой уровень безопасности надо установить для записи с помощью параметра `WriteConcern`.

9.2.2. Транзакции

Транзакции в традиционном реляционном смысле означают, что модификацию базы данных можно начинать с помощью команд `insert`, `update` и `delete`, примененных к разным таблицам, а затем решать, сохранить ли эти изменения или нет, используя команду `commit` или `rollback`. В базе данных NoSQL эти конструкции обычно не доступны — запись либо оказывается успешной, либо нет. Транзакции на уровне отдельного документа называются *атомарными* (atomic transactions). Транзакции, содержащие больше одной операции, как правило, невозможны, хотя существуют базы данных, такие как RavenDB, поддерживающие транзакции, охватывающие несколько операций.

По умолчанию все записи считаются успешными. Более тонкий контроль над записями можно получить с помощью параметра `WriteConcern`. Задав параметр `WriteConcern.REPLICAS_SAFE`, можно гарантировать, что объект `order` будет записан на несколько узлов и только потом его запись будет признана успешной. Выбирая разные значения параметра `WriteConcern`, можно выбрать уровень безопасности записей; например, записывая данные регистрационного журнала, можно использовать самый низкий уровень безопасности, `WriteConcern.NONE`:

```
final Mongo mongo = new Mongo(mongoURI);
mongo.setWriteConcern(REPLICAS_SAFE);
DBCollection shopping = mongo.getDB(orderDatabase)
                             .getCollection(shoppingCollection);
try {
```

```
   WriteResult result = shopping.insert(order, REPLICAS_SAFE);
//Запись на ведущий узел и хотя бы один ведомый
} catch (MongoException writeException) {
//Запись не выполнена как минимум на двух узлах, включая ведущий
   dealWithWriteFailure(order, writeException);
}
```

9.2.3. Доступность

Теорема CAP (см. раздел 5.3.1, "Теорема CAP") утверждает, что можно достичь только двух свойств из трех: согласованность данных, доступность и устойчивость к разделению. Документные базы данных пытаются повысить доступность, реплицируя данные с помощью конфигурации "ведущий–ведомый". Одни и те же данные доступны на разных узлах, и клиенты могут получить их, даже если ведущий узел выйдет из строя. Обычно код приложения не обязан выяснять, доступен ведущий узел или нет. База данных MongoDB реализует репликацию, обеспечивая высокий уровень доступности с помощью *наборов реплик* (replica sets).

Набор реплик состоит из двух или более узлов, участвующих в *асинхронной горизонтальной репликации* (master-slave replication). Узлы, входящий в набор реплик, выбирают ведущего, или главного, среди них. Несмотря на то что все узлы имеют одинаковые права голоса, некоторые узлы могут оказаться предпочтительнее за счет близости к остальным серверам, большей оперативной памяти и других преимуществ; пользователи могут повлиять на их выбор, назначая приоритет узла — число от 0 до 1000.

Все запросы поступают на ведущий узел, а данные реплицируются на ведомые узлы. Если ведущий узел выйдет из строя, остальные узлы в наборе реплик проводят голосование и выбирают в своей среде новый ведущий узел; все дальнейшие запросы направляются на этот новый ведущий узел, а ведомые узлы теперь будут получать данные от него. Если узел, ранее вышедший из строя, вернется к работе, то он будет подключен как ведомый и будет наверстывать упущенное с помощью остальных узлов, передающих ему недостающие данные.

На рис. 9.1 показана примерная конфигурация наборов реплик. В этой конфигурации предусмотрены два узла — **mongo A** и **mongo B**, база данных MongoDB, работающая в главном центре данных, и узел **mongo C** во вспомогательном центре данных. Если мы хотим, чтобы узлы в главном центре данных выбирались в качестве ведущих, то можем присвоить им более высокий приоритет по сравнению с остальными узлами. Добавлять узлы в наборы реплик можно без их отключения.

Приложения выполняют операции записи и чтения на ведущем узле. Установив соединение, приложение должно лишь связаться с одним узлом (неважно каким — ведущим или ведомым) из набор реплик, а остальные узлы будут обнаружены автоматически. Если ведущий узел выйдет из строя, то драйвер будет обмениваться данными с новым узлом, выбранным набором реплик. Приложение не обязано реагировать на сбои связи и критерии выбора узлов. Наборы реплик дают возможность обеспечить высокую доступность документных хранилищ.

Наборы реплик обычно используются для обеспечения избыточности данных, автоматического перехода при отказе, масштабирования чтения, бесперебойной работы

сервера и восстановления работы после аварии. Аналогичные возможности предоставляют базы данных CouchDB, RavenDB, Terrastore и др.

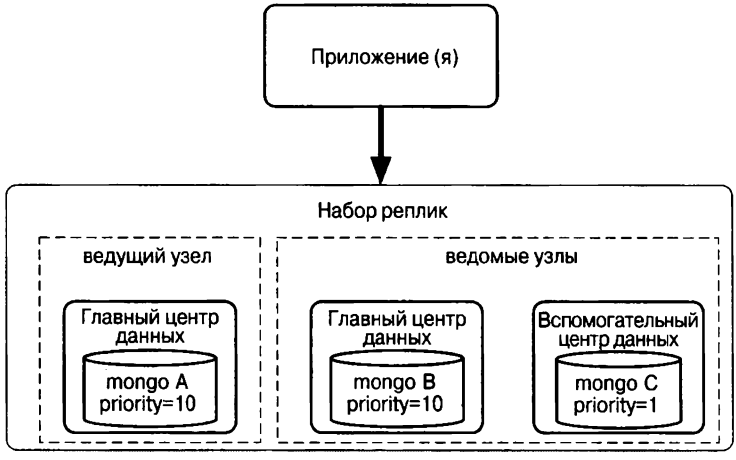


Рис. 9.1. Конфигурация набора реплик с высоким приоритетом, присвоенным узлам одного и того же центра данных

9.2.4. Функциональные возможности запросов

Документные базы данных обеспечивают разнообразные функциональные возможности запросов. База CouchDB позволяет осуществлять запросы через представления — сложные запросы к документам, которые могут быть материализованными (см. раздел 3.4, "Материализованные представления") или динамическими (представьте себе, что это представления RDBMS, которые могут быть как материализованными, так и не материализованными). Если в базе CouchDB вам нужен агрегат количества просмотров какого-то товара и их средний рейтинг, вы можете добавить представление, реализованное по схеме "отображение–свертка" (см. раздел 7.1, "Основы отображения–свертки"), чтобы получить количество просмотров и усреднить их рейтинги.

Если запросов слишком много, вычислять их количество и усреднять рейтинг каждого запроса нецелесообразно; вместо этого можно добавить материализованное представление, которое заранее вычислит эти значения и сохранит результаты в базе данных. Если данные с момента последнего обновления изменились, эти материализованные представления при запросе обновляются.

Одним из преимуществ документных баз данных по сравнению с хранилищами типа "ключ–значение" является то, что можно послать запрос к содержанию документа, не извлекая весь документ по его ключу, чтобы просмотреть его. Это свойство делает такие базы данных похожими на модель запроса RDBMS.

В базе MongoDB есть язык запросов, который выражается в формате JSON и имеет конструкции, такие как `$query` для выражения `where`, `$orderby` — для сортировки данных и `$explain` — для демонстрации плана выполнения запроса. Для создания запроса в базе MongoDB можно использовать многие другие конструкции, подобные этим.

Рассмотрим некоторые запросы к базе данных MongoDB. Допустим, мы хотим вернуть все документы из коллекции заказов (все строки из таблицы заказов). На языке SQL этот запрос выглядит так:

```
SELECT * FROM order
```

Эквивалентный запрос в оболочке базы Mongo выглядит следующим образом:

```
db.order.find()
```

Выбор заказов для конкретного идентификатора `customerId`, равного `883c2c5b4e5b`, можно записать так:

```
SELECT * FROM order WHERE customerId = "883c2c5b4e5b"
```

Эквивалентный запрос в базе Mongo на получение всех заказов для конкретного идентификатора `customerId`, равного `883c2c5b4e5b`, выглядит следующим образом:

```
db.order.find({"customerId":"883c2c5b4e5b"})
```

Аналогично выбрать записи `orderId` и `orderDate` для заданного клиента на языке SQL можно так:

```
SELECT orderId,orderDate FROM order WHERE customerId = "883c2c5b4e5b"
```

Эквивалент этого запроса в базе Mongo можно записать следующим образом:

```
db.order.find({customerId:"883c2c5b4e5b"},{orderId:1,orderDate:1})
```

Аналогично можно формировать запросы для подсчета, суммирования и выполнения других операций. Поскольку документы являются агрегированными объектами, можно легко сформировать запрос на документы с помощью полей с дочерними объектами. Допустим, мы хотим сформировать запрос на все заказы, в котором один из заказанных товаров имеет название `Refactoring`. На языке SQL это требование выглядело бы так:

```
SELECT * FROM customerOrder, orderItem, product
WHERE
customerOrder.orderId = orderItem.customerOrderId
AND orderItem.productId = product.productId
AND product.name LIKE '%Refactoring%'
```

Его эквивалент в базе Mongo записывается следующим образом:

```
db.orders.find({"items.product.name":/Refactoring/})
```

Запрос к базе MongoDB проще, потому что объекты вложены в отдельный документ и в запросе можно использовать вложенные дочерние документы.

9.2.5. Масштабирование

Идея масштабирования заключается в добавлении узлов или изменении хранилища данных без простого переноса базы данных в более крупный "ящик". Мы говорим не об изменениях приложения, направленных на обработку более крупного объема данных, а о функциональных возможностях базы данных, направленных на повышение рабочей нагрузки.

Масштабирование нагрузок, связанных с интенсивным чтением данных, можно обеспечить за счет увеличения количества вспомогательных узлов, выполняющих чтение и перенаправления всех операций чтения на эти узлы. В приложении, связанном с интенсивным чтением данных на кластере с набором реплик из трех узлов, можно расширить возможности чтения, добавив дополнительные вспомогательные узлы в набор реплик, выполняющих чтение с флагом `slaveOk` (рис. 9.2). Это горизонтальное масштабирование чтения.

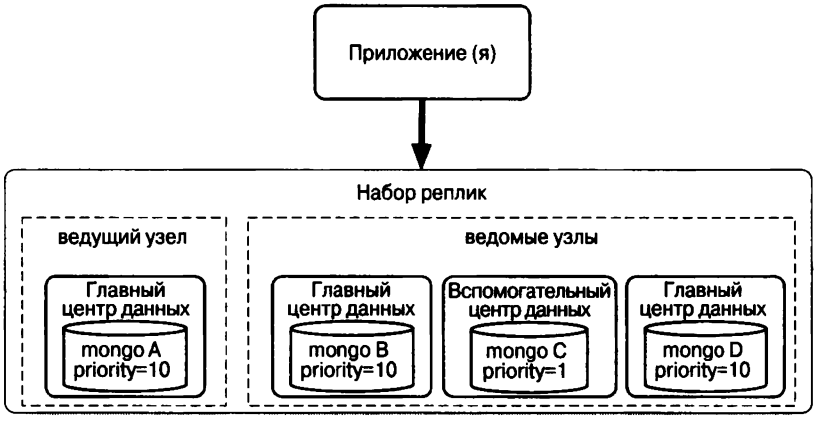


Рис. 9.2. Добавление нового узла mongo D в существующий кластер с набором реплик

После загрузки нового узла mongo D его необходимо добавить в набор реплик:

```
rs.add("mongod:27017");
```

После добавления нового узла он синхронизируется с существующими узлами, присоединяется к набору реплик в качестве вспомогательного узла и начинает выполнять запросы на чтение. Преимущество такой настройки заключается в том, что все остальные узлы не требуют перезагрузки, а приложение не простаивает.

При масштабировании для записи можно начать с фрагментации данных (см. раздел 4.2, "Фрагментация"). Фрагментация напоминает разделение в базах RDBMS, в которых данные разбиваются по значению в определенном столбце, например по штату или году. В базах RDBMS разделения обычно выполняются на одном и том же узле, поэтому приложение клиента должно посылать запрос не к конкретному разделу, а к таблице базы; база RDBMS сама найдет нужный раздел для запроса и вернет данные.

При фрагментации данные также разбиваются по определенному полю, но при этом они перемещаются на другой узел базы Mongo. Эти данные динамически

перемещаются между узлами, чтобы сбалансировать фрагменты. При горизонтальном масштабировании для записи можно добавить дополнительные узлы и увеличить количество узлов, выполняющих запись.

```
db.runCommand( { shardcollection : "ecommerce.customer",
key : {firstname : 1} } )
```

Разделение данных по фамилии клиента гарантирует сбалансированное распределение данных между фрагментами и оптимальную производительность записи; кроме того, каждый фрагмент может быть набором реплик, обеспечивающим более высокую производительность чтения в фрагменте (рис. 9.3). Добавляя новый фрагмент в существующий фрагментированный кластер, мы сбалансировано распределяем данные по четырем, а не трем фрагментам. В ходе перемещений данных и рефакторинга инфраструктуры приложение не испытывает простоев, хотя кластер не может функционировать оптимально, когда большой объем данных перемещается между фрагментами.

Ключ фрагмента играет важную роль. Если требуется разместить фрагменты базы MongoDB поближе к их пользователям, фрагментация, ориентированная на местоположение пользователя, может оказаться удачной идеей. При фрагментации по местоположению клиента все пользовательские данные на восточном побережье США хранятся во фрагментах, обслуживаемых на восточном побережье, а все пользовательские данные на западном побережье США хранятся во фрагментах, обслуживаемых на западном побережье.

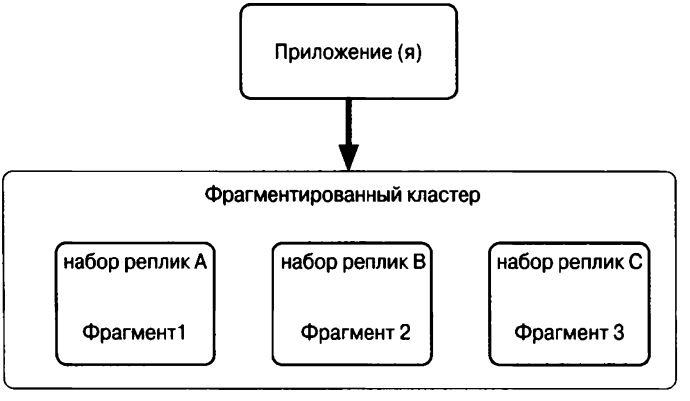


Рис. 9.3. Настройка фрагментированной базы MongoDB, в которой каждый фрагмент является набором реплик

9.3. Примеры использования

9.3.1. Регистрация событий

Приложения по-разному регистрируют события; на предприятиях существует множество разных приложений, желающих регистрировать события. Документные базы

данных могут хранить все эти типы событий и действовать как центральное хранилище событий. Это особенно важно в ситуациях, когда тип данных, собираемых событиями, постоянно изменяется. События могут фрагментироваться по имени приложения, породившего их, или по типу событий, например `order_processed` или `customer_logged`.

9.3.2. Системы управления информационным наполнением, блог-платформы

Поскольку документные базы данных не имеют предопределенной схемы и обычно понимают JSON-документы, они хорошо работают в системах управления информационным наполнением или в приложениях по публикации веб-сайтов, управляющих комментариями пользователей, их регистрацией, профилями, а также представлением документов в веб.

9.3.3. Веб-аналитика и аналитика в реальном времени

Документные базы данных могут хранить данные, необходимые для анализа в реальном времени; поскольку части документов можно обновлять, можно очень легко хранить представления страниц или информацию об отдельных посетителях, а также добавлять новые показатели эффективности без изменения схемы.

9.3.4. Приложения для электронной коммерции

Приложения для электронной коммерции часто должны иметь гибкую схему товаров и заказов, а также возможность изменять свои модели данных без дорогостоящего рефакторинга базы данных или миграции данных (см. раздел 12.3, "Изменения схем в хранилищах данных NoSQL").

9.4. Когда документные хранилища использовать не следует

Существуют ситуации, в которых документные хранилища оказываются не лучшим решением.

9.4.1. Сложные транзакции, охватывающие разные операции

Если вы хотите выполнять атомарные операции с несколькими документами, то документные базы данных для этого, как правило, не подходят. Однако существует несколько документных баз данных, поддерживающих такие операции, например RavenDB.

9.4.2. Запросы к изменяющейся агрегатной структуре

Гибкая схема означает, что база данных не накладывает на схему никаких ограничений. Данные сохраняются в виде сущностей приложения. Если возникает необходимость в специальном запросе к этим сущностям, ваши запросы можно изменять (в терминах баз данных RDBMS это означает, что, когда вы используете критерии, в которых объединяются несколько таблиц, эти таблицы сохраняют изменения). Поскольку данные сохраняются как агрегат, то при постоянном изменении структуры агрегата необходимо сохранять его с наименьшим уровнем детализации, т.е. фактически нормализовать данные. В таком сценарии документная база данных может оказаться неработоспособной.

Глава 10

Семейство столбцов

Хранилища, представляющие собой семейство столбцов, такие как Cassandra [Cassandra], HBase [Hbase], Hypertable [Hypertable] и Amazon SimpleDB [Amazon SimpleDB], позволяют хранить данные с ключами, отображаемыми в значения, и группировать значения в многочисленных семействах столбцов, каждое из которых является ассоциативным массивом данных.

RDBMS	Cassandra
экземпляр базы данных	кластер
база данных	пространство ключей
таблица семейств	столбец
строка	строка
столбец (один и тот же для всех строк)	столбец (может быть разным для разных строк)

10.1. Что такое семейство столбцов

Существует много баз данных, представляющих собой семейство столбцов. В данной главе речь пойдет о базе Cassandra, но будут упоминаться и другие базы данных на основе семейства столбцов, чтобы обсудить функциональные возможности, полезные в конкретных сценариях.

Семейство столбцов — это строка, содержащая множество столбцов, ассоциированных с ключом строки (рис. 10.1). Семейства столбцов группируют взаимосвязанные данные, доступ к которым часто обеспечивается как к единому целому. Например, в семействе столбцов, содержащем данные о клиентах, нас часто интересует информация об их профилях, а не о заказах.

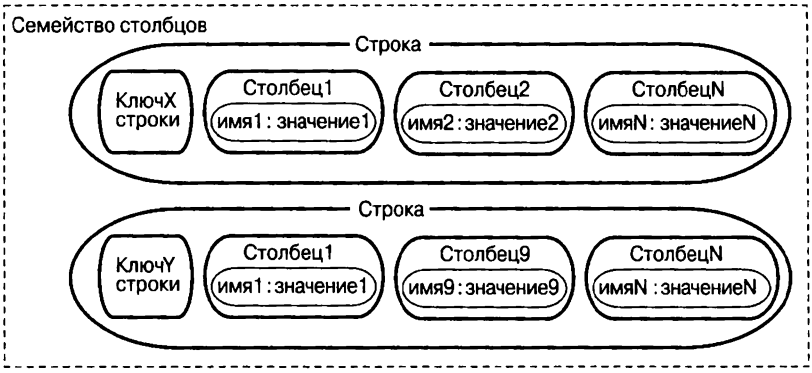


Рис. 10.1. Модель данных в базе Cassandra, представляющей собой семейство столбцов

Cassandra — одна их популярных баз данных, основанных на семействах столбцов, но существуют и другие базы такого типа — HBase, Hypertable и Amazon DynamoDB [Amazon DynamoDB]. Cassandra — это быстродействующая и легко масштабируемая база данных, в которой операции записи распределены по кластеру. Кластер не имеет ведущего узла, поэтому любую операцию чтения и записи можно выполнить на любом узле кластера.

10.2. Функциональные возможности

Для начала рассмотрим, как структурируются данные в базе Cassandra. Основной единицей хранения в базе Cassandra является столбец, состоящий из пары "имя–значение", в которой имя играет роль ключа. Каждая из пар "ключ–значение" всегда хранится с меткой времени, которая используется для того, чтобы задавать срок действия данных, разрешать конфликты записи, обрабатывать устаревшие данные и выполнять другие функции. Если данные столбца больше не используются, то это место можно восстановить позднее на этапе уплотнения.

```
{
  name: "fullName",
  value: "Martin Fowler",
  timestamp: 12345667890
}
```

Столбец `column` содержит ключ строки `fullName` и значение `Martin`, а также связанную с ними метку времени. Строка — это коллекция столбцов, ассоциированных с ключом; коллекция таких строк образует семейство столбцов. Если семейство столбцов состоит из обычных столбцов, оно называется *стандартным* (standard column family).

```
//Семейство столбцов
{
//row
  "pramod-sadalage" : {
```

```
    firstName: "Pramod",
    lastName: "Sadalage",
    lastVisit: "2012/12/12"
  }
//Строка
  "martin-fowler" : {
    firstName: "Martin",
    lastName: "Fowler",
    location: "Boston"
  }
}
```

Каждое семейство столбцов можно сравнить с контейнером строк в таблице RDBMS, в которой ключ идентифицирует строку, а строка состоит из множества столбцов. Отличие заключается в том, что разные строки не обязаны содержать одинаковые столбцы, причем столбцы могут добавляться в любую строку в любое время и добавлять их в остальные строки не обязательно. В нашем примере строки `pramod-sadalage` и `martin-fowler` имеют разные столбцы; обе эти строки являются частью семейства столбцов.

Если столбец состоит из ассоциативного массива столбцов, то возникает *суперстолбец* (super column), состоящий из имени и значения, которое является ассоциативным массивом столбцов. Суперстолбец можно интерпретировать как контейнер столбцов.

```
{
  name: "book:978-0767905923",
  value: {
    author: "Mitch Albon",
    title: "Tuesdays with Morrie",
    isbn: "978-0767905923"
  }
}
```

Если для создания семейства столбцов используется суперстолбец, возникает *семейство суперстолбцов* (super column family).

```
//Семейство суперстолбцов
{
//Строка
name: "billing:martin-fowler",
value: {
  address: {
    name: "address:default",
    value: {
      fullName: "Martin Fowler",
      street:"100 N. Main Street",
      zip: "20145"
    }
  },
  billing: {
    name: "billing:default",
```

```
    value: {
      creditcard: "8888-8888-8888-8888",
      expDate: "12/2016"
    }
  }
}
//Строка
name: "billing:pramod-sadalage",
value: {
  address: {
      name: "address:default",
      value: {
        fullName: "Pramod Sadalage",
        street:"100 E. State Parkway",
        zip: "54130"
      }
    },
    billing: {
      name: "billing:default",
      value: {
      creditcard: "9999-8888-7777-4444",
      expDate: "01/2016"
      }
    }
  }
}
```

Семейства суперстолбцов удобны для хранения взаимосвязанных данных вместе. Однако, если некоторые столбцы бо́льшую часть времени не нужны, они все равно будут извлекаться и десериализироваться базой данных Cassandra, что может оказаться неоптимальным решением.

База данных Cassandra помещает семейства стандартных столбцов и суперстолбцов в *пространство ключей* (keyspaces), которое можно сравнить с базой данных в модели RDBMS. В нем хранятся все семейства столбцов, связанные с приложением. При создании пространства ключей ему назначается семейство столбцов.

```
create keyspace ecommerce
```

10.2.1. Согласованность данных

Когда Cassandra получает запрос на запись, данные сначала записываются в *журнал закрепления* (commit log), а затем в структуру, находящуюся в оперативной памяти и называемую *таблицей в памяти* (memtable). Операция записи считается успешной, если она записана и в журнал закрепления, и в таблицу в памяти. Записи группируются в памяти и периодически записываются в структуры, которые называются SSTable. После очистки структуры SSTable она не перезаписывается; измененные данные записываются в новую структуру SSTable. Неиспользуемые структуры SSTables удаляются в процессе уплотнения.

Посмотрим, как настройки согласованности данных влияют на выполнение операций чтения. Если уровень согласованности для всех операций чтения равен по умолчанию ONE, то после выполнения запроса на чтение Cassandra возвращает данные из первой реплики, даже если эти данные устарели. В последнем случае последующие операции чтения вернут более новые данные, — этот процесс называется *чтением с исправлением* (read repair.) Низкий уровень согласованности данных удобно использовать, когда пользователя не волнует, получает ли он устаревшие данные или нет, если обеспечивается высокая производительность чтения.

Аналогично при выполнении запроса на запись Cassandra выполняет запись в один из журналов закрепления и возвращает ответ клиенту. Уровень согласованности ONE приемлем в ситуациях, когда обеспечивается высокая производительность записи и пользователь не беспокоится о том, что некоторые записи могут оказаться утерянными, если узел выйдет из строя до того, как запись будет реплицирована на другой узел.

```
quorum = new ConfigurableConsistencyLevel();
quorum.setDefaultReadConsistencyLevel(HConsistencyLevel.QUORUM);
quorum.setDefaultWriteConsistencyLevel(HConsistencyLevel.QUORUM);
```

Использование уровня согласованности QUORUM для операций чтения и записи гарантирует, что на запрос чтения ответит большинство узлов и клиенту будет возвращен столбец с новейшей меткой времени, в то время как реплики, не содержащие новейших данных, будут исправлены с помощью операций чтения с исправлением. Уровень согласованности QUORUM для операции записи означает, что операция записи во время выполнения будет размножена по большинству узлов, прежде чем будет признана успешной и клиент будет уведомлен об этом.

Уровень согласованности ALL означает, что на запросы чтения и записи должны отвечать все узлы. В этом случае кластер будет уязвим для отказов — если из строя выйдет хотя бы один узел, операция записи или чтения будет заблокирована и объявлена ошибочной. Таким образом, проектировщик базы должен настраивать уровни согласованности в соответствии с требованиями приложения. В рамках одного и того же приложения требования к согласованности данных могут быть разными; они также изменяются в зависимости от операций, например, демонстрация комментариев к товарам и чтение статуса последнего заказа клиента могут иметь разные уровни согласованности.

Во время создания *пространства ключей* можно указать, сколько реплик данных следует хранить в базе. Это число определяет коэффициент репликации данных. Если коэффициент репликации равен 3, то данные копируются на три узла. Если задать параметр согласованности при чтении и записи данных в базе Cassandra равным 2, то сумма `R + W` окажется больше коэффициента репликации `(2 + 2 > 3)`, т.е. при чтении и записи будет обеспечен более высокий уровень согласованности данных.

Можно применить команду восстановления узла к пространству ключей и заставить базу Cassandra сравнить каждый ключ с остальными репликами. Эта операция связана с большими затратами, поэтому можно просто восстановить конкретное семейство столбцов или список семейств столбцов.

```
repair ecommerce
repair ecommerce customerInfo
```

Если узел вышел из строя, предполагается, что его данные были переданы другим узлам. Когда узел вернется в строй, изменения, внесенные в его данные, будут отправлены обратно. Этот метод называется *направленной отправкой* (hinted handoff). Направленная отправка позволяет быстрее восстанавливать узлы, вышедшие из строя.

10.2.2. Транзакции

В базе данных нет транзакций в традиционном смысле этого слова — возможности начинать множество операций записи, а затем решать — подтверждать изменения или нет. В базе данных Cassandra операция записи является атомарной на уровне строки. Это значит, что вставка или обновление столбцов по заданному ключу строки трактуется как отдельная запись и может быть либо успешной, либо ошибочной. Записи сначала записываются в журнал закрепления и таблицы в памяти и считаются успешными только в тех случаях, когда записи в журнал закрепления и таблицу в памяти оказались успешными. Если узел выходит из строя, для внесения изменений на узел используется журнал закрепления, точно так же, как это делает команда `redo log` в базе данных Oracle.

Для синхронизации операций записи и чтения существует возможность использования библиотек внешних транзакций, таких как ZooKeeper [ZooKeeper]. Кроме того, существуют библиотеки, такие как Cages [Cages], позволяющие перекрывать транзакции библиотеки ZooKeeper.

10.2.3. Доступность

База данных Cassandra задумана как очень доступная, потому что в ней нет ведущего узла и все узлы в кластере имеют одинаковые права. Доступность кластера можно увеличить, уменьшив уровень согласованности запросов. Доступность управляется формулой `(R + W) > N` (см. раздел 5.5, "Кворумы"), где `W` — минимальное количество узлов, в которых запись должна быть успешно записана; `R` — минимальное количество узлов, подтвердивших успешное чтение; `N` — количество узлов, участвующих в репликации данных. Уровень доступности можно настраивать, изменяя значения `R` и `W` при фиксированном значении `N`.

Если в кластере базы данных Cassandra, состоящем из 10 узлов с коэффициентом репликации для пространства ключей равным 3 (`N = 3`), положить `R = 2` и `W = 2`, то мы получим неравенство `(2 + 2) > 3`. Если в этом сценарии один из узлов выйдет из строя, то уровень доступности снизится незначительно, поскольку данные можно будет получить от других двух узлов. Если `W = 2` и `R = 1` и из строя выйдут два узла, то кластер станет недоступным для записи, но по-прежнему будет доступным для чтения. Аналогично, если `R = 2` и `W = 1`, мы сможем выполнять операции записи, но не сможем читать данные с кластера. При выполнении неравенства `R + W > N` можно выбирать компромиссы между согласованностью и доступностью.

Настройки пространств ключей и операций чтения/записи следует выбирать в зависимости от ваших потребностей — повысить доступность записи или чтения.

10.2.4. Функциональные возможности запросов

При разработке модели данных в базе Cassandra рекомендуется оптимизировать столбцы и семейства столбцов для чтения данных, поскольку в этой базе нет содержательного языка запросов; когда данные вставляются в семейства столбцов, данные в каждой строке упорядочиваются по именам столбцов. С точки зрения производительности в качестве ключа лучше использовать это значение.

10.2.4.1. Основные запросы

В число основных запросов, которые может выполнить клиент в базе Cassandra, входят запросы GET, SET и DEL. Перед выполнением запроса необходимо выполнить команду пространства ключей `use ecommerce;`. В этом случае все запросы будут адресоваться пространству ключей, в котором записаны ваши данные. Перед использованием семейства столбцов в пространстве ключей его необходимо определить.

```
CREATE COLUMN FAMILY Customer
WITH comparator = UTF8Type
AND key_validation_class=UTF8Type
AND column_metadata = [
{column_name: city, validation_class: UTF8Type}
{column_name: name, validation_class: UTF8Type}
{column_name: web, validation_class: UTF8Type}
];
```

Итак, у нас есть семейство столбцов `Customer` со столбцами `name`, `city` и `web`, и мы вставляем данные в семейство столбцов с помощью клиента базы Cassandra.

```
SET Customer['mfowler']['city']='Boston';
SET Customer['mfowler']['name']='Martin Fowler';
SET Customer['mfowler']['web']='www.martinfowler.com';
```

Те же самые данные можно вставить в семейство столбцов, используя Java-клиент Hector [Hector].

```
ColumnFamilyTemplate<String, String> template =
         cassandra.getColumnFamilyTemplate();
ColumnFamilyUpdater<String, String> updater =
         template.createUpdater(key);
for (String name : values.keySet()) {
    updater.setString(name, values.get(name));
}
try {
    template.update(updater);
} catch (HectorException e) {
    handleException(e);
}
```

Эти данные можно считать с помощью команды GET. Есть несколько способов получить эти данные; мы можем извлечь все семейство столбцов.

```
GET Customer['mfowler'];
```

Мы можем даже получить отдельный столбец из семейства столбцов.

```
GET Customer['mfowler']['web'];
```

Извлечение отдельного столбца является более эффективным, поскольку при этом возвращаются только данные, которые нас интересуют. В этом случае экономится масса операций по перемещению данных, особенно если семейство столбцов содержит много столбцов. Обновление данных происходит аналогично с помощью применения команды SET по отношению к столбцу, который должен получить новое значение.

Используя команду DEL, можно удалить как столбец, так и все семейство столбцов.

```
DEL Customer['mfowler']['city'];
DEL Customer['mfowler'];
```

10.2.4.2. Сложные запросы и индексация

База данных Cassandra позволяет индексировать столбцы не по ключам семейства столбцов. Например, можно определить индекс столбца city.

```
UPDATE COLUMN FAMILY Customer
WITH comparator = UTF8Type
AND column_metadata = [{column_name: city,
                        validation_class: UTF8Type,
                        index_type: KEYS}];
```

Теперь можно послать запрос прямо индексированному столбцу.

```
GET Customer WHERE city = 'Boston';
```

Эти индексы реализованы в виде *побитовых индексов* (bit-mapped indexes) и эффективны для небольших значений столбцов.

10.2.4.3. Язык запросов Cassandra Query Language (CQL)

База данных Cassandra имеет язык запросов Cassandra Query Language (CQL), поддерживающий SQL-подобные команды. Применим команды языка CQL для создания семейства столбцов.

```
CREATE COLUMNFAMILY Customer (
  KEY varchar PRIMARY KEY,
  name varchar,
  city varchar,
  web varchar);
```

Вставим те же данные с помощью языка CQL.

```
INSERT INTO Customer (KEY,name,city,web)
  VALUES ('mfowler',
          'Martin Fowler',
          'Boston',
          'www.martinfowler.com');
```

Читать данные можно с помощью команды SELECT. Прочитаем все столбцы.

```
SELECT * FROM Customer
```

С помощью команды SELECT можно прочитать не только все, но и отдельные интересующие нас столбцы.

```
SELECT name,web FROM Customer
```

Индексация столбцов осуществляется с помощью команды CREATE INDEX, которую можно использовать для запросов.

```
SELECT name,web FROM Customer WHERE city='Boston'
```

Язык CQL предусматривает много больше функциональных возможностей для запроса данных, но он уступает языку SQL. Например, язык CQL не допускает соединения и подзапросы, а его оператор where обычно имеет простую структуру.

10.2.5. Масштабирование

Масштабирование существующего кластера базы Cassandra сводится к добавлению дополнительных узлов. Поскольку ни один узел не является ведущим, добавляя новые узлы в кластер, мы повышаем его емкость и увеличиваем количество выполняемых операций записи и чтения. Этот вид горизонтального масштабирования позволяет достичь максимальной работоспособности, поскольку при добавлении новых узлов кластер не прекращает выполнение запросов, поступивших от клиентов.

10.3. Примеры использования

Обсудим некоторые ситуации, в которых семейства столбцов зарекомендовали себя с лучшей стороны.

10.3.1. Регистрация событий

Семейства столбцов, способные хранить любые структуры данных, отлично приспособлены для хранения информации о событиях, например, состояние приложения или ошибки, обнаруженные приложением. В масштабах предприятия все приложения могут записывать свои события в базу Cassandra в предназначенные для этого столбцы, а ключ строки имеет вид appname:timestamp. Поскольку записи можно масштабировать, база Cassandra идеально подходит для регистрации событий (рис. 10.2).

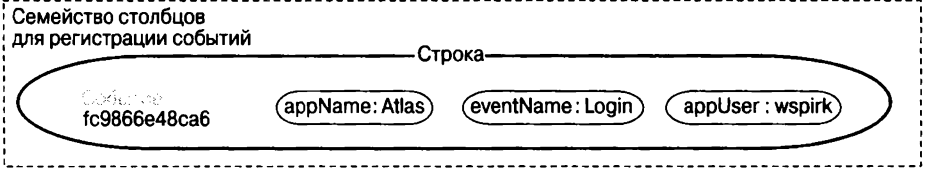


Рис. 10.2. Регистрация событий в базе Cassandra

10.3.2. Системы управления информационным наполнением, блог-платформы

С помощью семейств столбцов можно хранить записи блогов с ключевыми словами, категориями, ссылками и обратными ссылками в разных столбцах. Комментарии можно хранить либо в той же строке, либо переместить в другое пространство ключей; аналогично пользователей блога и актуальные блоги можно поместить в разные семейства столбцов.

10.3.3. Счетчики

В веб-приложениях часто возникает необходимость подсчитать и классифицировать посетителей, чтобы вычислить аналитические показатели веб-страницы. Во время создания семейства столбцов можно использовать счетчик `CounterColumnType`.

```
CREATE COLUMN FAMILY visit_counter
WITH default_validation_class=CounterColumnType
AND key_validation_class=UTF8Type AND comparator=UTF8Type;
```

После создания семейства столбцов каждому пользователю веб-приложения можно выделить произвольное количество столбцов для посещенных им веб-страниц.

```
INCR visit_counter['mfowler'][home] BY 1;
INCR visit_counter['mfowler'][products] BY 1;
INCR visit_counter['mfowler'][contactus] BY 1;
```

Увеличение счетчика на языке CQL выполняется следующим образом:

```
UPDATE visit_counter SET home = home + 1 WHERE KEY='mfowler'
```

10.3.4. Срок действия

Иногда возникает необходимость создать демо-версии для пользователей или в определенное время размещать на веб-сайте рекламные объявления. Для этого можно использовать *столбцы с ограниченным сроком действия* (expiring columns): база Cassandra позволяет создавать столбцы, которые автоматически удаляются по истечении определенного периода времени. Это время называется TTL (Time To Live — время существования) и задается в секундах. По истечении периода TTL столбец удаляется; если столбец больше не существует, запрос можно отменить, а рекламное объявление снять.

```
SET Customer['mfowler']['demo_access'] = 'allowed' WITH ttl=2592000;
```

10.4. Когда семейства столбцов использовать не следует

Существуют проблемы, которые семейство столбцов решает неэффективно. Например, это относится к системам, использующим для выполнения операций чтения и записи транзакции ACID. Если необходимо, чтобы база данных агрегировала данные с помощью запросов (например, SUM или AVG), это следует делать на клиентской стороне с помощью данных, извлеченных из всех строк.

Базу данных Cassandra не стоит использовать для создания ранних прототипов или первичных промышленных систем: на ранних стадиях еще не известно, как изменится шаблон запросов, а изменяя шаблоны запросов, мы вынуждены изменять проект семейства столбцов. Это создает сложности для коллектива инженеров-разработчиков и замедляет работу проектировщиков.

Парадигма RDBMS требует больших затрат для изменения схемы, но это компенсируется низкой стоимостью изменения запросов; в то же время затраты на изменение шаблонов запросов в базе данных Cassandra сопоставимы с затратами на изменение схемы в базах RDBMS.

Глава 11

Графовые базы данных

Графовые базы данных позволяют хранить сущности и отношения между ними. Сущности моделируются узлами, которые имеют свойства. Узел интерпретируется как экземпляр объекта в приложении. Отношения моделируются ребрами, которые могут иметь свойства. Ребра имеют направление; узлы организованы в соответствии с отношениями. Это позволяет находить требуемые шаблоны среди узлов. Организация графа позволяет один раз записать данные, а затем интерпретировать их разными способами в соответствии с отношениями.

11.1. Что такое графовая база данных

В примере, приведенном на рис. 11.1, показано множество узлов, связанных друг с другом. Узлы — это сущности, которые имеют свойства, например `name`. Узел `Martin` — это *узел*, имеющий *свойство* `name`, которому присвоено значение `Martin`.

Мы также видим, что ребра имеют типы, например `likes`, `author` и т.д. Эти свойства позволяют организовать узлы; например, узлы `Martin` и `Pramod` соединены *ребром*, имеющим тип отношения `friend`. Ребра могут иметь несколько свойств. Мы можем присвоить типу отношения `friend` между узлами `Martin` и `Pramod` свойство `since`. Типы отношения имеют направление; тип отношения `friend` является двунаправленным, а `likes` — нет. Если `Dawn likes NoSQL Distilled`, это не значит, что `NoSQL Distilled likes Dawn`.

Создав граф, состоящий из этих узлов и ребер, можно посылать к нему разнообразные запросы, например "найти все узлы для сотрудников компании `Big Co`, которым нравится книга `NoSQL Distilled`". Запрос к графу называется *обходом* графа. Преимущество графовых баз данных состоит в том, что мы можем изменять требования обхода, не изменяя узлы и ребра. Если нам нужно "найти все узлы для сотрудников компании `Big Co`, которым нравится книга `NoSQL Distilled`", то это можно сделать, не изменяя существующие данные или модель базы данных, поскольку мы можем обойти граф любым способом.

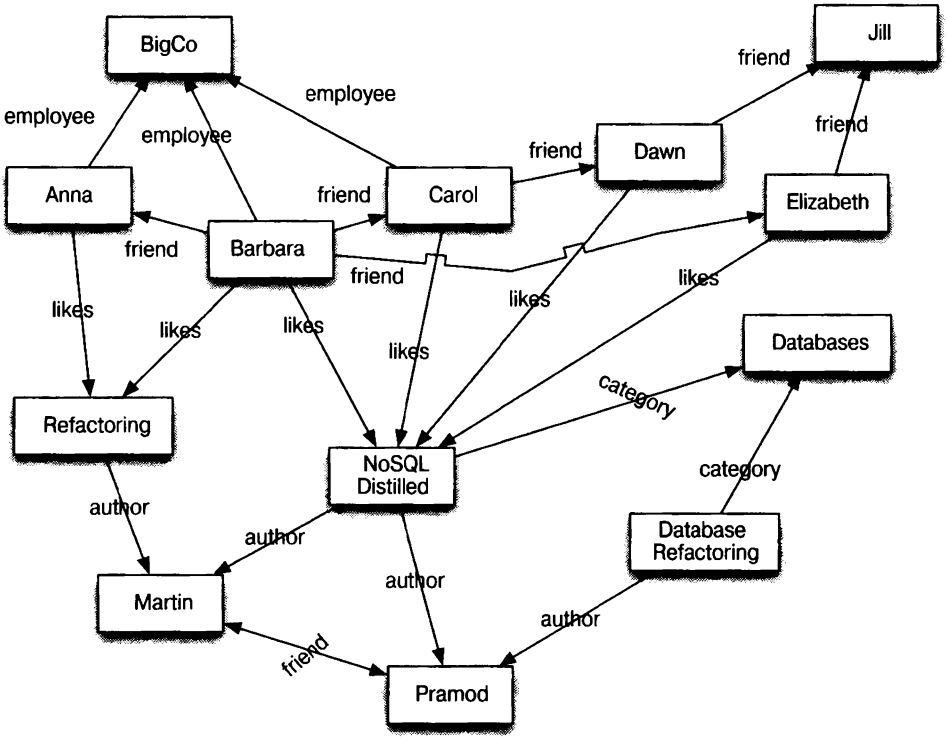


Рис. 11.1. Пример графовой структуры

Обычно, если графоподобная структура хранится в базе данных RDBMS, она имеет только один тип отношения (типичный пример — "кто мой начальник"). Для того чтобы добавить новое отношение в эту мешанину, приходится сильно изменять схему и перемещать много данных. В графовых базах данных этого делать не нужно. Аналогично в реляционных базах данных граф моделируется заранее в зависимости от требуемого свойства `Travesal`. Если свойство `Traversal` изменится, придется изменять данные.

В графовых базах данных обход отношений выполняется очень быстро. Отношение между узлами не вычисляется во время выполнения запроса, а постоянно хранится, т.е. является персистентным. Обход персистентных отношений выполняется быстрее, чем при вычислении отношений для каждого запроса.

Между узлами могут существовать разные типы отношений. Это позволяет представлять отношения как между предметными сущностями, так и создавать вторичные отношения для таких сущностей, как категория, путь, временны*е* деревья, деревья квадрантов для пространственного индексирования и связанные списки для упорядоченного доступа. Поскольку количество и виды отношений между узлами не ограничены, всех их можно представлять в одной и той же графовой базе данных.

11.2. Функциональные возможности

Существует много графовых баз данных, например Neo4J [Neo4J], Infinite Graph [Infinite Graph], OrientDB [OrientDB] и FlockDB [FlockDB] (которая представляет особый случай: эта графовая база данных поддерживает только отношения, глубина которых не превышает одного уровня, или списки смежности, в которых нельзя обойти отношения, имеющие несколько уровней глубины). В качестве типичного представителя графовой базы данных мы рассмотрим Neo4J и покажем на ее примере, как она работает и как с ее помощью решать прикладные задачи.

В базе Neo4J создание графа представляет собой простую задачу — достаточно создать два узла и отношение между ними. Создадим два узла: `Martin` и `Pramod`.

```
Node martin = graphDb.createNode();
martin.setProperty("name", "Martin");

Node pramod = graphDb.createNode();
pramod.setProperty("name", "Pramod");
```

Мы присвоили этим двум узлам свойство `name` со значениями `Martin` и `Pramod`. Поскольку в графе больше одного узла, мы можем создать отношение.

```
martin.createRelationshipTo(pramod, FRIEND);

pramod.createRelationshipTo(martin, FRIEND);
```

Мы должны создать отношение между узлами в двух направлениях, потому что направление отношения имеет значение. Например, узел `product` может нравиться узлу `user`, но узел `user` не может нравиться узлу `product`. Направленность отношения позволяет проектировать сложные предметно-ориентированные модели (рис. 11.2). Узлам известны входящие и исходящие отношения, которые можно обойти в обоих направлениях.

Отношения являются полноправными элементами графовых баз данных; ценность этих баз данных в основном обусловлена отношениями. Отношения имеют не только тип, начальный и конечный узел, но и свои собственные свойства. Используя эти свойства, в отношение можно внести информацию — например, когда узлы стали "друзьями", каково расстояние между узлами и что между ними общего. Эти свойства отношений можно использовать при создании запроса к графу.

Так как мощь графовых баз данных в основном обеспечивается отношениями и их свойствами, необходимо выполнить много аналитической и конструкторской работы, чтобы смоделировать отношения, существующие в предметной области, в которой мы работаем. Добавить новые типы сущностей легко; изменить существующие узлы и их отношения эквивалентно осуществлению миграции данных (см. раздел 12.3.2, "Миграция в графовых базах данных"), потому что эти изменения необходимо вносить в каждом узле и в каждом отношении между существующими данными.

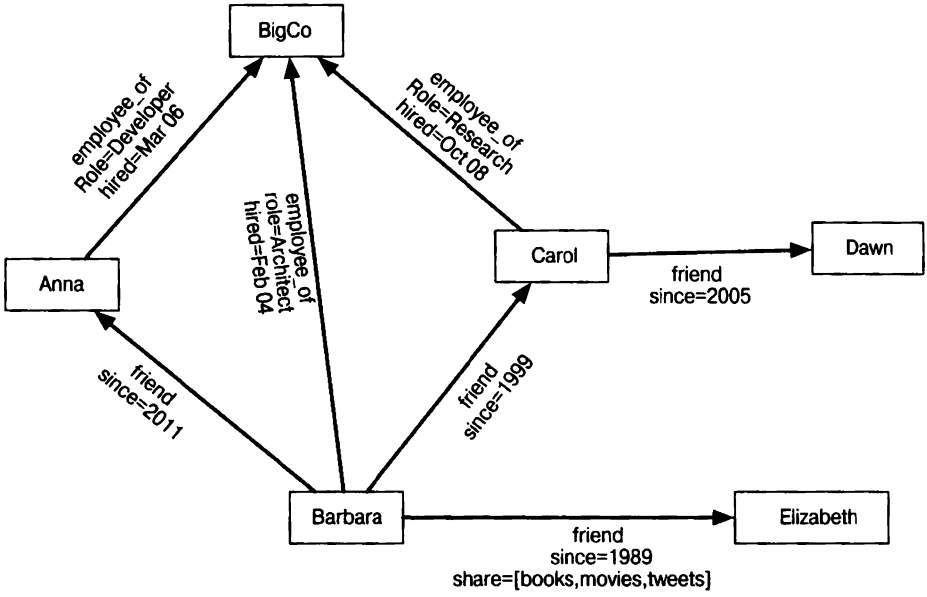


Рис. 11.2. Отношения и свойства

11.2.1. Согласованность данных

Поскольку графовые базы данных оперируют связанными узлами, большинство графовых баз данных не поддерживает распределение узлов по нескольким серверам. Однако существуют графовые базы данных, например Infinite Graph, поддерживающие распределение узлов по кластеру серверов. В рамках отдельного сервера данные всегда являются согласованными, особенно в базе Neo4J, которая полностью поддерживает транзакции ACID. Если база Neo4J работает на кластере, запись на ведущий узел синхронизируется с ведомыми узлами, которые всегда доступны для чтения. Операции записи на ведомые узлы всегда доступны и немедленно синхронизируются с ведущим узлом; остальные ведомые узлы не синхронизируются немедленно — они ждут, пока данные не будут распределены с ведущего узла.

Графовые базы обеспечивают согласованность данных с помощью транзакций. Они не допускают "висячие" отношения: начальный и конечный узлы всегда должны существовать, причем узел может быть удален только в том случае, если у него нет никаких отношений.

11.2.2. Транзакции

База Neo4J поддерживает транзакции ACID. Прежде чем изменить какой-нибудь узел или добавить какое-то отношение к существующим узлам, необходимо начать транзакцию. Если мы не упакуем операции в транзакции, то получим исключение

NotInTransactionException. Операции чтения можно выполнять без создания транзакций.

```
Transaction transaction = database.beginTx();
try {
    Node node = database.createNode();
    node.setProperty("name", "NoSQL Distilled");
    node.setProperty("published", "2012");
    transaction.success();
} finally {
    transaction.finish();
}
```

В этом коде мы начали транзакцию над базой данных, затем создали узел и задали его свойства. Мы пометили транзакцию функцией `success` и закончили функцией `finish`. Транзакция должна быть помечена функцией `success`, иначе база Neo4J будет считать, что произошла ошибка, и выполнит откат при вызове функции `finish`. Вызов функции `success` без вызова функции `finish` также не закрепит данные в базе данных. При разработке приложения следует помнить об этой особенности транзакций, которая отличается от транзакций в базе данных RDBMS.

11.2.3. Доступность

Начиная с версии Neo4J 1.8 высокая доступность базы обеспечивается реплицированными ведомыми узлами. Кроме того, эти ведомые узлы могут выполнять операции записи: свои записи они синхронизируют с текущим ведущим узлом и сначала закрепляют их на ведущем узле, а потом на ведомом. Это обновление в конце концов получают все ведомые узлы. В других база данных, таких как Infinite Graph и FlockDB, используется распределенное хранение узлов.

База Neo4J использует программу Apache ZooKeeper [ZooKeeper] для отслеживания идентификаторов последней транзакции на каждом ведомом и текущем ведущем узле. При загрузке сервер связывается с программой ZooKeeper и выясняет, какой узел является ведущим. Если сервер первым присоединяется к кластеру, он становится ведущим узлом; если ведущий узел выходит из строя, то кластер выбирает ведущий узел среди доступных узлов, обеспечивая высокую доступность.

11.2.4. Функциональные возможности запросов

Базы данных снабжаются языками запросов, такими как Gremlin [Gremlin]. Gremlin — это предметно-ориентированный язык для обхода графов; он может выполнять обход любых графовых баз данных, реализующих граф свойств Blueprints [Blueprints]. База Neo4J также поддерживает язык запросов Cypher [Cypher] для обхода графа. Помимо этих языков запросов, база Neo4J позволяет запрашивать свойства узлов, обходить граф и перемещаться по отношениям с помощью языковых привязок.

Свойства узла можно индексировать с помощью службы индексации. Свойства отношений или ребер также можно индексировать, поэтому узел или ребро можно найти по значению. Индексы следует запрашивать для того, чтобы найти стартовую точку для обхода графа. Рассмотрим поиск узла с помощью индексации.

Если мы ищем узел в графе, изображенном на рис. 11.1, то можем индексировать узлы либо в момент их добавления в базу данных, либо при их последующем обходе. Сначала необходимо создать индекс узлов с помощью поискового механизма **IndexManager**.

```
Index<Node> nodeIndex = graphDb.index().forNodes("nodes");
```

Мы индексируем узлы по свойству `name`. В качестве службы индексации в базе Neo4J используется механизм Lucene [Lucene]. Позднее мы продемонстрируем, как использовать полнотекстовые возможности поиска в механизме Lucene. После создания новых узлов их можно добавить в индекс.

```
Transaction transaction = graphDb.beginTx();
try {
    Index<Node> nodeIndex = graphDb.index().forNodes("nodes");
    nodeIndex.add(martin, "name", martin.getProperty("name"));
    nodeIndex.add(pramod, "name", pramod.getProperty("name"));
    transaction.success();
} finally {
    transaction.finish();
}
```

Добавление узлов в индекс выполняется в контексте транзакции. Как только узлы будут проиндексированы, мы можем искать их с помощью индексированного свойства. Если мы ищем узел `Barbara`, то можем запросить индекс для свойства `name`, имеющего значение `Barbara`.

```
Node node = nodeIndex.get("name", "Barbara").getSingle();
```

Найдем узел, в котором свойство `name` равно `Martin`. Имея этот узел, мы можем выяснить все его отношения.

```
Node martin = nodeIndex.get("name", "Martin").getSingle();
allRelationships = martin.getRelationships();
```

Кроме того, можно выяснить входящие и исходящие отношения.

```
incomingRelations = martin.getRelationships(Direction.INCOMING);
```

При запросе на отношение можно применять фильтры направлений. Если мы хотим найти всех людей, которым нравится книга `NoSQL Distilled` в графе на рис. 11.1, мы можем найти узел `NoSQL Distilled` и получить все его отношения со свойством `Direction.INCOMING`. В этот фильтр запроса можно также добавлять тип отношения, поскольку мы ищем только те узлы, которые связаны с узлом `NoSQL Distilled` отношением `LIKE`.

```
Node nosqlDistilled = nodeIndex.get("name",
                                    "NoSQL Distilled").getSingle();
relationships = nosqlDistilled.getRelationships(INCOMING, LIKES);
for (Relationship relationship : relationships) {
likesNoSQLDistilled.add(relationship.getStartNode());
}
```

Поиск узлов и их прямых отношений не представляет сложности, но это же можно делать в базах данных RDBMS. Настоящая мощь графовых баз данных проявляется в ситуациях, когда необходимо обойти граф на любой глубине и указать начальную точку обхода. Это особенно полезно при попытках найти узлы, связанные с начальной точкой, на более чем одном уровне глубины. При увеличении глубины графа более целесообразно обходить отношения с помощью функции `Traverser`, в которой можно задать, что мы ищем отношения типа `INCOMING`, `OUTGOING` или `BOTH`. Кроме того, можно задать способ обхода графа — в ширину или в глубину, — задав значения свойства `Order` равными `BREADTH_FIRST` или `DEPTH_FIRST`. Обход должен начинаться с какого-то узла — в нашем примере мы пытаемся найти все узлы на любой глубине, связанные с узлом `Barbara` отношением, имеющим тип тип `FRIEND`.

```
Node barbara = nodeIndex.get("name", "Barbara").getSingle();

Traverser friendsTraverser = barbara.traverse(Order.BREADTH_FIRST,
StopEvaluator.END_OF_GRAPH,
ReturnableEvaluator.ALL_BUT_START_NODE,
EdgeType.FRIEND,
Direction.OUTGOING);
```

Объект `friendsTraverser` позволяет найти все узлы, связанные с узлом `Barbara` отношением, имеющим тип `FRIEND`. Узлы могут находиться на любой глубине — друзья друзей любого уровня, — позволяя создавать древовидные структуры.

Одно из преимуществ графовых баз данных заключается в разнообразии возможностей поиска путей между двумя узлами — можно определить, есть ли несколько путей, найти все пути или кратчайший путь. В графе на рис. 11.1 мы знаем, что узел `Barbara` связан с узлом `Jill` двумя разными путями; для того чтобы найти все эти пути и расстояние между узлами `Barbara` и `Jill` по разным путям, можно использовать следующий код:

```
Node barbara = nodeIndex.get("name", "Barbara").getSingle();
Node jill = nodeIndex.get("name", "Jill").getSingle();
PathFinder<Path> finder = GraphAlgoFactory.allPaths(
      Traversal.expanderForTypes(FRIEND,Direction.OUTGOING)
                                    ,MAX_DEPTH);
Iterable<Path> paths = finder.findAllPaths(barbara, jill);
```

Эта функциональная возможность используется в социальных сетях для демонстрации отношений между двумя узлами. Для того чтобы найти все пути и расстояние между узлами вдоль каждого пути, сначала необходимо получить список разных путей между двумя узлами. Длина каждого пути равна *количеству переходов* на графе, которые необходимо сделать, чтобы достичь узла назначения из начальной точки. Часто

необходимо найти кратчайший путь между двумя узлами; среди путей, связывающих узлы Barbara или Jill, кратчайший путь можно найти с помощью следующих команд:

```
PathFinder<Path> finder = GraphAlgoFactory.shortestPath(
      Traversal.expanderForTypes(FRIEND, Direction.OUTGOING)
                                    , MAX_DEPTH);
Iterable<Path> paths = finder.findAllPaths(barbara, jill);
```

К графам можно применять разные алгоритмы, например, алгоритм Дийкстры[Dijkstra's] для поиска кратчайшего или самого дешевого пути между узлами.

```
START beginingNode = (beginning node specification)
MATCH (relationship, pattern matches)
WHERE (filtering condition: on data in nodes and relationships)
RETURN (What to return: nodes, relationships, properties)
ORDER BY (properties to order by)
SKIP (nodes to skip from top)
LIMIT (limit results)
```

Для создания запросов к графам база Neo4J предусматривает язык запросов Cypher. Для того чтобы начать запрос с помощью команды START, в языке Cypher необходимо указать начальный узел. Его можно задать с помощью идентификатора, списка идентификаторов узла или просмотра индекса. Для сравнения шаблонов в отношениях в языке Cypher используется ключевое слово MATCH; ключевое слово WHERE фильтрует свойства узла или отношения. Ключевое слово RETURN указывает, что будет возвращено запросом — узлы, отношения или поля в узлах и отношениях.

В языке Cypher есть методы ORDER, AGGREGATE, SKIP и LIMIT для упорядочения, агрегирования, пропуска и ограничения данных. Используя обозначение --, мы можем найти на рис. 11.2 все узлы, связанные с узлом Barbara входящими или исходящими отношениями.

```
START barbara = node:nodeIndex(name = "Barbara")
MATCH (barbara)--(connected_node)
RETURN connected_node
```

Если нас интересуют отношения с заданным направлением, можно записать команду

```
MATCH (barbara)<--(connected_node)
```

для входящих отношений или команду

```
MATCH (barbara)-->(connected_node)
```

для исходящих отношений. Сравнение можно применять и к конкретным отношениям, используя соглашение :RELATIONSHIP_TYPE и возвращая требуемые поля или узлы.

```
START barbara = node:nodeIndex(name = "Barbara")
MATCH (barbara)-[:FRIEND]->(friend_node)
RETURN friend_node.name,friend_node.location
```

Мы начинаем поиск с узла Barbara, находим все исходящие отношения с типом FRIEND и возвращаем имена друзей. Запрос типа отношения работает только на глубине, равной одному уровню; для того чтобы увеличить глубину и выяснить глубину для каждого результирующего узла, можно выполнить следующий код:

```
START barbara=node:nodeIndex(name = "Barbara")
MATCH path = barbara-[:FRIEND*1..3]->end_node
RETURN barbara.name,end_node.name, length(path)
```

Аналогично можно создать запрос на отношения с заданным свойством. Можно также создать фильтр свойства отношений и послать запрос, существует ли такое свойство или нет.

```
START barbara = node:nodeIndex(name = "Barbara")
MATCH (barbara)-[relation]->(related_node)
WHERE type(relation) = 'FRIEND' AND relation.share
RETURN related_node.name, relation.since
```

В языке Cypher есть много других запросов, которые можно использовать при работе с графовыми базами данных.

11.2.5. Масштабирование

При масштабировании баз данных NoSQL широко используется фрагментация, в ходе которой данные разделяются и распределяются по разным серверам. В графовых базах данных фрагментацию сделать трудно, поскольку они ориентированы не на агрегаты, а на отношения. Поскольку любой узел может быть связан отношением с любым другим узлом, хранение связанных узлов на одном и том же сервере позволяет повысить эффективность обхода графа. Обход графа, узлы которого разбросаны по разным компьютерам, имеет низкую эффективность. Зная об этом ограничении графовых баз данных, мы все же можем масштабировать их с помощью общепринятых методов, описанных Джимом Уэббером [Webber Neo4J Scaling].

В принципе существуют три способа масштабирования графовых баз данных. Поскольку в настоящее время компьютеры могут иметь большой объем оперативной памяти, можно добавить на сервер столько микросхем, чтобы работать с множеством узлов и отношений, находящимся целиком в оперативной памяти. Этот метод оказывается полезным только в тех случаях, когда множество данных, с которым мы работаем, действительно может поместиться в оперативной памяти.

Можно улучшить масштабирование чтения, добавив дополнительные ведомые узлы, обеспечивающие только чтение данных, а все операции записи выполнять на ведущем узле. Такой способ, предусматривающий однократную запись данных и их чтение со многих серверов, хорошо зарекомендовал себя на кластерах MySQL и оказался действительно полезным при работе с наборами данных, которые достаточно велики, чтобы не помещаться в оперативной памяти одного компьютера, но достаточно малы, чтобы создать их реплики на разных компьютерах. Ведомые узлы повышают доступность и облегчают масштабирование чтения, поскольку их можно сконфигурировать так, чтобы они никогда не становились ведущими и выполняли только операции чтения.

Если набор данных настолько велик, что создавать его реплики нецелесообразно, можно выполнить фрагментацию данных (см. раздел 4.2, "Фрагментация") на стороне приложения, используя знания о предметной области. Например, узлы, связанные с Северной Америкой, можно создать на одном сервере, а узлы, связанные с Азией, — на другом. Выполняя такую фрагментацию на стороне приложения, следует понимать, что узлы хранятся в физически разных базах данных (рис. 11.3).

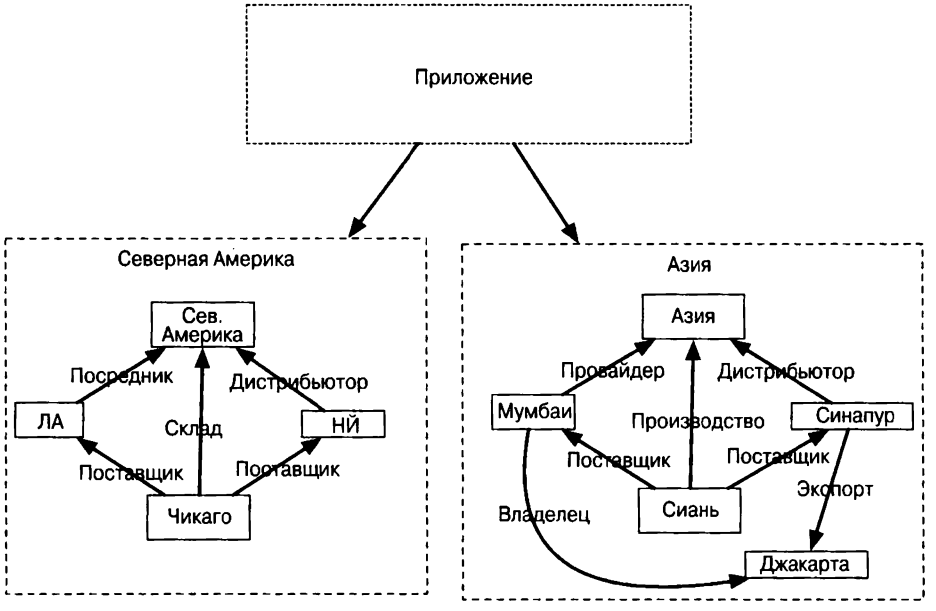


Рис. 11.3. Фрагментация узлов на уровне приложения

11.3. Примеры использования

Рассмотрим примеры использования графовых баз данных.

11.3.1. Связанные данные

Графовые данные можно развернуть и очень эффективно использовать в социальных сетях. Эти социальные графы не обязательно должны охватывать друзей; например, они могут представлять сотрудников, их знания, а также сотрудничество с коллегами в разных проектах. Любая предметная область с богатыми взаимными связями подходит для описания с помощью графовой базы данных.

Если в одной и той же базе данных существуют отношения между сущностями из разных предметных областей (например, социальные, географические и коммерческие связи), можно повысить их информативность, предусмотрев возможность обхода графа с пересечением границ предметных областей.

11.3.2. Маршрутизация, диспетчеризация и геолокационные сервисы

Каждое место назначения или адрес можно представить в виде узла, а все узлы, в которые необходимо выполнить доставку, можно моделировать с помощью графа. Отношения между узлами могут иметь отношение, описывающее расстояние. Это позволяет обеспечить эффективную доставку товаров. Свойства расстояния и адреса можно использовать в графах, описывающих предпочтения пользователей, так что ваше приложение может выдавать рекомендации о хороших ресторанах или местах для развлечений, расположенных поблизости. Можно также создать узлы для точек продаж, например книжных магазинов или ресторанов, и уведомить пользователей, что они находятся поблизости, используя геолокационную службу.

11.3.3. Справочные базы данных

После того как в системе созданы узлы и отношения, их можно использовать для выдачи рекомендаций типа "ваши друзья также купили этот товар" или "при заказе этого товара обычно также заказывают следующие товары". Кроме того, можно рассылать рекомендации туристам, сообщая им, что гости, приезжающие в Барселону, обычно осматривают архитектурные творения Антонио Гауди.

У таких справочных баз данных есть один интересный побочный эффект — при увеличении объема баз данных количество узлов и отношений, доступных для выдачи рекомендаций, быстро увеличивается. Те же самые данные можно использовать для добычи информации — например, какие товары всегда покупают или заказывают вместе; если эти условия не выполняются, можно выдавать предупреждение. Как и другие справочные системы, графовые базы данных можно использовать для поиска шаблонов среди отношений, чтобы выявить нарушения в транзакциях.

11.4. Когда не следует использовать графовые базы данных

В некоторых ситуациях графовые базы данных могут оказаться неприемлемыми. Если вы хотите обновлять все или часть сущностей, например, при выработке аналитического решения, в котором свойства всех сущностей могут быть изменены, графовые базы данных могут стать неоптимальными, поскольку изменение свойства во всех узлах — непростая операция. Даже если модель данных для предметной области оказывается работоспособной, некоторые базы данных не могут справиться с большим объемом данных, особенно при выполнении глобальных операций над графом (т.е. операций, затрагивающих весь граф).

Глава 12

Миграции схем

12.1. Изменения схемы

В последнее время при обсуждении баз данных NoSQL в первую очередь подчеркивают их *бесструктурную* природу — популярную функциональную возможность, позволяющую разработчикам сконцентрироваться на предметно-ориентированном проектировании, не беспокоясь об изменениях схемы. Это особенно справедливо в контексте гибких методов [Agile Methods], в которых важно учитывать возможности изменения схемы.

Для правильного понимания данных необходимы обсуждения, итерации и циклы обратной связи, в которых участвуют эксперты по предметной области и владельцы программ, причем эти дискуссии нельзя загромождать сложностью схем баз данных. В хранилищах данных NoSQL изменения схемы можно сделать с наименьшими затратами сил, повысив производительность проектирования (см. раздел 1.5, "Появление баз данных NoSQL"). Разработка и сопровождение приложений в новом мире неструктурированных баз данных требует повышенного внимания к миграции схем.

12.2. Изменение схем в базах данных RDBMS

При разработке баз по стандартной технологии RDBMS мы разрабатываем объекты, соответствующие им таблицы и их отношения. Рассмотрим простую объектную модель и модель данных, содержащую объекты `Customer`, `Order` и `OrderItems`. Модель "объект–отношение" в этой базе будет выглядеть так, как показано на рис. 12.1.

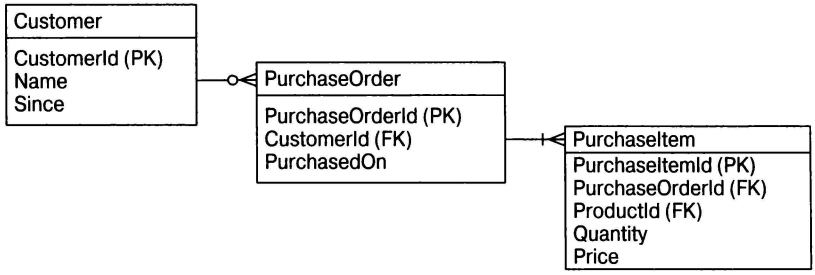


Рис. 12.1. Модель данных в системе электронной коммерции

Пока эта модель данных соответствует текущей объектной модели, все хорошо. Но при первом же изменении объектной модели, например при вводе поля `preferredShippingType` в объект `Customer`, придется изменить объект и таблицу, поскольку без изменения таблицы приложение не будет синхронизировано с базой данных. Получив сообщение об ошибке наподобие `ORA-00942: table or view does not exist` или `ORA-00904: "PREFERRED_SHIPPING_TYPE": invalid identifier`, мы узнаем, что возникла проблема.

Обычно миграция схемы базы данных сама по себе является предметом проекта. Для развертывания изменений схемы разрабатываются сценарии изменения базы и применяются дифференциальные методы. Этот подход к созданию сценариев миграции во время развертывания и выпуска баз данных уязвим к ошибкам и не поддерживает гибкие методы разработки.

12.2.1. Миграция в проектах, начинающихся с нуля

Разрабатывать сценарии изменений схемы базы данных лучше во время разработки, поскольку эти изменения можно хранить вместе со сценариями миграции данных в одном и том же файле сценария. Эти файлы сценариев следует называть с указанием возрастающих номеров, чтобы отобразить версии базы данных; например, первое изменение базы данных может иметь файл сценария с именем `001_Description_Of_Change.sql`.

Создание сценариев изменений позволяет сохранить порядок изменений при миграции данных. На рис. 12.2 показана папка со всеми изменениями, внесенными в базу данных до текущего момента.

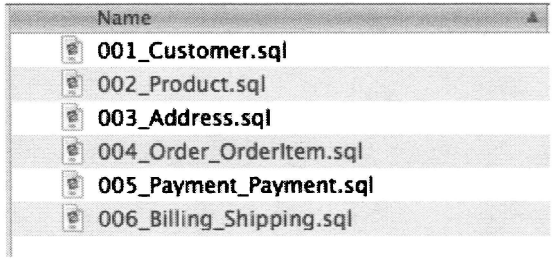


Рис. 12.2. Последовательность миграций в базе данных

Теперь предположим, что нам необходимо изменить таблицу `OrderItem`, чтобы хранить в ней записи `DiscountedPrice` и `FullPrice`. Для этого придется изменить таблицу `OrderItem`. Это изменение в нашей последовательности изменений получит номер `007`, как показано на рис. 12.3.

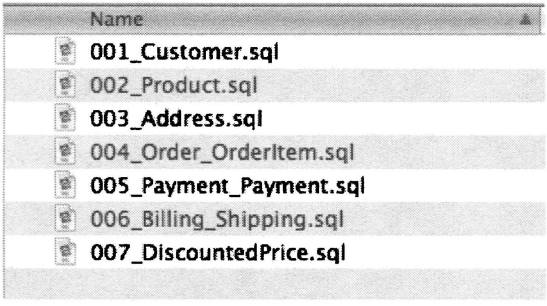


Рис. 12.3. Новое изменение 007_DiscountedPrice.sql

Мы внесли в базу данных новое изменение. Сценарий этого изменения содержит код, предусматривающий добавление нового столбца, удаление существующего столбца и миграцию данных, необходимых для обеспечения новой функциональной возможности. Ниже показан сценарий изменений из файла `007_DiscountedPrice.sql`.

```
ALTER TABLE orderitem ADD discountedprice NUMBER(18,2) NULL;
UPDATE orderitem SET discountedprice = price;
ALTER TABLE orderitem MODIFY discountedprice NOT NULL;
ALTER TABLE orderitem RENAME COLUMN price TO fullprice;
--//@UNDO
ALTER TABLE orderitem RENAME fullprice TO price;
ALTER TABLE orderitem DROP COLUMN discountedprice;
```

Этот сценарий изменений показывает, что при изменении схемы базы необходимо осуществлять миграцию данных. В показанном примере для управления изменениями базы данных мы использовали каркас DBDeploy [DBDeploy], поддерживающий в базе данных таблицу `ChangeLog`, в которой хранятся все изменения, внесенные в базу данных. В этой таблице есть запись `Change_Number`, в которой хранятся номера изменений, внесенных в базу данных. Значение `Change_Number`, представляющее собой номер версии базы данных, используется для поиска соответствующего пронумерованного сценария и внесения всех изменений, которые еще не были внесены. Записывая сценарий изменений с номером `007` и применяя его к базе данных с помощью каркаса DBDeploy, мы проверяем таблицу `ChangeLog` и извлекаем из папки все сценарии, которые еще не были осуществлены. На рис. 12.4 показан снимок экрана во время применения каркаса DBDeploy к базе данных.

Для обеспечения интеграции с остальными разработчиками лучше всего использовать репозиторий контроля версий, в котором хранятся все сценарии изменений. Это позволяет отслеживать версии программного обеспечения и базы данных в одном и том же месте и исключить возможные несогласованности между базой данных и приложением. Для таких обновлений существует много других инструментов, например Liquibase [Liquibase], MyBatis Migrator [MyBatis Migrator] и DBMaintain [DBMaintain].

```
project $>ant dbupgrade
Buildfile: /project/build.xml

init:

dbupgrade:
 [dbdeploy] dbdeploy 3.0M3
 [dbdeploy] Reading change scripts from directory /project/db/migrations...
 [dbdeploy] Changes currently applied to database:
 [dbdeploy]   1..6
 [dbdeploy] Scripts available:
 [dbdeploy]   1..7
 [dbdeploy] To be applied:
 [dbdeploy]   7
 [dbdeploy] Applying #7: 007_DiscountedPrice.sql...
 [dbdeploy]  -> statement 1 of 4...
 [dbdeploy]  -> statement 2 of 4...
 [dbdeploy]  -> statement 3 of 4...
 [dbdeploy]  -> statement 4 of 4...

BUILD SUCCESSFUL
Total time: 0 seconds
project $>
```

Рис. 12.4. Обновление базы данных с помощью сценария 007 и каркаса DBDeploy

12.2.2. Миграция в унаследованных проектах

Не каждый проект начинается с нуля. Как реализовать миграцию, если приложение уже находится в производстве? Для этого можно создать базовый вариант проекта, отражающий структуру существующей базы данных в виде сценариев, содержащих код и все ссылки. Однако это не относится к транзакциям. После того как базовый вариант будет готов, дальнейшие изменения можно осуществить с помощью методов миграции, описанных выше (рис. 12.5).

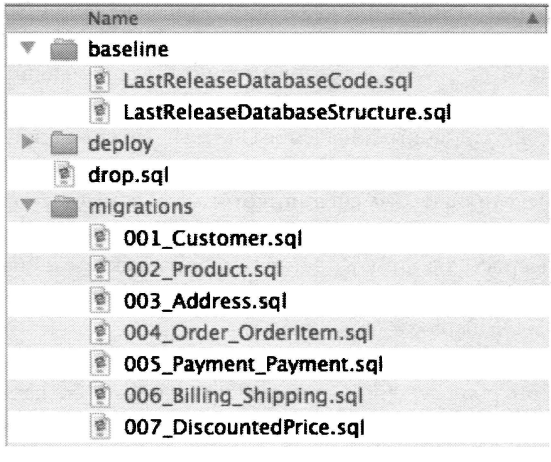


Рис. 12.5. Использование сценариев базового варианта с унаследованной базой данных

Один из главных аспектов миграции — поддержка обратной совместимости схемы базы данных. Во многих предприятиях существуют многочисленные приложения, использующие базы данных; когда изменяется база данных для одного приложения, это изменение не должно нарушать работу других приложений. Обеспечить обратную совместимость можно с помощью переходного этапа, подробно описанного в книге *Refactoring Databases* [Ambler and Sadalage].

В течение *переходного этапа* старая и новая схемы поддерживаются одновременно и остаются доступными для всех приложений, использующих базу данных. Для этого необходимо создать вспомогательный код, например триггеры, представления и виртуальные столбцы. Это обеспечит остальным приложениям доступ к схеме и данным без изменения кода приложений.

```
ALTER TABLE customer ADD fullname VARCHAR2(60);
UPDATE customer SET fullname = fname;
CREATE OR REPLACE TRIGGER SyncCustomerFullName
BEFORE INSERT OR UPDATE
ON customer
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
BEGIN
IF :NEW.fname IS NULL THEN
:NEW.fname := :NEW.fullname;
END IF;
IF :NEW.fullname IS NULL THEN
:NEW.fullname := :NEW.fname
END IF;
END;
/
--Drop Trigger and fname
--when all applications start using customer.fullname
```

В данном примере мы попытались переименовать столбец `customer.fname` в `customer.fullname`, поскольку хотели избежать неоднозначности интерпретации имени `fname` как `fullname` или `firstname`. Прямое переименование столбца `fname` и изменение кода приложения, с которым мы работаем, может сработать для нашего приложения, но не для остальных приложений предприятия, имеющих доступ к той же самой базе данных.

На переходном этапе мы вводим новый столбец `fullname`, копируем данные в него, но оставляем старый столбец `fname`. Мы также вводим триггер `BEFORE UPDATE`, синхронизирующий данные до того, как они будут закреплены в базе данных. Теперь, когда приложения считают данные из таблицы, они прочитают столбец `fname` или `fullname`, но данные всегда будут правильными. После того как все приложения перейдут на использование нового столбца `fullname`, триггер и столбец `fname` можно будет удалить.

Миграция схемы в больших базах данных RDBMS всегда представляет собой трудную задачу, особенно если необходимо сохранить доступность базы для остальных приложений, поскольку перенос больших массивов данных и структурных изменений обычно вызывает блокировки таблиц в базах данных.

12.3. Изменение схем в хранилищах данных NoSQL

База данных RDBMS должна быть изменена до того, как будет изменено приложение. Именно этого стремится избежать *бесструктурный подход* (schema-free, or schemaless), обеспечивая гибкость изменения схем на уровне сущностей. Частые изменения схемы являются следствием реакции на частые изменения рынка и появление новых товаров.

В некоторых случаях при разработке баз данных NoSQL схема не должна рассматриваться заблаговременно. Мы по-прежнему должны думать о проекте и его аспектах, таких как типы отношений (в графовых базах), имена строк, столбцов и семейств столбцов, порядок столбцов (в семействах столбцов), а также присвоение ключей и структуры данных в объекте-значении (в хранилищах типа "ключ–значение"). Даже если мы не рассмотрели все эти вопросы заранее или хотим изменить свое решение, это легко исправить.

Утверждение, что базы данных NoSQL являются полностью неструктурированными, может ввести в заблуждение. Несмотря на то что базы данных NoSQL хранят данные без оглядки на их схему, она определяется приложением, поскольку приложение должно выполнять синтаксический анализ потока данных при его чтении из базы. Кроме того, база данных должна создавать данные, которые должны храниться в базе. Если приложение не может выполнить синтаксический анализ данных из базы, возникает несогласованность схемы, но в данном случае сообщение об этой ошибке выдает не база данных RDBMS, а приложение. Таким образом, даже в неструктурированных базах данных схема данных должна учитываться при рефакторинге приложения.

Изменения схемы становятся особенно важными, если приложение уже развернуто и данные записаны в базу. Для простоты предположим, что мы используем документную базу данных, например MongoDB [MongoDB], и ту же модель данных, что и прежде: `customer`, `order` и `orderItems`.

```
{
"_id": "4BD8AE97C47016442AF4A580",
"customerid": 99999,
"name": "Foo Sushi Inc",
"since": "12/12/2012",
"order": {
    "orderid": "4821-UXWE-122012","orderdate": "12/12/2001",
    "orderItems": [{"product": "Fortune Cookies",
                    "price": 19.99}]
   }
}
```

Код приложения для записи этого документа в базу данных MongoDB выглядит следующим образом:

```
BasicDBObject orderItem = new BasicDBObject();
orderItem.put("product", productName);
orderItem.put("price", price);
orderItems.add(orderItem);
```

Код для чтения документа из базы данных имеет следующий вид:

```
BasicDBObject item = (BasicDBObject) orderItem;
String productName = item.getString("product");
Double price = item.getDouble("price");
```

Изменение объектов путем добавления атрибута `preferredShippingType` не требует внесения изменений в базу данных, поскольку базе данных безразлично, что разные документы имеют разные схемы. Это позволяет быстрее разрабатывать и проще развертывать базы данных Достаточно лишь развернуть приложение, а на стороне базы данных никаких изменений не требуется. Код лишь должен сделать так, чтобы эти документы больше не были обязаны анализировать атрибут `preferredShippingType` — вот и все.

Разумеется, мы упростили ситуацию. Рассмотрим схему изменений, внесенных ранее: ввод атрибута `discountedPrice` и переименование атрибута `price` цены в `fullPrice`. Для того чтобы осуществить эти изменения, мы переименовываем атрибут `price` в `fullPrice` и добавляем атрибут `discountedPrice`. Измененный атрибут выглядит так:

```
{
"_id": "5BD8AE97C47016442AF4A580",
"customerid": 66778,
"name": "India House",
"since": "12/12/2012",
"order": {
  "orderid": "4821-UXWE-222012",
    "orderdate": "12/12/2001",
    "orderItems": [{"product": "Chair Covers",
                    "fullPrice": 29.99,
                    "discountedPrice":26.99}]
  }
}
```

Когда мы развернем это изменение, информация о новых клиентах и их заказах будет записываться и считываться без проблем, но мы не сможем прочитать цены для существующих заказов, потому что код будет искать атрибут `fullPrice`, а в документе есть лишь атрибут `price`.

12.3.1. Постепенная миграция

Несогласованность схем порождает проблему преобразования данных в технологии NoSQL. Если изменение схемы происходит в приложении, мы должны преобразовать все существующие данные в новую схему (если данных много, это может оказаться затратной операцией).

Другая возможность заключается в том, что перед изменением схемы ее можно анализировать в новом коде, а при сохранении вернуться к старой схеме. Этот метод, известный как *постепенная миграция* (incremental migration), подразумевает

последовательный перенос данных; некоторые данные никогда не будут перенесены, поскольку к ним никто не обращался. Считаем атрибуты `price` и `fullPrice` из документа.

```
BasicDBObject item = (BasicDBObject) orderItem;
String productName = item.getString("product");
Double fullPrice = item.getDouble("price");
if (fullPrice == null) {
    fullPrice = item.getDouble("fullPrice");
}
Double discountedPrice = item.getDouble("discountedPrice");
```

При записи этого документа старый атрибут `price` не сохраняется.

```
BasicDBObject orderItem = new BasicDBObject();
orderItem.put("product", productName);
orderItem.put("fullPrice", price);
orderItem.put("discountedPrice", discountedPrice);
orderItems.add(orderItem);
```

При использовании постепенной миграции на стороне приложения могут возникнуть несколько версий объекта, которые могут трансформировать старую схему в новую; при возвращении объекта в базу будет записываться его новая версия. Эта постепенная миграция данных обеспечивает более быструю эволюцию приложения.

Метод постепенной миграции усложняет проектирование объектов, особенно при введении новых изменений и сохранении старых. Период между развертыванием изменений и переносом последнего объекта в базе данных в новую схему называется переходным (рис. 12.6). Этот период следует сделать как можно короче, сосредоточив внимание на минимально возможном количестве изменений, — это позволит сохранить порядок среди объектов.

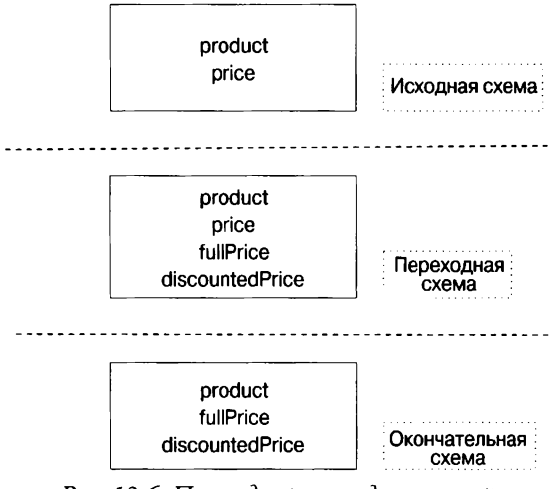


Рис. 12.6. Переходный период изменений

Метод постепенной миграции можно осуществить с помощью поля данных `schema_version`, которое приложение использует для выбора правильного кода при анализе данных. При сохранении данные переходят на последнюю версию, и этот факт отображается с помощью обновления поля `schema_version`.

Очень важно иметь переходной слой между предметной областью и базой данных, чтобы при изменении схемы управление несколькими версиями ограничивалось лишь переходным слоем и не распространялось на все приложение.

Мобильные приложения выдвигают особые требования. Поскольку мы не можем принудительно обновлять самое свежее приложение, само приложение должно обрабатывать почти все варианты схемы.

12.3.2. Миграция в графовых базах данных

В графовых базах данных есть ребра, которые имеют типы и свойства. Если в коде базы изменить тип всех этих ребер, обойти базу станет невозможно, а рендеринг будет бесполезен. Для того чтобы решить эту проблему, можно обойти все ребра и изменить их тип. Эта операция может оказаться затратной и потребовать написать код для обхода всех ребер в базе данных.

Если необходимо обеспечить обратную совместимость или вы не хотите изменять весь граф сразу, можно просто создать новые ребра между узлами; позднее, в более удобной ситуации, старые ребра можно удалить. При обходе графа можно использовать как новые, так и старые ребра. Этот метод очень удобен при работе с крупными базами данных, особенно, если требуется обеспечивать их высокую доступность.

Если необходимо изменить свойства всех узлов или ребер, мы должны найти все узлы и изменить все требуемые свойства. Например, можно добавить атрибуты `NodeCreatedBy` и `NodeCreatedOn` ко всем существующим узлам, чтобы отслеживать изменения, сделанные в каждом узле.

```
for (Node node : database.getAllNodes()) {
     node.setProperty("NodeCreatedBy", getSystemUser());
     node.setProperty("NodeCreatedOn", getSystemTimeStamp());
}
```

Иногда в узлах необходимо изменить данные. Новые данные могут быть выведены из данных в существующих узлах или импортированы из других источников. Такую миграцию можно осуществить, найдя все узлы с помощью индекса, предоставленного источником, и записать в каждый узел соответствующие данные.

12.3.3. Изменение агрегатной структуры

Иногда требуется изменить проект схемы, например, разделить крупные объекты на более мелкие и хранить их независимо друг от друга. Допустим, у нас есть агрегат, содержащий все заказы клиента, и мы хотим отделить информацию о клиенте от его заказов, разместив их в разных агрегатах.

Затем необходимо сделать так, чтобы код мог работать как с новыми, так и со старыми версиями агрегатов. Если он не найдет старые объекты, то будет искать новые.

Код, работающий в фоновом режиме, может считывать по одному агрегату, вносить требуемые изменения и сохранять данные в разных агрегатах. Преимущество работы с одним агрегатом в каждый момент времени заключается в том, что это не влияет на доступность приложения.

12.4. Рекомендации для дальнейшего чтения

Более подробно о миграции в реляционных базах данных написано в работе [Ambler and Sadalage]. Несмотря на то что эта книга посвящена реляционным базам данных, общие принципы миграции относятся ко всем базам данных.

12.5. Резюме

- Миграцию баз данных со строгими схемами, например реляционных, можно осуществить с помощью сохранения изменений каждой схемы и переноса данных в соответствии с порядком версий.
- Неструктурированные базы данных требуют осторожности при миграции из-за того, что любой код, имеющий доступ к данным, неявно подразумевает наличие определенной схемы.
- Неструктурированные базы данных могут использовать те же методы миграции, что и базы данных со строгими схемами.
- Неструктурированные базы данных также могут читать данные, используя способ, устойчивый к изменениям в неявной схеме данных, и допускают постепенную миграцию при обновлении данных.

Глава 13

Многовариантная персистентность

Разные базы предназначены для решения разных проблем. Использование единственного процессора базы данных для всех случаев жизни обычно приводит к неэффективным решениям; хранение транзакционных данных, кеширование информации о сеансе, обход графа, содержащего данные о пользователях и товарах, купленных их друзьями, — это совершенно разные проблемы. Даже в области баз данных RDBMS требования систем OLAP и OLTP сильно отличаются друг от друга, и тем не менее они часто втискиваются в одну и ту же схему.

Подумайте об отношениях между данными. Парадигма RDBMS отличается тем, что требует, чтобы эти отношения существовали изначально. Если же мы хотим идентифицировать эти отношения или найти данные из разных таблиц, принадлежащие одному и тому же объекту, то применять базы RDBMS становится сложно.

Процессоры баз данных предназначены для высокоэффективного выполнения определенных операций над определенными структурами данных и определенными объемами данных, например, для быстрой обработки наборов данных, или хранения и извлечения ключей и их значений, или хранения информационно насыщенных документов или сложных графов информации.

13.1. Разброс требований к хранилищам данных

Многие предприятия стараются использовать один и тот же процессор базы данных и для хранения деловых транзакций, и для управления информацией о сеансах, и для других целей, таких как создание отчетов, бизнес-аналитики, организации информационных хранилищ и регистрации информации (рис. 13.1).

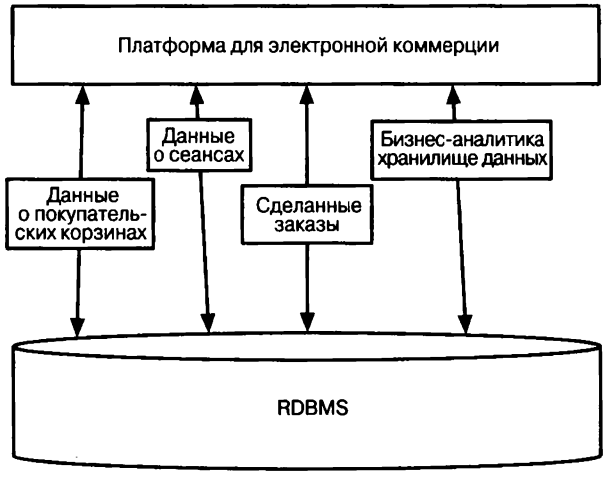


Рис. 13.1. Использование баз RDBMS для любого аспекта хранения данных, предназначенных для приложения

Данные о сеансах, покупательских корзинах или заказах не требуют одинаковой доступности, согласованности или резервного копирования. Разве хранилище данных о сеансах должно реализовывать такую же строгую стратегию резервного копирования и восстановления, как и хранилище данных о заказах? Разве хранилище данных о сеансах должно быть более доступным, чем процессор базы данных для записи и чтения сеансовых данных?

В 2006 году Нил Форд (Neal Ford) изобрел термин *многоязычное программирование* (polyglot programming), означающий, что приложения следует писать на смеси языков, извлекая пользу из того, что разные языки предназначены для решения разных задач. Сложные приложения объединяют задачи разных типов, поэтому выбор правильного языка для каждой задачи может оказаться более продуктивным, чем попытка использовать один язык для учета всех аспектов.

Аналогично при решении бизнес-проблем в системах электронной коммерции для хранения информации о покупательских корзинах следует использовать хранилище, обеспечивающее высокую степень доступности и допускающее масштабирование, но это же хранилище не поможет при поиске товаров, купленных друзьями клиентов, поскольку это совершенно другая задача. Для обозначения гибридного подхода к персистентности мы будем использовать термин *многовариантная персистентность* (polyglot persistence).

13.2. Использование многовариантного хранилища данных

Рассмотрим систему для электронной коммерции и применим многовариантный подход, чтобы увидеть, как можно применить разные хранилища данных (рис. 13.2). Хранилище типа "ключ–значение" можно использовать для записи информации о покупательских корзинах до подтверждения заказа клиентом, а также для хранения

данных о сеансах, отказавшись от хранения переменных данных в базах RDBMS. Применение хранилища "ключ–значение" для этих целей более эффективно, потому что доступ к корзине покупателя обычно открывается по идентификатору пользователя, а после подтверждения заказа и его оплаты эту информацию можно записать в базу данных RDBMS. Аналогично ключами для данных о сеансе служат идентификаторы сеанса.

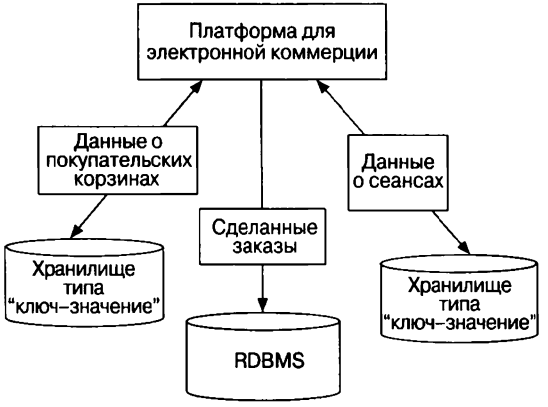


Рис. 13.2. Использование хранилища типа "ключ–значение" для выгрузки данных о сеансах и хранения информации о покупательских корзинах

Если вы хотите порекомендовать клиенту какие-то товары в момент, когда он помещает покупки в свою корзину, например, "ваши друзья тоже купили эти товары" или "ваши друзья купили эти аксессуары для этого товары", целесообразно использовать графовое хранилище данных (рис. 13.3).

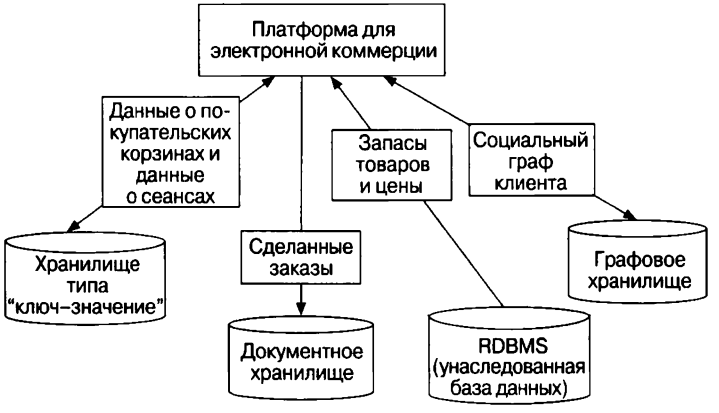


Рис. 13.3. Пример реализации многовариантной персистентности

Приложение не обязано использовать одно и то же хранилище для удовлетворения всех своих потребностей, поскольку разные базы данных предназначены для разных целей и не все проблемы можно одинаково элегантно решить с помощью одной базы данных.

Даже использование специализированных баз данных для разных целей, таких как организация хранилищ данных или бизнес-аналитика в рамках одного и того же приложения, можно рассматривать как проявление многовариантной персистентности.

13.3. Использование сервисов при работе с хранилищем данных

При переходе на работу приложения с несколькими хранилищами данных на предприятии могут обнаружиться и другие приложения, использующие наши хранилища. В нашем примере графовое хранилище может обслуживать данные других приложений, которым необходимо выяснить, какие товары чаще покупают клиенты из определенного сегмента пользовательской базы.

Вместо независимой работы каждого приложения с графовой базой данных ее можно упаковать в сервис, чтобы все отношения между узлами можно было хранить в одном месте и открыть для всех приложений (рис. 13.4). Возможность владения данными и интерфейсы прикладного программирования, предоставляемые сервисом, оказываются более полезными, чем работа одного приложения с несколькими базами данных.

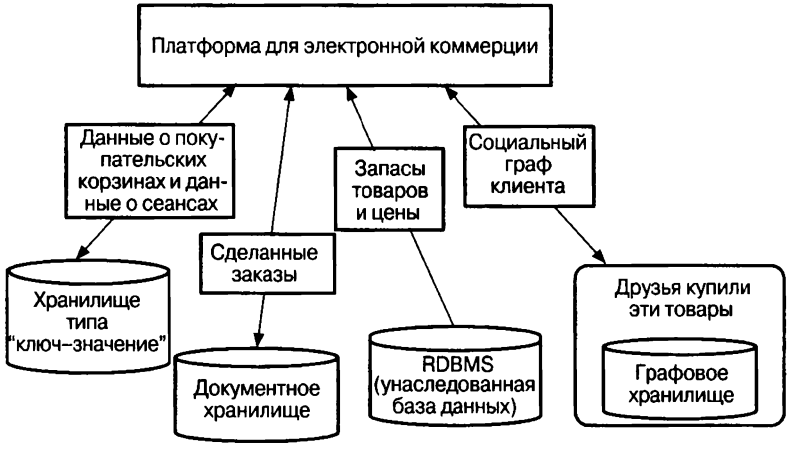


Рис. 13.4. Пример упаковки хранилищ данных в сервисы

Идею упаковки хранилищ в сервис можно развить. Мы могли бы упаковать все базы данных в сервисы, разрешив приложению общаться только с этими сервисами (рис. 13.5). Это позволит модифицировать базы данных в сервисах без изменения зависящих от них приложений.

Многие хранилища NoSQL, такие как Riak [Riak] и Neo4J [Neo4J], на самом деле предоставляют готовые интерфейсы прикладного программирования REST.

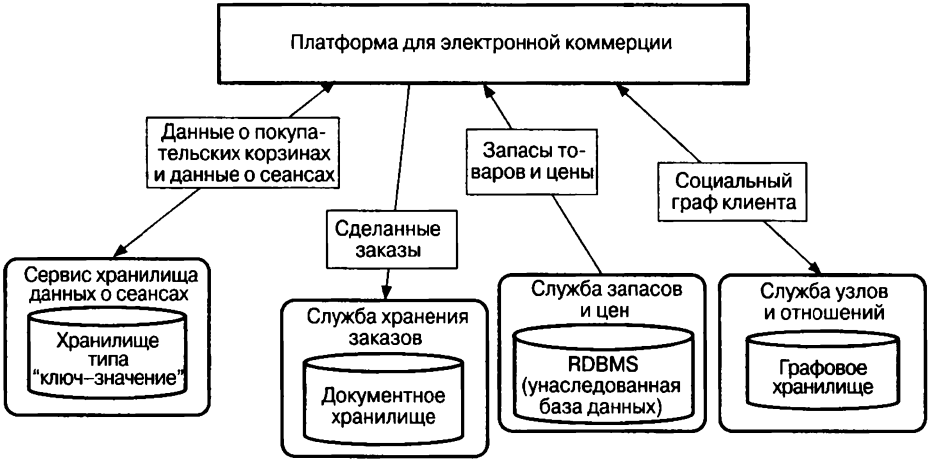


Рис. 13.5. Использование сервисов вместо обращения к базам данных

13.4. Расширение функциональных возможностей

Часто оказывается невозможным перестроить хранилище данных из-за унаследованных приложений и их зависимости от существующих хранилищ. Однако можно добавить функциональные возможности, например кеширование или механизмы индексации, такие как Solr [Solr], чтобы повысить эффективность поиска (рис. 13.6). Применяя такие технологии, необходимо синхронизировать данные между хранилищем и кешем или механизмом индексации.

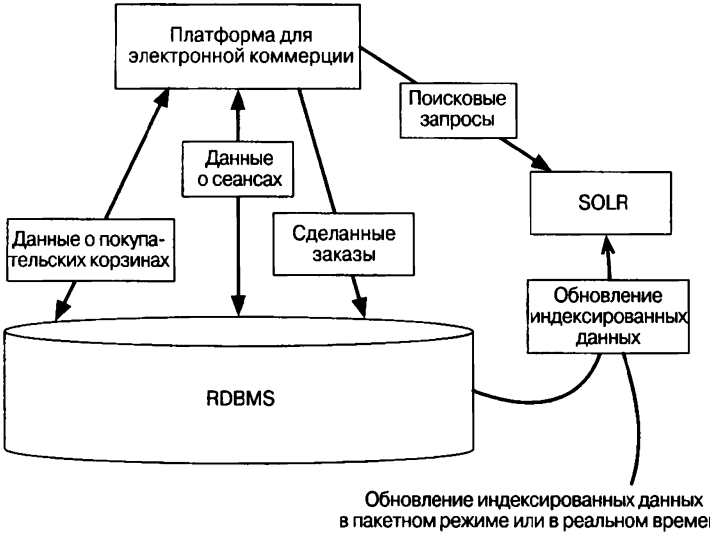


Рис. 13.5. Использование дополнительного хранилища для расширения возможностей унаследованного хранилища

В таком случае необходимо обновлять индексированные данные при изменении данных в базе приложения. Обновление данных можно выполнять в пакетном режиме или в реальном времени, чтобы приложение могло работать с устаревшими приложениями в механизме индексации и поиска. Для обновления индекса можно использовать шаблон порождения событий (см. раздел 14.2, "Порождение событий").

13.5. Выбор правильной технологии

Существует богатый выбор хранилищ данных. Сначала маятник качнулся от специализированных баз данных к отдельной базе RDBMS, допускающей хранение любых моделей данных, хотя и с некоторой натяжкой. В настоящее время маятник качнулся в сторону использования хранилищ, естественным образом поддерживающих реализацию решений.

Если требуется рекомендовать товары клиентам на основе информации об их покупательской корзине и других клиентах, купивших этот же товар, можно выбрать любое хранилище, содержащее данные с соответствующими атрибутами. Главное — выбрать правильную технологию, чтобы при изменении вопросов можно было использовать то же самое хранилище без утраты старых данных или изменения их формата.

Вернемся к нашему примеру. Для расширения функциональных возможностей мы могли бы использовать хранилище RDBMS с иерархическими запросами и соответствующими таблицами. Если потребуется изменить обход, мы будем вынуждены перестроить базу данных, перенести данные и записать новые данные. Если же мы используем хранилище, отслеживающее отношения между узлами, то будет достаточно перепрограммировать новые отношения и хранить их в том же хранилище с минимальными изменениями.

13.6. Многовариантная персистентность в масштабе предприятия

Внедрение технологий NoSQL вынудило администраторов баз данных подумать об использовании нового хранилища. Предприятия привыкли использовать однородную среду RDBMS; та база данных, которую предприятие использовало первой, на многие годы становилась ориентиром для всех приложений. В новом мире многовариантной персистентности группы управления базами данных становятся многосторонними — для того чтобы понять, как работают технологии NoSQL, как осуществлять мониторинг этих систем, выполнять их резервное копирование, считывать данные и записывать их в эти системы.

Решив перейти на технологию NoSQL, предприятие сталкивается с такими вопросами, как лицензирование, сопровождение, инструментарий, обновления, драйверы, аудит и безопасность. Многие технологии NoSQL являются проектами с открытым кодом и поддерживаются активными участниками специализированных сообществ;

кроме того, существуют компании, осуществляющие коммерческую поддержку. Существует широкий выбор инструментов, но поставщики инструментов и сообщества разработчиков программ с открытым кодом постоянно выпускают новые продукты, такие как MongoDB Monitoring Service [Monitoring], Datastax Ops Center [OpsCenter] или браузер Rekon для базы данных Riak [Rekon].

Еще одна проблема, которая встает перед предприятиями, — безопасность данных, т.е. возможность создавать пользователей и присваивать им привилегии, регламентирующие доступ к данным. Большинство баз данных NoSQL не имеют очень надежных систем безопасности. Это объясняется тем, что они разрабатывались для разных целей. В традиционных системах RDBMS данные обслуживались базой данных и выдавались пользователям по запросам. В базах NoSQL также существует механизм запросов, но идея заключается в том, что приложение само владеет данными и обслуживает их с помощью сервисов. В рамках такого подхода ответственность за безопасность перекладывается на приложение. С другой стороны, существуют технологии NoSQL, решающие вопросы безопасности.

Предприятия часто имеют системы управления хранилищами данных и аналитические системы, запрашивающие данные из многовариантных источников данных. Предприятия должны гарантировать, что инструменты ETL и другие механизмы, предусматривающие перенос данных из источника в хранилище, смогут прочитать данные из хранилища NoSQL. Появляются поставщики инструментов ETL, предусматривающие способность общения с базами данных NoSQL, например, программное обеспечение Pentaho [Pentaho] может работать с базами данных MongoDB и Cassandra.

Каждое предприятие выполняет определенные аналитические расчеты. При увеличении объема необходимых данных предприятие сталкивается с необходимостью масштабировать системы RDBMS, чтобы записывать в них данные. Огромное количество операций записи и необходимость их масштабирования предоставляет прекрасную возможность для внедрения баз данных NoSQL, позволяющих записывать большие объемы данных.

13.7. Сложность развертывания

Вступив на путь многовариантной персистентности, мы должны внимательно рассмотреть вопросы, связанные со *сложностью развертывания* (deployment complexity). Теперь приложение требует, чтобы все базы данных разрабатывались одновременно. Все эти базы данных одновременно должны находиться в средах UAT, QA и Dev. Поскольку большинство продуктов NoSQL имеют открытый исходный код, расходы на покупку лицензий невелики. Кроме того, они поддерживают автоматическую инсталляцию и конфигурирование. Например, для того чтобы инсталлировать базу данных, достаточно лишь загрузить ее и распаковать архив с помощью команд `curl` и `unzip`. Эти продукты также предусматривают довольно разумные настройки по умолчанию и могут начинать работу в минимальной конфигурации.

13.8. Резюме

- Многовариантная персистентность означает использование разных технологий хранения данных для удовлетворения разных потребностей приложений.
- Многовариантная персистентность может реализовываться как в рамках всего предприятия, так и в пределах отдельного приложения.
- Инкапсуляция доступа данных в сервисах уменьшает влияние выбора хранилища на другие части системы.
- Расширение спектра технологий хранения данных усложняет программирование и выполнение операций, поэтому преимущества эффективного хранилища должны соответствовать его уровню сложности.

Глава 14

За рамками технологии NoSQL

Появление технологии NoSQL встряхнуло мир баз данных и открыло новые перспективы, но, по нашему мнению, разновидности баз данных NoSQL, которые мы рассмотрели ранее, образуют лишь часть общей картины многовариантной персистентности. По этой причине целесообразно потратить немного времени на обсуждение решений, выходящих за рамки технологии NoSQL.

14.1. Файловые системы

Базы данных распространены очень широко, но файловые системы практически вездесущи. В течение последних десятилетий файловые системы широко использовались для хранения персональных документов, но не применялись в промышленных приложениях. У них нет никакой внутренней структуры, поэтому они напоминают хранилища типа "ключ–значение" с иерархическим ключом. Кроме того, в файловых системах практически нет средств для управления параллельной работой, кроме простого блокирования файлов. Впрочем, это напоминает технологию NoSQL, в которой предусмотрена лишь блокировка отдельного агрегата.

Преимущество файловых систем состоит в их простоте и широком распространении. Они хорошо справляются с хранением крупных объектов, таких как видео- и аудиофайлы. Часто базы данных применяются для индексирования аудиовизуальной информации, хранящейся в файле. Файлы очень удобны для последовательного доступа, например потоков. Это удобно, если данные только добавляются в файловую систему.

Недавно возникший интерес к кластерным средам стимулировал разработку распределенных файловых систем. Такие технологии, как Google File System и Hadoop [Hadoop], поддерживают репликацию файлов. Большая часть обсуждения парадигмы "отображение–свертка" сводится к изучению манипуляций крупными файлами на кластерных системах с помощью средств автоматического разделения файлов на сегменты, обрабатываемые на многочисленных узлах. Организации, использовавшие систему Hadoop, естественным образом перешли на технологию NoSQL.

Файловые системы лучше всего справляются с малочисленными большими файлами, которые можно обрабатывать крупными порциями, желательно в потоковом режиме. Большое количество маленьких файлов обычно обрабатываются плохо. Именно

здесь хранилища данных оказываются более эффективными. Файлы также не поддерживают обработку запросов без применения дополнительных механизмов индексации, таких как Solr [Solr].

14.2. Порождение событий

Порождение событий (event sourcing) — это подход, концентрирующий внимание на долговременном хранении всех изменений персистентного состояния, а не самого текущего состояния. Это архитектурный шаблон, достаточно хорошо работающий с большинством технологий обеспечения персистентности, включая реляционные базы данных. Мы упоминаем его здесь потому, что он позволяет обсудить необычные подходы к персистентности.

Рассмотрим в качестве примера систему, хранящую регистрационный журнал, в котором записано местоположение кораблей (рис. 14.1). Он содержит простую запись о корабле, в котором указано его название и текущие координаты. При обычном подходе, когда мы слышим, что корабль *King Roy* прибыл в Сан-Франциско, мы изменяем значение поля `location`, связанное с кораблем *King Roy*, на `San Francisco`. Позже мы слышим, что корабль отплыл, поэтому меняем значение этого поля на `at sea`, а затем снова изменяем его, узнав, что корабль прибыл в Гонконг.

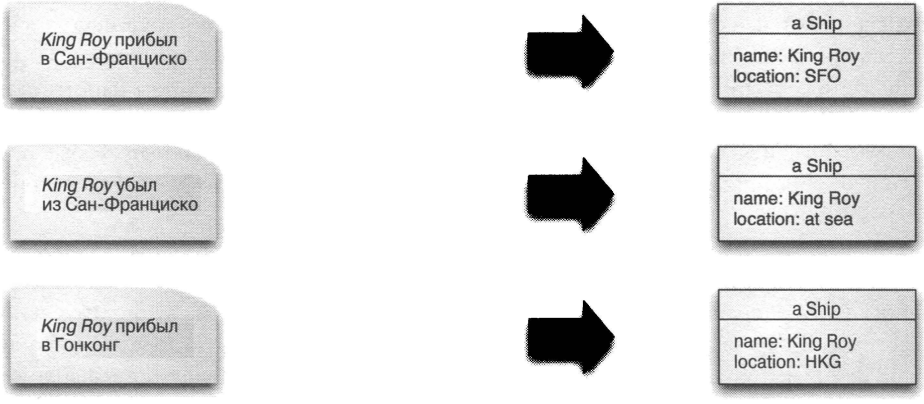


Рис. 14.1. В типичной системе уведомление об изменении приводит к обновлению состояния приложения

В системе порождения событий на первом этапе конструируется объект события, хранящий информацию об изменении (рис. 14.2). Этот объект события хранится в долговременном журнале регистрации событий. В заключение мы обрабатываем события по порядку, чтобы обновить состояние приложения.

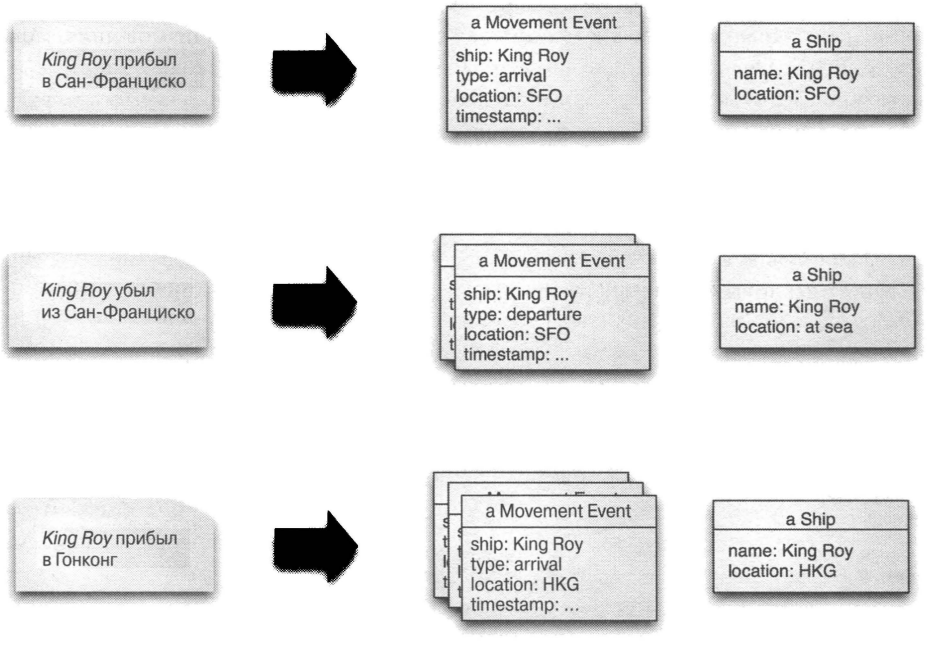


Рис. 14.2. При порождении событий система хранит каждое событие вместе с соответствующим состоянием приложения

Таким образом, в системе порождения событий мы храним каждое событие, влияющее на состояние системы, в журнале регистрации событий, а состояние приложения полностью определяется этим журналом. В любой момент времени мы можем безопасно удалить состояние приложения и восстановить его по журналу.

С теоретической точки зрения журналы регистрации событий — это все, что нужно, потому что состояние приложения всегда можно восстановить по журналу. На практике этот процесс может оказаться слишком медленным. В результате обычно лучше всего обеспечить возможность для хранения и восстановления состояния приложения в виде моментального снимка. *Моментальный снимок* (snapshot) предназначен для хранения образа памяти, оптимизированного для быстрого восстановления состояния. Поскольку снимки предназначены для оптимизации, по авторитетности они всегда уступают журналу регистрации событий.

Частота создания моментальных снимков зависит от требований к сроку безотказной работы системы. Моментальный снимок не обязательно должен быть совершенно свежим, поскольку можно перестроить память, загрузив последний моментальный снимок, а затем восстановить все события, произошедшие после момента его создания. В нашем примере моментальные снимки можно было бы делать каждую ночь; если система выйдет из строя в течение дня, то можно загрузить моментальный снимок, сделанный прошлой ночью, и восстановить ход событий, произошедших днем. Если это можно сделать достаточно быстро, все будет в порядке.

Для того чтобы получить полную запись о каждом изменении в состоянии приложения, необходимо вести журнал событий с самого начала работы приложения. Однако во многих случаях такие долговременные записи не нужны, поскольку старые события можно свернуть в моментальный снимок и использовать только ту часть журнала, которая была заполнена после создания снимка.

Порождение событий имеет массу преимуществ. События можно рассылать нескольким системам, каждая из которых может создавать разные состояния приложения для разных целей (рис. 14.3). В системах с интенсивным чтением можно предусмотреть несколько узлов для чтения с потенциально разными схемами, сконцентрировав операции записи на другой системе (этот подход широко известен как CQRS [CQRS]).

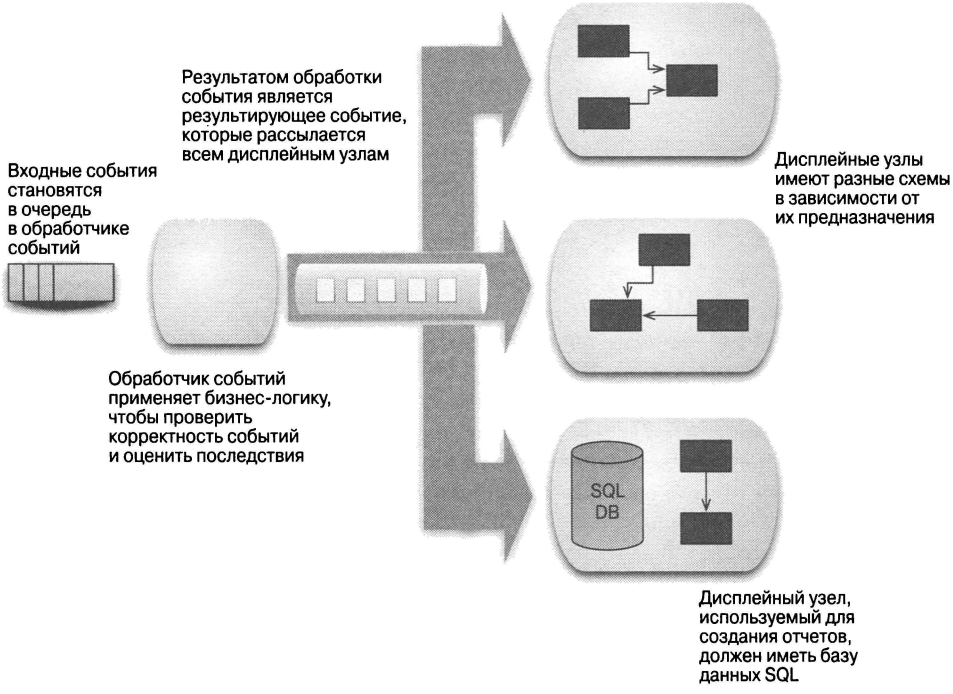


Рис. 14.3. События можно рассылать нескольким дисплейным системам

Кроме того, порождение событий является эффективной платформой для анализа исторической информации, поскольку в журнале событий можно реплицировать любое прошлое состояние. Можно также легко анализировать альтернативные сценарии, вводя в обработчик гипотетические события.

Порождение событий создает определенные сложности. Самой большой сложностью является необходимость фиксации и хранения всех изменений состояния в виде событий. Некоторые архитектуры и инструменты для этого оказываются неудобными. Любое взаимодействие с внешними системами должно учитываться системой порождения событий; следует остерегаться внешних побочных эффектов при воспроизведении событий для восстановления состояния приложения.

14.3. Образ памяти

Одним из последствий порождения событий является то, что журнал регистрации событий по существу становится персистентной записью, но состояние приложения не обязано быть персистентным. Это открывает возможность сохранения состояния приложения в виде структур данных, хранящихся в памяти. Хранение всех оперативных данных в памяти повышает производительность работы, поскольку при обработке событий не приходится выполнять операции чтения с диска и записи на диск. Это также упрощает программирование, поскольку отпадает необходимость выполнять отображение между диском и структурами данных, находящимися в памяти.

Очевидное ограничение заключается в том, что все необходимые данные приходится хранить в памяти. Со временем это требование все легче выполнить — в настоящее время размер оперативной памяти часто намного превышает размер диска. Кроме того, необходимо иметь возможность быстро восстанавливать работу системы после сбоя, либо заново загружая события из журнала регистрации событий, либо запуская дублирующую систему и переводя данные из одной систему в другую.

Для обеспечения параллельной работы нужен какой-то явный механизм. Одним из возможных вариантов является система транзакционной памяти, например, система, поставляемая вместе с языком Clojure. Другой вариант — выполнять всю обработку входной информации в одном потоке. Если тщательно спроектировать этот процесс, то однопотоковый обработчик событий может обеспечить впечатляющую скорость обработки данных с очень маленькой задержкой [Fowler Imax].

Разделение данных на оперативные и персистентные влияет на процесс обработки ошибок. Общепринятым подходом является обновление модели и откат всех изменений при возникновении ошибки. При использовании образа памяти автоматический откат обычно не используется; вы либо обязаны написать свою собственную процедуру отката (что довольно сложно), либо гарантировать тщательную проверку до внесения каких-либо изменений.

14.4. Контроль версий

Для большинства программистов опыт работы с системой регистрации сообщений ограничивается системой контроля версий. Контроль версий позволяет многим членам команды координировать свои модификации сложной взаимосвязанной системы, а также исследовать старые состояния этой системы и ее альтернативные варианты.

Размышляя о хранилище данных, мы обычно думаем о нем с точки зрения отдельного разработчика. Это очень узкая точка зрения по сравнению со сложностью, обеспечиваемой системой управления версиями. Таким образом, нет ничего удивительного в том, что инструменты для хранилищ данных позаимствовали некоторые идеи у систем управления версиями. Помимо всего прочего, во многих ситуациях требуются запросы к историческим данным и поддержка нескольких точек зрения.

Системы управления версиями создаются на основе файловых систем и поэтому накладывают на хранилище данных те же ограничения, что и на файловые системы. Они не предназначены для создания хранилищ данных, поэтому их сложно использовать в этом контексте. Однако следует учитывать сценарии, в которых их способность регистрировать события во времени может оказаться полезной.

14.5. Базы данных XML

В начале тысячелетия люди полагали, что язык XML будет использоваться повсюду, поэтому возник взрыв интереса к базам данных, разработанным для хранения XML-документов и обработки запросов к этим документам. Хотя этот порыв не оказал большого влияния на доминирование реляционных баз данных, как и многие предшествующие пустые обещания, базы данных XML не исчезли.

Мы рассматриваем базы данных XML как документные базы, в которых документы хранятся в модели данных, совместимой с языком XML, а для манипуляции с документами используются технологии XML. В этих базах можно проверять формат документов с помощью разных схем XML (DTD, XML Schema, RelaxNG), выполнять запросы с помощью языков XPath и Xquery и выполнять трансформации с помощью языка XSLT.

Реляционные базы данных интегрировали язык XML и объединили его возможности с реляционными. Обычно это выражается в виде использования документов XML в качестве типа столбца и смешивании языков запросов SQL и XML.

Разумеется, нет никаких препятствий для того, чтобы использовать язык XML как механизм структуризации в хранилище типа "ключ–значение". Язык XML в настоящее время стал менее модным, чем формат JSON, но он также удобен для хранения сложных агрегатов, а возможности схемы языка XML и его механизма выполнения запросов шире, чем у формата JSON. Использование базы данных XML означает, что сама база может использовать преимущества структуры языка, а не хранить данные в виде простого двоичного объекта. Однако это преимущество необходимо согласовывать с другими характеристиками базы данных.

14.6. Объектные базы данных

Во времена расцвета популярности объектно-ориентированного программирования возник сильный интерес к объектно-ориентированным базам данных. При этом упор делался на сложность отображения структур данных, находящихся в памяти, в реляционные таблицы. Объектно-ориентированные базы данных должны были устранить эту сложность — база данных должна была автоматически управлять процессом записи на жесткий диск структур данных, находящихся в записи. Такую систему можно представить как систему персистентной виртуальной памяти, позволяющей программе работать с персистентными данными, не уведомляя об этом базу данных.

Объектные базы не получили развития. Одна из причин заключалась в том, что выигрыш от тесной интеграции с приложением компенсировался усложнением доступа к данным в обход приложения. Переход от интегрированных баз данных к базам данных приложения мог бы обеспечить развитие объектных баз данных в будущем.

Важным аспектом объектных баз данных является миграция при изменении структуры данных. Тесная связь между персистентным хранилищем и структурами данных, хранящимися в памяти, может стать проблемой. Некоторые объектные базы данных предусматривают возможность добавлять функции миграции в определения объекта.

14.7. Резюме

- NoSQL — это просто технология хранения данных. Поскольку она способствует многовариантной персистентности, мы должны решить, заслуживает ли та или иная технология хранения данных названия NoSQL.

Глава 15

Выбор базы данных

К данному моменту мы обсудили многие из общих вопросов, касающихся принятия решений в области многовариантной персистентности. Настало время поговорить о выборе базы данных для будущего проекта. Естественно, мы не знаем конкретных обстоятельств и поэтому не можем ни дать определенный совет, ни свести его к набору простых правил. Более того, системы NoSQL появились относительно недавно, поэтому эта область знаний еще не достигла зрелости — через несколько лет мы, возможно, будем думать иначе.

Существуют две причины для выбора базы данных NoSQL: производительность работы программиста и эффективность доступа к данным. В разных ситуациях эти условия могут усиливать друг друга или противоречить друг другу. Их трудно обеспечить на ранних стадиях проекта, поскольку проблему выбора модели данных трудно абстрагировать так, чтобы со временем ее можно было заменить другой.

15.1. Производительность работы программиста

Поговорите с любым программистом, работающим с промышленным приложением, и вы почувствуете его неудовлетворение от работы с реляционными базами данных. Информация обычно собирается и отображается в терминах агрегатов, но при этом она должна трансформироваться в отношения, чтобы ее можно было сохранить. Эта задача лишь внешне кажется легкой; на протяжении 1990-х годов многие разработчики проектов испытывали сложности, пытаясь создать свои объектно-реляционные отображения.

В 2000-х годах появились популярные каркасы ORM, такие как Hibernate, iBATIS и Rails Active Record, которые существенно облегчили работу программистов, но не устранили проблему. Каркасы ORM представляют собой "протекающую абстракцию", — всегда находятся ситуации, требующие дополнительного внимания, особенно, если требуется достичь приличной производительности.

В этой ситуации агрегатно-ориентированные базы данных могли показаться привлекательным решением. Мы можем отказаться от каркасов ORM и хранить агрегаты так же естественно, как и работать с ними. Мы видели несколько проектов, которые достигли ощутимого преимущества благодаря переходу на агрегатно-ориентированные базы данных.

Графовые базы данных упрощают работу иначе. Реляционные базы данных неудобны для работы с данными, имеющими много отношений. Графовая база данных предлагает более естественный интерфейс для хранилища таких данных и возможности для выполнения запросов, предназначенных именно для таких структур.

Все виды систем лучше приспособлены для работы с неоднородными данными. Если вы пытаетесь преодолеть строгую схему, чтобы получить возможность хранить специальные поля, то неструктурированные базы данных NoSQL могут значительно облегчить ваше положение.

Существует несколько основных причин, по которым программная модель базы данных NoSQL может повысить производительность работы вашей команды. Сначала следует выяснить, что должно делать ваше программное обеспечение. Рассмотрите его текущие функциональные возможности и определите, насколько они согласованы с использованием данных. Сделав это, можете начинать анализ конкретной модели данных, которую считаете подходящей, т.е. позволяющей упростить программирование.

При этом следует помнить, что многовариантная персистентность предусматривает использование разных способов хранения данных. Разные модели могут соответствовать разным аспектам ваших данных. Таким образом, разные данные следует хранить в разных базах данных. Использование нескольких баз данных сложнее, чем одной, но преимущества хорошей согласованности в целом могут принести положительный эффект.

При оценке согласованности функций и модели данных особое внимание следует уделить проблемным ситуациям. Возможно, со многими агрегатами ваша система будет работать хорошо, а с некоторыми нет. Наличие нескольких функций, не соответствующих модели, не является причиной для отказа от модели — трудности, связанные с плохим согласованием, могут компенсироваться общим положительным результатом. Однако всегда полезно выявить проблемные места и помнить о них.

Анализируя функциональные возможности и оценивая необходимость доступа к своим данным, вы столкнетесь с несколькими альтернативными вариантами выбора базы данных. Это является отправной точкой проекта, но на следующем этапе необходимо приступать к реальной разработке программного обеспечения. Выберите несколько функций и реализуйте их, уделив внимание тому, насколько просто они используют рассматриваемую технологию. В этой ситуации целесообразно реализовать одни и те же функции для работы с разными базами данных, чтобы увидеть, какая из них окажется лучше остальных. Люди часто сопротивляются таким советам, так как никому не нравится писать программы, которые потом придется выбросить. И все же это эффективный путь для оценки эффективности конкретного каркаса.

К сожалению, пока не существует правильного показателя производительности программирования. Мы не можем точно оценить результат. Даже если вы создали совершенно одинаковые функции, производительность программирования невозможно оценить точно, потому что их повторная реализация оказывается проще, чем первая, а мы не можем себе позволить, чтобы разные команды программистов писали совершенно одинаковые программы. Мы можем лишь предоставить людям высказать свое мнение. Большинство разработчиков могут почувствовать, насколько одна среда производительнее другой. Несмотря на субъективность такой оценки и возможные разногласия

между членами одной команды, это самый лучший способ оценки, который у нас есть. В конце концов, выбор остается за командой, которая будет реализовывать проект.

Пытаясь правильно оценить производительность баз данных, важно выявить проблемные места. Команда программистов может легко пройти по одному пути и столкнуться с трудностями на другом. Это позволит получить общее представление.

Этот подход имеет несколько недостатков. Часто трудно составить правильное мнение о технологии, не поработав с ней много месяцев. Такое долгое оценивание нельзя признать эффективным с экономической точки зрения. Но, как часто бывает в жизни, приходится ограничиваться максимально точной оценкой, которую мы можем достичь, помнить о ее недостатках и смириться с этим. Важно лишь понимать, что основой вашего решения должна стать как можно более реалистичная оценка сложности программирования. Даже неделя работы с определенной технологией может выявить такие вещи, о которых вы не узнаете из сотен презентаций поставщиков.

15.2. Эффективность доступа к данным

Популярность баз NoSQL объясняется быстрым доступом к многочисленным данным. С появлением крупных веб-сайтов возникла необходимость в горизонтальном масштабировании и работе на больших кластерах. Первые базы данных NoSQL были предназначены для эффективной работы именно с такими архитектурами. Пользователи других приложений, в которых часто использовались еще более крупные объемы данных, выбирали эту технологию также из-за того, что она обеспечивала быстрый доступ к данным.

Существует много факторов, благодаря которым базы данных NoSQL в разных ситуациях обеспечивают более быстрый доступ к данным, чем реляционные базы. Агрегатно-ориентированные базы данных могут намного быстрее считывать или извлекать агрегаты по сравнению с реляционными базами, в которых данные разбросаны по многим таблицам. Более простая фрагментация и репликация на кластерах обеспечивает горизонтальное масштабирование. Графовая база данных может извлекать тесно связанные данные намного быстрее, чем реляционная.

Исследуя базы данных NoSQL с точки зрения производительности, важнее всего выбрать правильный сценарий. Размышляя о преимуществах базы данных, можно написать короткий список, но единственный способ правильно оценить производительность — измерить ее во время работы.

При оценке производительности труднее всего разработать реалистичные тесты. Вы не можете создать реальную систему, поэтому необходимо выбрать репрезентативное подмножество. Однако важно, чтобы это подмножество было как можно более типичным. Нельзя, чтобы производительность базы данных, предназначенной для работы с сотнями параллельных пользователей, оценивалась по тесту, в котором участвует только один клиент. Кроме того, необходимо создать репрезентативные нагрузки и типичные объемы данных.

При создании публичного веб-сайта может оказаться трудным создать набор тестов с высокой нагрузкой. В таком случае целесообразно использовать облачные вычислительные ресурсы для генерации нагрузки и создания тестового кластера. Гибкая природа облака оказывается очень удобной для создания краткосрочных проектов по оценке производительности.

Если вы не собираетесь проверять все условия, в которых будет работать ваше приложение, все же необходимо создать репрезентативное подмножество таких ситуаций. Выберите наиболее типичные сценарии, сильно зависящие от производительности системы, а также сценарии, мало подходящие для вашей модели данных. Последние помогут вам понять риски, возникающие, когда вы выходите за пределы обычных сценариев.

Выбор объема тестирования на ранних этапах проекта может оказаться сложным, поскольку в этот момент еще не известно, насколько крупным окажется проект. Однако для дальнейшей работы нужна какая-то основа, поэтому ее необходимо выявить и обсудить со всеми коллегами. Выявление такой основы снижает шансы того, что разные люди будут иметь разные идеи о том, что значит "высокая загрузка при чтении". Это также поможет вам выявить потенциальные проблемы, которые могут возникнуть при отклонении от первоначальных предположений. Без явной формулировки таких предположений легко оказаться в ситуации, когда их нарушение остается незамеченным.

15.3. Продолжение традиций

Технология NoSQL является жизнеспособной во многих ситуациях — в противном случае мы не стали бы тратить несколько месяцев на эту книгу. Но мы понимаем также, что во многих случаях, а в действительности в большинстве случаев, для вас было бы лучше придерживаться традиционных реляционных баз данных.

Реляционные базы данных хорошо известны; вы легко найдете специалистов, имеющих опыт их использования. Эти базы достигли зрелости, поэтому вряд ли способны преподнести неожиданные неприятные сюрпризы, в отличие от новых технологий. Существует множество инструментов для разработки реляционных баз данных, которыми можно воспользоваться. Кроме того, при использовании реляционных баз данных не возникают политические проблемы, связанные с принятием необычных решений, — выбор новой технологии всегда вносит риск возникновения проблем, усложняющих жизнь.

Итак, подытоживая сказанное, мы склоняемся к той точке зрения, что базы данных NoSQL следует выбирать только в тех случаях, когда они имеют явное преимущество над реляционными базами.

Нет ничего зазорного в том, чтобы выполнить оценку легкости программирования и эффективности доступа к данным, не обнаружить никаких явных преимуществ и выбрать традиционные реляционные базы данных. Мы полагаем, что во многих случаях использование баз данных NoSQL может принести пользу, но "во многих" не значит "во всех" или "в большинстве случаев".

15.4. Подстраховка

Одна из самых больших сложностей при выработке рекомендаций по выбору способа хранения данных заключается в том, что у нас пока мало сведений. На момент написания книги было всего несколько проектов, в рамках которых испытывались эти технологии, и все аргументы "за" и "против" пока неизвестны.

В такой неопределенной ситуации целесообразно инкапсулировать свой выбор базы данных, т.е. хранить коды базы в разделе, который относительно легко изменить, если вы решите изменить свое решение. Классический способ сделать это — использовать явный слой хранилища данных в своем приложении с помощью шаблонов проектирования, таких как Data Mapper и Repository [Fowler PoEAA]. Такой слой инкапсуляции требует затрат, особенно, если вы не уверены, что на самом деле будете использовать другие модели данных, например, "ключ–значение", а не графовую модель. Что еще хуже, у нас пока нет опыта инкапсуляции таких слоев между очень разными хранилищами данных.

Итак, наш совет: инкапсулируйте по умолчанию, но не забывайте о стоимости такого решения. Если это приводит к возрастанию нагрузки на систему и потере некоторых полезных функций базы данных, то целесообразно использовать такую базу данных, которая имеет такие свойства. Возможно, именно такая информация станет решающей при выборе базы данных, и в инкапсуляции слоя в таком случае не будет необходимости.

Это еще один аргумент в пользу декомпозиции такого слоя базы данных на сервисы, инкапсулирующие хранилище данных (см. раздел 13.3 "Использование сервисов при работе с хранилищем данных"). Помимо уменьшения связанности разных сервисов, это приносит дополнительное преимущество за счет упрощения последующей замены базы данных при необходимости. Это разумный подход, даже если вы будете всегда использовать одну и ту же базу данных, — если что-то пойдет не так, как ожидалось, вы можете постепенно переключать сервисы, концентрируя основное внимание на проблематичных сервисах.

Этот совет остается в силе, даже если вы будете по-прежнему использовать реляционные базы данных. Инкапсуляция сегментов базы данных в виде сервисов облегчит переключение частей вашего хранилища данных на технологию NoSQL по мере того, как она будет развиваться, а ее преимущества станут более очевидными.

15.5. Резюме

- Существуют две главные причины для перехода на технологию NoSQL.
  - Повышение производительности работы программистов путем использования базы данных, которая точнее соответствует потребностям приложения.
  - Повышение эффективности доступа к данным с помощью обработки более крупных объемов данных, уменьшения времени отклика и улучшения пропускной способности.

- Очень важно правильно оценить ожидаемое повышение производительности работы программиста и/или эффективность доступа к данным за счет применения технологии NoSQL.
- Инкапсуляция сервисов позволяет изменять технологии хранения данных при необходимости. Разделение частей приложений на сервисы позволяет также внедрить технологию NoSQL в существующее приложение.
- Большинство приложений, особенно не имеющих стратегического характера, должны по-прежнему использовать реляционную технологию, по крайней мере, пока технология NoSQL не достигнет зрелости.

15.6. Заключительные мысли

Мы надеемся, что наша книга оказалась понятной читателям. Когда мы начинали ее писать, мы столкнулись с недостатком информации о технологии NoSQL. В ходе написания книги мы сами выполнили обзор доступной информации, и это было интересно. Надеемся, что наше изложение было достаточно лаконичным и вместе с тем не менее интересным.

Возможно, вы задумались об использовании технологии NoSQL в своем проекте. Если это так, то следует учесть, что книга представляет собой всего лишь первый шаг на вашем пути освоения этой технологии. Рекомендуем вам загрузить несколько баз данных и поработать с ними, потому что мы твердо убеждены, что понять технологию можно, лишь работая с нею и выявляя ее сильные стороны и неизбежные проколы, которые никогда не отражаются в документации.

Думаем, что большинство людей, включая читателей нашей книги, еще не использовали базы данных NoSQL. Это новая технология, и мы пока еще только начинаем понимать, когда и как ее использовать. Но в мире программного обеспечения ситуация изменяется быстрее, чем можно предсказать, поэтому следует внимательно следить за всем, что в нем происходит.

Мы надеемся, что вы прочитаете другие книги и статьи на эту тему. Полагаем, что после опубликования нашей книги появятся еще более глубокие материалы по технологии NoSQL, поэтому пока не можем посоветовать вам для чтения ничего конкретного. Мы активно работаем в веб, и наши актуальные мысли о технологии NoSQL можно узнать на веб-сайтах `www.sadalage.com` и `http://martinfowler.com/nosql.html`.

Библиография

[Agile Methods] www.agilealliance.org.

[Amazon's Dynamo] www.allthingsdistributed.com/2007/10/amazons_dynamo.html.

[Amazon DynamoDB] http://aws.amazon.com/dynamodb.

[Amazon SimpleDB] http://aws.amazon.com/simpledb.

[Ambler and Sadalage] Ambler, Scott and Pramodkumar Sadalage. *Refactoring Databases: Evolutionary Database Design.* Addison-Wesley. 2006. ISBN 978-0321293534.

[Berkeley DB] www.oracle.com/us/products/database/berkeley-db.

[Blueprints] https://github.com/tinkerpop/blueprints/wiki.

[Brewer] Brewer, Eric. *Towards Robust Distributed Systems.* www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.

[Cages] http://code.google.com/p/cages.

[Cassandra] http://cassandra.apache.org.

[Chang etc.] Chang, Fay, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh,
Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and
Robert E. Gruber. *Bigtable: A Distributed Storage System for Structured Data.*
http://research.google.com/archive/bigtable-osdi06.pdf.

[CouchDB] http://couchdb.apache.org.

[CQL] www.slideshare.net/jericevans/cql-sql-in-cassandra.

[CQRS] http://martinfowler.com/bliki/CQRS.html.

[C-Store] Stonebraker, Mike, Daniel Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. *C-Store: A Columnoriented DBMS.* http://db.csail.mit.edu/projects/cstore/vldb.pdf.

[Cypher] http://docs.neo4j.org/chunked/1.6.1/cypher-query-lang.html.

[Daigneau] Daigneau, Robert. *Service Design Patterns.* Addison-Wesley. 2012. ISBN 032154420X.

[DBDeploy] http://dbdeploy.com.

[DBMaintain] www.dbmaintain.org.

[Dean and Ghemawat] Dean, Jeffrey and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters.* http://static.usenix.org/event/osdi04/tech/full_papers/dean/dean.pdf.

[Dijkstra's] http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

[Evans] Evans, Eric. *Domain-Driven Design.* Addison-Wesley. 2004. ISBN 0321125215. (Русский перевод: Эванс Э. Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. — М.: Вильямс, 2011. — 448 с.)

[FlockDB] https://github.com/twitter/flockdb.

[Fowler DSL] Fowler, Martin. *Domain-Specific Languages.* Addison-Wesley. 2010. ISBN 0321712943. (Русский перевод: Фаулер М.: Предметно-ориентированные языки программирования. — М.: Вильямс, 2011. — 576 с.)

[Fowler lmax] Fowler, Martin. *The LMAX Architecture.* http://martinfowler.com/articles/lmax.html.

[Fowler PoEAA] Fowler, Martin. *Patterns of Enterprise Application Architecture.* Addison-Wesley. 2003. ISBN 0321127420. (Русский перевод: Фаулер М. Шаблоны корпоративных приложений. — М.: Вильямс, 2012. — 544 с.)

[Fowler UML] Fowler, Martin. *UML Distilled.* Addison-Wesley. 2003. ISBN 0321193687.

[Gremlin] https://github.com/tinkerpop/gremlin/wiki.

[Hadoop] http://hadoop.apache.org/mapreduce.

[HamsterDB] http://hamsterdb.com.

[Hbase] http://hbase.apache.org.

[Hector] https://github.com/rantav/hector.

[Hive] http://hive.apache.org.

[Hohpe and Woolf] Hohpe, Gregor and Bobby Woolf. *Enterprise Integration Patterns.* Addison-Wesley. 2003. ISBN 0321200683.

[HTTP] Fielding, R., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol—HTTP/1.1.* www.w3.org/Protocols/rfc2616/rfc2616.html.

[Hypertable] http://hypertable.org.

[Infinite Graph] www.infinitegraph.com.

[JSON] http://json.org.

[LevelDB] http://code.google.com/p/leveldb.

[Liquibase] www.liquibase.org.

[Lucene] http://lucene.apache.org.

[Lynch and Gilbert] Lynch, Nancy and Seth Gilbert. *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services.* http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf.

[Memcached] http://memcached.org.

[MongoDB] www.mongodb.org.

[Monitoring] www.mongodb.org/display/DOCS/MongoDB+Monitoring+Service.

[MyBatis Migrator] http://mybatis.org.

[Neo4J] http://neo4j.org.

[NoSQL Debrief] http://blog.oskarsson.nu/post/22996140866/nosql-debrief.

[NoSQL Meetup] http://nosql.eventbrite.com.

[Notes Storage Facility] http://en.wikipedia.org/wiki/IBM_Lotus_Domino.

[OpsCenter] www.datastax.com/products/opscenter.

[OrientDB] www.orientdb.org.

[Oskarsson] *Private Correspondence.*

[Pentaho] www.pentaho.com.

[Pig] http://pig.apache.org.

[Pritchett] www.infoq.com/interviews/dan-pritchett-ebay-architecture.

[Project Voldemort] http://project-voldemort.com.

[RavenDB] http://ravendb.net.

[Redis] http://redis.io.

[Rekon] https://github.com/basho/rekon.

[Riak] http://wiki.basho.com/Riak.html.

[Solr] http://lucene.apache.org/solr.

[Strozzi NoSQL] www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL.

[Tanenbaum and Van Steen] Tanenbaum, Andrew and Maarten Van Steen. *Distributed Systems.* Prentice-Hall. 2007. ISBN 0132392275.

[Terrastore] http://code.google.com/p/terrastore.

[Vogels] Vogels, Werner. *Eventually Consistent—Revisited.* www.allthingsdistributed.com/2008/12/eventually_consistent.html.

[Webber Neo4J Scaling] http://jim.webber.name/2011/03/22/ef4748c3-6459-40b6-bcfa-818960150e0f.aspx.

[ZooKeeper] http://zookeeper.apache.org.

Предметный указатель

Необходимость обрабатывать все более крупные объемы данных является одним из факторов, влияющих на внедрение нового класса нереляционных баз данных NoSQL. Сторонники баз NoSQL утверждают, что их можно использовать для создания более производительных, легче масштабируемых и проще программируемых систем.

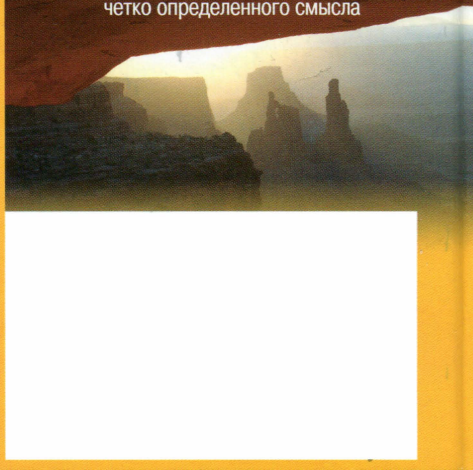
Эта книга – краткое, но полное введение в быстро развивающуюся технологию NoSQL. Прамодкумар Дж. Садаладж и Мартин Фаулер объясняют, как работают базы данных NoSQL, и демонстрируют, в каких ситуациях они могут стать более успешной альтернативой традиционным системам RDMBS. Авторы излагают материал в быстром темпе, знакомя читателей с критериями, которые необходимо применять, чтобы принять правильное решение, стоит ли использовать базы NoSQL и какие технологии следует при этом выбирать.

Первая часть книги посвящена основным концепциям, включая неструктурированные модели данных, агрегаты, новые модели распределения, теорему CAP и отображение–свертку. Во второй части авторы исследуют архитектурные и проектные вопросы, связанные с реализацией баз данных NoSQL. Они также описывают реалистичные сценарии использования, демонстрирующие работу баз данных NoSQL и возможности баз Riak, MongoDB, Cassandra и Neo4j.

Кроме того, основываясь на новаторской работе Прамодкумара Садаладжа, авторы книги показывают, как реализовать эволюционное проектирование на основе миграции схем – важный метод, необходимый для применения баз данных NoSQL. Книга завершается описанием новой эры многовариантной персистентности, открытой благодаря технологии NoSQL. В этом мире сосуществуют разнообразные базы данных, и архитектор может выбирать технологию, наилучшим образом подходящую для обеспечения любого вида доступа к данным.

Основные темы книги:

- Оценка применимости корпоративных приложений NoSQL
- Архитектурные компромиссы, связанные с развертыванием баз данных NoSQL
- Использование технологии NoSQL для упрощения разработки и предотвращения проблем, связанных с отображением данных между структурами в памяти и системами RDBMS
- Сравнение современных возможностей баз данных NoSQL
- Исследование языков запросов: CQL и Cypher
- Эффективность управления, надежность, доступность и способность к восстановлению
- Использование технологии NoSQL для гибкой разработки программного обеспечения
- Применение технологии NoSQL для управления поиском и извлечением метаданных, анализа текстов, организации социальных сетей, проведения бизнес-анализа и выполнения финансовых операций
- Кластеризация баз данных NoSQL для более дешевого решения проблем, связанных с обработкой крупных объемов данных
- Применение теоремы CAP для анализа согласованности, доступности и времени ожидания
- Анализ возможностей, которые метод отображения-свертки открывает для параллельных вычислений на кластере
- Почему термин NoSQL не имеет четко определенного смысла



ПРАМОДКУМАР ДЖ. САДАЛАДЖ, главный консультант компании ThoughtWorks, занимается редким делом – наведением мостов между специалистами в области баз данных и разработчиками приложений. Он регулярно консультирует клиентов, испытывающих особенно большие сложности при обработке данных и нуждающихся в новых технологиях и методах. Садаладж создал новаторский метод, позволяющий разрабатывать реляционные базы данных эволюционным путем с помощью контролируемой миграции схем, сопровождающейся контролем версий. Вместе со Скоттом Эмблером (Scott Ambler) он написал книгу *Рефакторинг баз данных*.

МАРТИН ФАУЛЕР, главный научный сотрудник компании ThoughtWorks, занимается исследованием оптимальных способов разработки программного обеспечения и повышения производительности разработчиков. Он – автор книг *Шаблоны корпоративных приложений*; *UML. Основы, Третье издание*; *Предметно-ориентированные языки программирования* (в соавторстве с Ребеккой Парсонс); *Рефакторинг. Улучшение существующего кода* (в соавторстве с Кентом Беком, Джоном Брантом и Уильямом Апдайком). Все они изданы компанией Addison-Wesley и переведены на русский язык.

informit.com/aw
Макет обложки – Чути Прасертит (Chuti Prasertith)
Фотографии на обложке – Мартин Фаулер

вильямс

Издательский дом "Вильямс":
http://www.williamspublishing.com

Addison-Wesley
Pearson Education