

The
Pragmatic
Programmers

Семь баз данных за семь недель

Введение в современные базы данных
и идеологию NoSQL

Эрик Редмонд
Джим Р. Уилсон



Эрик Редмонд
Джим. Р. Уилсон

Семь баз данных за семь недель

Введение в современные базы данных
и идеологию NoSQL

под редакцией Жаклин Картер

Seven Databases in Seven Weeks

A Guide to Modern Databases
and the NoSQL Movement

Eric Redmond
Jim R. Wilson

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Семь баз данных за семь недель

Введение в современные базы данных
и идеологию NoSQL

Эрик Редмонд,
Джим. Р. Уилсон



Москва, 2013

УДК 004.6
ББК 32.973.26
Р33

Р33 Эрик Редмонд, Джим. Р. Уилсон

Семь баз данных за семь недель. Введение в современные базы данных и идеологию NoSQL. Под редакцией Жаклин Картер / Пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2013. – 384с.: ил.

ISBN 978-5-94074-866-3

В книге описаны большинство из современных баз данных с открытым исходным кодом: Redis, Neo4J, CouchDB, MongoDB, HBase, PostgreSQL и Riak. Для каждой базы приведены примеры работы с реальными данными, демонстрирующие основные идеи и сильные стороны.

Эта книга прольет свет на сильные и слабые стороны каждой из семи баз данных и научит вас выбирать ту, которая лучше отвечает требованиям.

Издание предназначено для программистов разной квалификации, использующих базы данных в своей профессиональной деятельности.

УДК 004.6
ББК 32.973.26

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-93435-692-0 (англ.)
ISBN 978-5-94074-866-3 (рус.)

© 2012 Pragmatic Programmers, LLC.
© Оформление, перевод на русский язык
ДМК Пресс, 2013



СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ	13
БЛАГОДАРНОСТИ	15
ВСТУПЛЕНИЕ	17
Почему именно семь баз данных?	17
Что есть в этой книге?	17
Чего нет в этой книге?	18
Это не руководство по установке	18
Руководство администратора? Пожалуй, нет	19
Замечание для пользователей Windows	19
Примеры кода и соглашения	19
Сетевые ресурсы	20
Глава 1. Введение	21
1.1. Все начинается с вопроса	21
1.2. Жанры	23
Реляционные СУБД	24
Хранилища ключей и значений	25
Столбцовые базы данных	26
Документно-ориентированные базы данных	27
Графовые базы данных	28
Многостороннее хранение	28
1.3. Вперед и вверх	29
Глава 2. PostgreSQL	30
2.1. Произносится Post-greS-Q-L	30
2.2. День 1: отношения, операции CRUD и соединения	32
Введение в SQL	33

Быстрый поиск с применением индексов	41
День 1: итоги	43
День 1: домашнее задание	44
2.3. День 2: более сложные запросы, код и правила	45
Агрегатные функции	45
Оконные функции	48
Транзакции	49
Хранимые процедуры	50
Триггеры	52
Представление о мире	54
Правила	55
Создание сводных таблиц с помощью <code>crosstab()</code>	57
День 2: итоги	59
День 2: домашнее задание	59
2.4. День 3: полнотекстовый поиск и многомерные кубы	60
Нечеткий поиск	62
Полнотекстовый поиск	65
День 3: итоги	75
День 3: домашнее задание	75
2.5. Резюме	75
Сильные стороны PostgreSQL	76
Слабые стороны PostgreSQL	77
Перед расставанием	77
Глава 3. Riak	78
3.1. Riak дружит с веб	79
3.2. День 1: CRUD, ссылки и типы MIME	80
Лучше REST может быть только REST (или как завивать локоны)	82
Ссылки	85
Типы MIME в Riak	89
День 1: итоги	90
День 1: домашнее задание	90
3.3. День 2: <code>mapreduce</code> и кластеры серверов	91
Скрипт для загрузки данных	91
Введение в Mapreduce	92
Mapreduce в Riak	95
О согласованности и долговечности	101
День 2: итоги	109
День 2: домашнее задание	109

3.4. День 3: разрешение конфликтов и расширение Riak ..	110
Разрешение конфликтов с помощью векторных часов.....	110
Расширение Riak.....	117
День 3: итоги.....	121
День 3: домашнее задание.....	122
3.5. Резюме ..	122
Сильные стороны Riak.....	123
Слабые стороны Riak.....	123
Riak и теорема CAP ..	123
Перед расставанием ..	124
Глава 4. HBase.....	125
4.1. Введение в HBase ..	126
4.2. День 1: операции CRUD и администрирование таблиц ..	127
Конфигурирование HBase ..	128
Оболочка HBase ..	129
Создание таблицы ..	129
Вставка, обновление и выборка данных ..	131
Добавление данных из программы.....	136
День 1: итоги.....	137
День 1: домашнее задание.....	138
4.3. День 2: работа с «большими данными» ..	139
Импорт данных, выполнение скриптов ..	139
Потоковая загрузка XML.....	140
Загрузка википедии ..	141
Сжатие и фильтры Блума.....	143
Контакт? Есть контакт!.....	143
Знакомство с регионами и мониторингом места на диске.....	145
Опрос регионов ..	146
Сканирование одной таблицы для построения другой.....	149
Построение сканера.....	150
Запуск скрипта.....	152
Исследование результатов.....	153
День 2: итоги.....	154
День 2: домашнее задание.....	155
4.4. День 3: переходим в облако ..	156
Разработка «бережливого» приложения для HBase ..	156
Введение в Whirr ..	160
Подготовка к работе с EC2 ..	160
Подготовка Whirr ..	161

Настройка кластера	162
Запуск кластера	163
Подключение к кластеру.....	163
Уничтожение кластера	164
День 3: итоги.....	164
День 3: домашнее задание	165
4.5. Резюме	166
Сильные стороны HBase.....	166
Слабые стороны HBase	167
HBase и теорема CAP	167
Перед расставанием	168
Глава 5. MongoDB	169
5.1. Монстр.....	169
5.2. День 1: операции CRUD и вложенность	171
Поработаем с командной строкой	171
JavaScript	173
Чтение: продолжаем изучать Mongo.....	175
Копнем глубже	177
Обновление	181
Ссылки	183
Удаление	184
Функциональные критерии.....	185
День 1: итоги.....	186
День 1: домашнее задание	186
5.3. День 2: индексирование, группировка, mapreduce	187
Индексирование: когда быстродействия не хватает	187
Агрегированные запросы	191
Команды на стороне сервера	194
Mapreduce (и Finalize)	197
День 2: итоги.....	201
День 2: домашнее задание	201
5.4. День 3: наборы реплик, сегментирование, пространственные данные и GridFS	201
Наборы реплик.....	202
Сегментирование.....	206
Пространственные запросы	208
GridFS	210
День 3: итоги.....	211
День 3: домашнее задание	211
5.5. Резюме	212

Сильные стороны Mongo	212
Слабые стороны Mongo	212
Перед расставанием	213

Глава 6. CouchDB 214

6.1. Располагайтесь на кушетке	214
Сравнение CouchDB с MongoDB	215
6.2. День 1: операции CRUD, Futon и снова cURL	215
Знакомство с Futon	216
Выполнение операций CRUD с помощью REST-интерфейса и cURL	219
Чтение документа с помощью GET	220
Создание документа с помощью POST	221
Обновление документа с помощью PUT	222
Удаление документа с помощью DELETE	223
День 1: итоги	223
День 1: домашнее задание	223
6.3. День 2: создание и опрос представлений	224
Доступ к документам через представления	224
Создание первого представления	226
Сохранение представления в виде проектного документа	229
Поиск исполнителей по имени	229
Поиск альбомов по названию	230
Опрос представлений исполнителей и альбомов	231
Импорт данных в CouchDB с помощью программы на Ruby	233
День 2: итоги	238
День 2: домашнее задание	238
6.4. День 3: более сложные представления, Changes API и репликация данных	239
Создание более сложных представлений с помощью редукторов	239
Отслеживание изменений в CouchDB	243
Непрерывное отслеживание изменений	249
Фильтрация изменений	250
Репликация данных в CouchDB	252
День 3: итоги	256
День 3: домашнее задание	256
6.5. Резюме	257
Сильные стороны CouchDB	257
Слабые стороны CouchDB	258
Перед расставанием	258

Глава 7. Neo4J..... 259

7.1. Neo4J дружит с доской.....	259
7.2. День 1: графы, Groovy и операции CRUD	261
Веб-интерфейс Neo4j.....	262
Neo4j и Gremlin	264
Конвейеры	267
Конвейер и вершина	269
Бессхемная социальная сеть.....	270
Дорога меряется шагами	271
Обновляем, удаляем, стираем	278
День 1: итоги.....	279
День 1: домашнее задание.....	279
7.3. День 2: REST, индексы и алгоритмы.....	279
REST-интерфейс.....	279
Интересные алгоритмы.....	286
День 2: итоги.....	292
День 2: домашнее задание.....	292
7.4. День 3: распределенность и высокая доступность	293
Транзакции	293
Высокая доступность	294
HA-кластер.....	295
Резервное копирование	301
День 3: итоги.....	302
День 3: домашнее задание.....	302
7.5. Резюме	302
Сильные стороны Neo4j.....	303
Слабые стороны Neo4j	303
Neo4j и теорема CAP	304
Перед расставанием	304

Глава 8. Redis..... 305

8.1. Хранилище сервера структур данных	305
8.2. День 1: операции CRUD и типы данных.....	306
Приступая к работе	307
Транзакции	309
Составные типы данных	309
Блокирующие списки	313
Диапазоны	316
Пространства имен	319
И это еще не всё.....	320

День 1: итоги.....	321
День 1: домашнее задание.....	321
8.3. День 2: более сложные применения, распределенные вычисления.....	321
Простой интерфейс	322
Информация о сервере	325
Настройка Redis	325
Репликация главный-подчиненный.....	330
Загрузка данных.....	330
Кластер Redis	333
Фильтры Блума	334
SETBIT и GETBIT	337
День 2: итоги.....	338
День 2: домашнее задание.....	338
8.4. День 3: комбинирование с другими базами данных....	339
Служба многостороннего хранения.....	339
Заполнение данными	341
Фаза 1: трансформация данных	342
Фаза 2: вставка в каноническую систему.....	344
Хранилище связей	347
Веб-служба	349
Развитие веб-службы.....	351
День 3: итоги.....	352
День 3: домашнее задание.....	353
8.5. Резюме	353
Сильные стороны Redis.....	353
Слабые стороны Redis.....	354
Перед расставанием	354
Глава 9. Подводя итоги	356
9.1. Снова о жанрах	356
Реляционные базы данных	356
Хранилища ключей и значений	357
Столбцовые базы данных	358
Документные базы данных	359
Графовые базы данных	360
9.2. Как сделать выбор?.....	361
9.3. В каком направлении двигаться дальше?	362
ПРИЛОЖЕНИЕ 1. Сравнительный обзор баз данных	363

ПРИЛОЖЕНИЕ 2. Теорема CAP	367
A2.1. Согласованность в конечном счете.....	368
A2.2. CAP на практике	369
A2.3. Компромиссный выбор задержки.....	370
СПИСОК ЛИТЕРАТУРЫ	371
ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ	372



ПРЕДИСЛОВИЕ

Сидя в креслах подъемника Бивер Ран Суперчер в Брикенридже, штат Колорадо, мы недоумевали, где же свежий снег. Склоны горы содержались в безукоризненном порядке, но везде условия огорчали неизменным однообразием. Без свежевывающего снега на настоящее удовольствие рассчитывать было нечего.

В 1994, работая в лаборатории IBM по разработке баз данных в Остине, я испытывал примерно такие же чувства. В Техасском университете в Остине я изучал объектно-ориентированные базы данных, так как полагал, что после десятилетнего доминирования реляционных СУБД у них появился реальный шанс пустить корни. Но и следующее десятилетие не принесло ничего, кроме все той же реляционной модели. Я удрученно наблюдал, как Oracle, IBM, а позднее продукты с открытым исходным кодом и, прежде всего, MySQL раскидывали ветви, полностью лишая солнечного света любые всходы, пытающиеся взрасти на плодородной почве внизу.

Пользовательские интерфейсы эволюционировали – зеленые буквы на черном экране сменились клиент-серверными, а потом и Интернет-приложениями, но код работы с реляционным уровнем был отмечен печатью унылого постоянства, пусть и доведенного до совершенства. Все ждали, когда же выпадет свежий снег.

И наконец начался снегопад. Поначалу снега не хватало даже для того, чтобы запорошить вчерашнюю лыжню, но постепенно пурга усиливалась, меняла ландшафт, неся с собой разнообразие и радость катания, которого мы так жаждали. Как раз в тот год я осознал, что мир баз данных тоже накрыло покрывалом свежевывающего снега. Разумеется, реляционные СУБД никуда не делись, и продукты с открытым исходным кодом по-прежнему способны решать поразительно разнообразные задачи. В них появилась кластеризация, полнотекстовый и даже нечеткий поиск. Но этот подход перестал быть единственным. За год я не создал ни одного решения, которое было бы целиком реляционным. В тот период я использовал документо-ориентированную базу данных и два хранилища ключей и значений.

Реляционные базы данных лишились монополии на гибкость и даже на масштабируемость. Для приложений, над которыми мы работали, существовали более подходящие модели – проще, быстрее и надежнее. Для человека, который десять лет занимался в Остинском подразделении IBM базами данных в интересах как самой лаборатории, так и заказчиков такое развитие событий было просто ошеломительным. В этой книге мы на примерах рассмотрим представительную выборку наиболее важных прорывов в области баз данных, цель которых – поддержать разработку Интернет-приложений. Вы узнаете о хранилищах ключей и значений – потрясающей масштабируемости и надежности Riak и красивом механизме запросов в Redis. Мы поговорим о достижениях сообщества столбцовых баз данных – мощи HBase, близкого родственника реляционных СУБД. Документо-ориентированные базы данных мы рассмотрим на примере прекрасно масштабируемой системы MongoDB, позволяющей элегантно хранить документы с глубокой вложенностью. Вы познакомитесь также с подходом Neo4J к графовым базам данных, которые позволяют быстро обходить связи.

Чтобы повысить свою квалификацию как программиста или администратора баз данных, необязательно использовать все эти продукты. Эрик Редмонд и Джим Уилсон станут вашими проводниками в этом удивительном путешествии, каждый этап которого сделает вас образованнее и принесет бесценные знания, так необходимые современному профессиональному разработчику программного обеспечения. Вы узнаете о сильных сторонах и ограничениях каждой из рассматриваемых платформ. Вы увидите, в каком направлении движется индустрия, и поймете, какие побудительные мотивы стоят за этим движением.

Приятного путешествия.

*Брюс Тейт,
Автор книги «Seven Languages
in Seven Weeks»
Остин, Техас, май 2012*



БЛАГОДАРНОСТИ

Чтобы справиться с книгой такого объема и широты тематики, двух авторов недостаточно. Требуются усилия многих одаренных людей с острым взглядом, способных заметить многочисленные ошибки и внести ценный вклад, поведав о тонких деталях технологий.

Мы хотели бы поблагодарить всех, кто жертвовал своим временем и делился опытом (порядок перечисления произвольный): Ян Деес (Ian Dees), Марк Филлипс (Mark Phillips), Ян Ленхардт (Jan Lenhardt), Роберт Стэм (Robert Stam), Олег Бартунов, Дэйв Пэррингтон (Dave Purrington), Дэниэл Бретау (Daniel Bretoi), Мэтт Адамс (Matt Adams), Шон Копенхэйвер (Sean Copenhaver), Лоран Сэндс-Рэмшоу (Loren Sands-Ramshaw), Эмиль Эйфрем (Emil Eifrem), Андреас Колледжер (Andreas Kollegger).

И, наконец, спасибо Брюсу Тейту (Bruce Tate) за его опыт и рекомендации.

Мы также искренне благодарим весь коллектив издательства Pragmatic Bookshelf – за то, что они взялись за этот дерзкий проект и помогли нам в его реализации. Отдельное спасибо нашему редактору, Джеки Картер (Jackie Carter). Благодаря вашим терпеливым замечаниям книга приобрела свой настоящий вид. Спасибо всем, кто упорно работал, чтобы отшлифовать текст и исправить все наши огрехи.

И не в последнюю очередь мы хотим выразить благодарность Фредерику Дюмону (Frederic Dumont), Мэттью Флауэру (Matthew Flower), Ребекке Скиннер (Rebecca Skinner) и всем нашим требовательным читателям. Если бы не ваше неутолимое желание учиться, у нас не было бы шанса быть вам полезными.

Приносим извинения всем, кого мы забыли упомянуть, – разумеется, не по злому умыслу.

От Эрика. Дорогая Ноэль, ты не просто особенная, ты неповторима, и этим все сказано. Спасибо тебе, что сумела пережить еще одну книгу. Спасибо также создателям баз данных и их добровольным помощникам за то, что у нас есть, о чем писать и тем зарабатывать себе на жизнь.

От Джима. Прежде всего, я обязан поблагодарить свою семью – Рути за безграничное терпение и поддержку, которые согревали мне сердце; Эмму и Джимми – двух смысленных проказников, которые всегда могут рассчитывать на папину любовь. И отдельная благодарность всем невоспетым героям, которые читают IRC-каналы, форумы, списки рассылки и системы регистрации ошибки – всегда готовые помочь тем, кто в этом нуждается. Только благодаря вашей преданности движению за открытость исходного кода эти проекты и могут существовать.



ВСТУПЛЕНИЕ

Кто-то сравнил данные с сырой нефтью. Раз так, то базы данных – это месторождения, буровые установки, насосы и нефтеочистительные заводы. Данные хранятся в базах и, если вы хотите добраться до них, то должны для начала познакомиться с современным оборудованием.

Базы данных – это инструменты, средства достижения конечного результата. У каждой базы есть своя история и свой взгляд на мир. Чем глубже вы их понимаете, тем лучше оснащены для обуздания мощи, скрытой в постоянно растущем корпусе доступных данных.

Почему именно семь баз данных?

Еще в марте 2010 года мы захотели написать книгу о NoSQL-технологиях. Тогда этот термин только входил в обиход, о нем много говорили, но общего понимания еще не было. Что на самом деле означает слово *NoSQL*? Какие типы систем включать в эту категорию? Какое влияние новые технологии окажут на практику создания программ? Именно на эти вопросы мы и собирались ответить – как для самих себя, так и для других.

Прочитав книгу Брюса Тейта «Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages»¹ [Tat10], мы поняли, что он нашел правильный подход. Прогрессивная методика введения в языки нашла отклик в наших сердцах. Мы решили, что подобный способ можно применить и к базам данных, и это позволит доходчиво ответить на поставленные вопросы о технологиях NoSQL.

Что есть в этой книге?

Эта книга адресована опытным разработчикам, которые хотели бы получить общее представление о положении дел в современных тех-

1 Семь языков за семь недель. Прагматическое руководство по изучению языков программирования. *Прим. перев.*

нологиях баз данных. Опыт работы с базами данных приветствуется, хотя и необязателен.

После краткого введения следуют семь глав, посвященных семи базам данных, представляющим пять разных жанров, или стилей, упомянутых во введении: PostgreSQL, Riak, Apache HBase, MongoDB, Apache CouchDB, Neo4J и Redis.

Предполагается, что на проработку каждой главы потребуется полный выходной – три дня. Каждый день завершается упражнениями, в которых развиваются затронутые идеи и темы, а в конце каждой главы приведено резюме, где обсуждаются сильные и слабые стороны базы данных. Вы можете продвигаться быстрее или медленнее – как будет удобнее, но перед тем как продолжить чтение, важно как следует усвоить изложенные концепции. Мы старались подобрать примеры, наглядно высвечивающие характерные особенности каждой базы данных. Чтобы по-настоящему понять, что может предложить та или иная база данных, необходимо некоторое время поработать с ней, то есть засучить рукава и попрактиковаться.

Возможно, у вас возникнет искушение пропустить некоторые главы, однако имейте в виду, что книга рассчитана на последовательное чтение. Некоторые концепции, например распределение-редукция (mapreduce), подробно рассматриваются в начальных главах и лишь бегло – в последующих. Цель книги – дать полное представление о положении дел в современной индустрии баз данных, поэтому мы рекомендуем изучить все главы.

Чего нет в этой книге?

Прежде чем приступать к чтению книги, вы, наверное, захотите узнать, чего в ней нет.

Это не руководство по установке

Установка описанных в книге баз данных иногда осуществляется просто, иногда – с некоторыми сложностями, а иногда – откровенно безобразно. Для одних баз имеются готовые пакеты, другие придется компилировать из исходного кода. Кое-где мы будем давать полезные советы, но, вообще говоря, вы предоставлены сами себе. Опустив описание процедуры установки, мы смогли включить больше полезных примеров и пояснений, а ведь именно это вам и нужно, не так ли?

Руководство администратора?

Пожалуй, нет

В этой книге вы не найдете и того, что обычно входит в состав руководства по администрированию. У каждой базы данных бесчисленное множество параметров, флагов и тонкостей настройки; по большей части, все это хорошо документировано в Сети. Нас больше интересует обучение полезным идеям и полное погружение, нежели рутинные повседневные операции. Хотя характеристики работы базы данных могут зависеть от параметров – и в некоторых местах эти характеристики обсуждаются, – мы не собираемся вдаваться в тонкости всех возможных конфигураций. На это просто нет места!

Замечание для пользователей Windows

Эта книга принципиально посвящена разнообразию выбора, преимущественно из различного ПО с открытым исходным кодом на платформах *nix. Корпорация Microsoft стремится к созданию интегрированных сред, что ограничивает возможность выбора более узким набором предопределенных компонентов. Поэтому базы данных с открытым исходным кодом, которые мы здесь рассматриваем, разрабатываются пользователями (и преимущественно *для* пользователей) *nix-систем. Это не наше собственное предубеждение, а отражение текущего положения дел. Поэтому предполагается, что все учебные примеры будут запускаться из оболочки *nix. Если вы работаете в Windows, но всё же хотите поэкспериментировать, рекомендуем построить среду Cygwin². Можно также запустить виртуальную Linux-машину.

Примеры кода и соглашения

В этой книге встречается код на различных языках. Отчасти это диктуется конкретной рассматриваемой базой данных. Мы старались ограничиться только языками Ruby/JRuby и JavaScript. Мы предпочитаем командные утилиты скриптам, но, когда это нужно для решения поставленной задачи, применяем и другие языки, например PL/pgSQL (Postgres) или Gremlin/Groovy (Neo4J). Мы также рассмотрим программирование некоторых серверных приложений на языке JavaScript с применением Node.js.

² <http://www.cygwin.com/>

Если явно не оговорено противное, листинги содержат законченный код, который можно выполнить на досуге. В примерах и фрагментах синтаксические конструкции графически выделены в соответствии с правилами языка. Команды оболочки начинаются со знака \$.

Сетевые ресурсы

Ценным ресурсом является посвященная этой книге страница на сайте издательства Pragmatic Bookshelf³. Там вы найдете весь представленный в книге исходный код, а также средства обратной связи – форум сообщества и форму для отправки сообщений о найденных ошибках, где заодно можно внести предложения по изменениям в будущих изданиях этой книги.

Благодарим всех, кто готов сопровождать нас в путешествии по миру современных баз данных.

Эрик Редмонд и Джим Р. Уилсон

³ <http://pragprog.com/book/rwdata/seven-databases-in-seven-weeks>



ГЛАВА 1.

Введение

Мы являемся свидетелями поворотного момента в мире баз данных. В течение многих лет реляционная модель была стандартом де-факто для решения любых задач – больших и малых. Мы не думаем, что в обозримом будущем реляционные СУБД полностью сойдут со сцены, но многие люди оглядываются по сторонам в поисках альтернатив, в частности: структур данных без схемы, нетрадиционных структур данных, простой репликации, высокой доступности, горизонтального масштабирования и новых способов формулирования запросов. Все эти технологии получили собирательное название *NoSQL*, и именно о них пойдет речь в этой книге.

Мы рассмотрим семь баз данных, представляющих широкий спектр технологий. Вы узнаете о функциональности разных СУБД и принятых при их проектировании компромиссах – долговечность данных или быстродействие, абсолютная согласованность или согласованность в конечном счете и т. д., а также поймете, как принимать оптимальное решение о выборе базы в каждом конкретном случае.

1.1. Все начинается с вопроса

Главный вопрос настоящей книги таков: какая база данных или их комбинация решает задачу оптимальным способом? Если после ее прочтения вы будете понимать, как сделать выбор, принимая во внимание конкретные требования и располагаемые ресурсы, то мы будем довольны.

Но чтобы ответить на этот вопрос, надо знать, какие имеются варианты. Для этого мы собираемся глубоко погрузиться в каждую из семи баз данных, показав ее сильные и слабые стороны. Вам предстоит попрактиковаться в операциях *CRUD*¹, освежить знания о схемах и найти ответы на следующие вопросы.

1 Create, Read, Update, Delete (создание, чтение, обновление, удаление). *Прим. перев.*

- *К какому типу относится это хранилище данных?* Существует много «жанров» баз данных: реляционные, ключи и значения, столбцовые, документо-ориентированные, графовые. Популярные СУБД, в том числе рассматриваемые в этой книге, обычно можно отнести к одной из этих категорий. Вы узнаете о том, для каких задач предназначена каждая категория. Мы специально выбрали базы, покрывающие все категории: одну реляционную (Postgres), два хранилища ключей и значений (Riak, Redis), столбцовую (HBase), две документо-ориентированных (MongoDB, CouchDB) и одну графовую (Neo4J).
- *Что было побудительным мотивом?* Базы данных создаются не в вакууме. Они предназначены для решения конкретных практических задач. РСУБД появились, когда гибкость запросов считалась важнее, чем гибкость схемы. С другой стороны, столбцовые базы данных прекрасно приспособлены для хранения больших объемов данных на нескольких машинах; связи между данными при этом отходят на второй план. Мы рассмотрим сценарии применения каждой базы данных и соответствующие примеры.
- *Как обращаться к базе?* Существуют различные способы подключения к базе данных. Если предлагается интерактивная командная утилита, то мы начинаем изучение с нее и только потом переходим к другим вариантам. Если требуется программирование, то мы стараемся ограничиться языками Ruby и JavaScript, хотя временами прибегаем и к другим, например PL/pgSQL (Postgres) или Gremlin (Neo4J). На более низком уровне обсуждаются протоколы типа REST (CouchDB, Riak) и Thrift (HBase). В последней главе рассматривается более сложная конфигурация баз данных, объединенных серверным скриптом для Node.js, написанным на JavaScript.
- *В чем состоит уникальность?* Любое хранилище данных поддерживает запись и обратное считывание данных. Все остальное изменяется в широких пределах. Иногда поддерживаются запросы по произвольным полям. Иногда – индексирование для ускорения поиска. Некоторые базы поддерживают произвольные запросы, для других запросы следует планировать заранее. Схема может быть жесткой и контролироваться СУБД, а может быть просто набором рекомендаций, которые переосматриваются по желанию. Понимание возможностей и огра-

ничений станет существенным подспорьем при выборе СУБД, которая в наибольшей степени отвечает решаемой задаче.

- *Как обстоит дело с производительностью?* Как функционирует база данных и во что это обходится? Поддерживается ли сегментирование? Как насчет репликации? Равномерно ли распределены данные за счет хороших функций хеширования или все данные оказываются в одном узле? Для чего оптимизирована база: для чтения, для записи или для какой-то другой операции? Есть ли какие-нибудь средства настройки и насколько они развиты?
- *Как масштабируется база данных?* Масштабируемость тесно связана с производительностью. Говорить о масштабируемости вне контекста – не указывая, что именно вы хотите *масштабировать*, в общем случае бессмысленно. В этой книге мы расскажем, как задавать правильные вопросы, устанавливающие такой контекст. Вопрос о том, *как* масштабируются разные базы, намеренно не акцентируется, но мы тем не менее поясним, к какому виду масштабирования наиболее приспособлена та или иная база: горизонтальному (MongoDB, HBase, Riak), традиционному вертикальному (Postgres, Neo4J, Redis) или чему-то среднему.

Наша цель заключается не в том, чтобы довести начинающего до уровня эксперта по всем рассматриваемым СУБД. Для детального рассмотрения любой из них потребовалась бы целая книга – и не одна (и такие книги есть). Однако к концу этой книги вы будете ясно представлять себе сильные стороны каждой базы данных и понимать, чем они отличаются друг от друга.

1.2. Жанры

Как и музыкальные произведения, базы данных можно отнести к одному или нескольким стилям. В отдельно взятой композиции используются те же самые ноты, что и в любой другой, но предназначены они для разных слушателей. Мало кто станет слушать мессу симинор Баха в открытом кабриолете, мчащемся по скоростной трассе 405 в Южной Калифорнии. Вот так и с базами данных – одни лучше приспособлены для решения одних задач, другие – других. Поэтому задавать следует не вопрос «Можно ли использовать эту СУБД для хранения и повышения качества данных?», а вопрос «Нужно ли это делать?».

В этом разделе мы изучим пять основных жанров баз данных, а также скажем, какие базы будем использовать для иллюстрации каждого жанра.

Важно помнить, что большая часть задач, с которыми вы сталкиваетесь на практике, может быть решена с помощью большинства, если не всех, рассматриваемых в этой книге, а также многих других СУБД. Вопрос не столько в том, можно ли приспособить СУБД конкретного стиля к моделированию данных, сколько в том, насколько эффективно будет ее применение в данной предметной области при данных сценариях использования и располагаемых ресурсах. Вы овладеете искусством предсказывать, насколько полезной окажется для вас конкретная база данных.

Реляционные СУБД

Реляционная модель первой приходит на ум большинству разработчиков с опытом в области баз данных. Реляционные системы управления базами данных (РСУБД) основаны на теории множеств, в основе их реализации лежат двумерные таблицы, состоящие из строк и столбцов. Канонический способ взаимодействия с РСУБД – написание запросов на языке Structured Query Language (SQL). Значения данных типизированы, это могут быть числа, строки, даты, неструктурированные двоичные объекты (BLOB) и т. п. Тип данных контролируется системой. Существенно, что благодаря математическим основаниям реляционной модели (теории множеств) исходные таблицы можно соединять и трансформировать в новые, более сложные.

Существует немало реляционных СУБД с открытым исходным кодом – MySQL, H2, HSQLDB, SQLite и многие другие, так что выбрать есть из чего. В этой книге мы остановимся на СУБД PostgreSQL (глава 2).

PostgreSQL

Испытанная в боях СУБД PostgreSQL – самая старая и надежная из всех рассматриваемых в этой книге. Совместимая со стандартом SQL, она покажется знакомой любому, кто раньше работал с реляционными базами данных, и послужит эталоном для сравнения с другими базами. Мы также рассмотрим некоторые малоизвестные средства SQL и уникальные достоинства Postgres. Здесь каждый – от новичка до эксперта – найдет себе что-то по душе.

Хранилища ключей и значений

Хранилище ключей и значений (КЗ-хранилище) – простейшая из всех рассматриваемых моделей. Как следует из названия, КЗ-хранилище сопоставляет значения ключам, как словарь (или хеш-таблица) в любом популярном языке программирования. Некоторые КЗ-хранилища допускают в качестве значений составные типы данных, например хеши или списки, но это необязательно. Есть реализации, в которых ключи можно перебирать, но это также считается дополнительным бонусом. Примером КЗ-хранилища можно считать файловую систему, если рассматривать путь к файлу как ключ, а его содержимое – как значение. Поскольку от КЗ-хранилища требуется так мало, то базы данных этого типа могут демонстрировать невероятно высокую производительность, но в общем случае бесполезны, когда требуются сложные запросы и агрегирование.

Как и в случае реляционных СУБД, имеется много продуктов с открытым исходным кодом. Из наиболее популярных отметим memcached (и родственные ему memcachedb и membase), Voldemort и две системы, рассматриваемые в этой книге: Redis и Riak.

Riak

Riak (глава 3) – это не просто хранилище ключей и значений, система изначально поддерживает такие веб-технологии, как HTTP и REST. Это точный повтор системы Dynamo, используемой компанией Amazon, с некоторыми дополнительными функциями, например, векторные часы для разрешения конфликтов. Значением в Riak может быть всё что угодно: простой текст, XML-документ, изображение и т. д., а связи между ключами описываются именованными структурами, которые называются *ссылками* (links). Riak – не самая известная из рассмотренных в этой книге баз данных, но постепенно она набирает популярность, и на ее примере мы впервые рассмотрим исполнение запросов с помощью технологии распределения-редукции (mapreduce).

Redis

Система Redis поддерживает составные типы данных, в частности отсортированные множества и хеши, а также базовые коммуникационные средства, в том числе публикацию-подписку и блокирующие очереди. Она также располагает одним из самых развитых механизмов запросов среди всех КЗ-хранилищ. А за счет кэширования операций

записи в памяти Redis достигает впечатляющей производительности ценой повышенного риска потери данных в случае аппаратного сбоя. Благодаря этой характеристике она может служить неплохим средством кэширования некритических данных, а также брокером сообщений. Мы рассмотрим эту систему в конце книги (глава 8), где построим приложение, в котором гармонично сочетаются Redis и другие базы данных.

Столбцовые базы данных

Столбцовые, или ориентированные на хранение данных по столбцам базы данных получили свое название благодаря одному существенно-му аспекту дизайна: данные, принадлежащие одному столбцу (в смысле двумерных таблиц) хранятся рядом. Напротив, в строковых базах данных (к числу которых относятся реляционные), рядом хранятся данные, принадлежащие одной строке. Различие может показаться второстепенным, однако последствия такого проектного решения весьма глубоки. В столбцовых базах данных добавление нового столбца обходится дешево и производится построчно. В каждой строке набор столбцов может быть разным, возможно даже, что в некоторых строках столбцов вообще нет, и, значит, таблица может быть *разреженной* без накладных расходов на хранение null-значений. С точки зрения структуры, столбцовые базы данных занимают промежуточное положение между реляционными СУБД и КЗ-хранилищами.

На рынке столбцовых баз данных конкуренция меньше, чем среди реляционных СУБД и хранилищ ключей и значений. Наиболее популярны три системы: HBase (мы рассмотрим ее в главе 4), Cassandra и Hypertable.

HBase

Из всех рассмотренных нами нереляционных СУБД эта столбцовая база данных ближе всего к реляционной модели. HBase спроектирована по образцу системы Google BigTable, построена на базе Hadoop (механизм распределения-редукции) и предназначена для горизонтального масштабирования на кластерах, составленных из стандартного оборудования. HBase дает строгие гарантии непротиворечивости данных и предоставляет таблицы, состоящие из строк и столбцов, – поклонникам SQL это придется по нраву. Готовая поддержка версионирования и сжатия ставит эту СУБД на первое место в категории «большие данные».

Документо-ориентированные базы данных

В документо-ориентированных, или просто документных базах данных хранятся – естественно – документы. В двух словах документ – это некий аналог хеша, в котором имеется поле уникального идентификатора, а в качестве значения могут выступать данные произвольного типа, в том числе другие хеши. Документы могут содержать вложенные структуры и обладают высокой гибкостью, что делает их пригодными для применения в разных предметных областях. Система налагает немного ограничений на входные данные при условии, что они удовлетворяют базовым требованиям к представимости в виде документа. В различных документных базах данных применяются различные подходы к индексированию, формулированию произвольных запросов, репликации, обеспечению согласованности и другим аспектам. Для правильного выбора системы нужно хорошо понимать эти различия и их влияние на конкретный сценарий использования.

Два основных игрока на поле документных баз данных с открытым исходным кодом – MongoDB (рассматривается в главе 5) и CouchDB (глава 6).

MongoDB

СУБД MongoDB проектировалась для хранения *гигантских* объемов данных (*mongo* – часть слова *humongous* – «монструозный»²). При настройке сервера Mongo предпочтение отдается согласованности – после операции записи все последующие операции чтения извлекают одно и то же значение (до следующего обновления). Эта особенность делает MongoDB привлекательной альтернативой для тех, кто имеет опыт работы с РСУБД. Кроме того, MongoDB поддерживает атомарные операции чтения-записи, в том числе инкрементирование значения и запросы к вложенным документам. Благодаря использованию JavaScript в качестве языка запросов MongoDB поддерживает как простые запросы, так и сложные задания с распределением-редукцией.

CouchDB

Система CouchDB рассчитана на разнообразные сценарии развертывания – от центра обработки данных до настольного ПК и даже

2 Объединение слов *huge* и *monstrous*. Прим. перев.

смартфона. Написанная на языке Erlang, CouchDB может похвастаться уровнем живучести, нечасто встречающимся среди баз данных. Ее файлы данных почти невозможно повредить, при этом CouchDB сохраняет высокую доступность даже в условиях спорадической потери связи или аппаратных сбоев. Как и в Mongo, языком запросов в CouchDB является JavaScript. Представления описываются функциями `mapreduce`, которые хранятся в виде документов и реплицируются между узлами как обычные данные.

Графовые базы данных

Из реже используемых стилей следует отметить графовые базы данных, достоинства которых ярко проявляются при обработке данных с большим количеством связей. Графовая база состоит из узлов и связей между ними. Как с узлами, так и со связями можно ассоциировать свойства – пары ключ-значение, – в которых хранятся данные. Истинная сила графовых баз данных заключается в возможности обхода узлов, следуя связям.

В главе 7 мы рассмотрим наиболее популярную на сегодня графовую базу данных – Neo4J.

Neo4J

Операция, на которой другие базы данных часто сдают, – это обход данных со ссылками на себя или с другими сложно устроенными связями. Именно здесь достоинства Neo4J проявляются во всем блеске. Преимущество графовой базы данных в том и состоит, что обеспечивается быстрый просмотр узлов и связей для поиска нужных данных. Такие базы часто используются в социальных сетях и завоевали признание за свою гибкость, кульминацией которой служит Neo4j.

Многостороннее хранение

На практике различные базы данных часто используются в сочетании. Все еще нетрудно встретить приложение, где применяется только реляционная СУБД, но со временем все популярнее становятся комбинации разных баз данных, в которых сильные стороны каждой позволяют создать экосистему, которая оказывается более мощной, функциональной и надежной, чем сумма ее частей. Эту практику, получившую название *многостороннее хранение* (*polyglot persistence*) мы рассмотрим в главе 9.

1.3. Вперед и вверх

Мы сейчас находимся в середине кембрийского взрыва разнообразия способов хранения данных; трудно предсказать, куда пойдет эволюция. Но есть достаточные основания полагать, что доминирование какой-то одной стратегии (реляционной или нет) маловероятно. Скорее, мы станем свидетелями появления различных высоко специализированных баз данных, каждая из которых приспособлена к решению задачи из некоторой идеализированной предметной области (хотя, конечно, будут и пересечения). И если сейчас имеются рабочие места, требующие квалификации в администрировании реляционных баз данных, то в будущем все больше будет спрос на специалистов по нереляционным системам.

Базы данных, как языки программирования и библиотеки, являются инструментарием, которым должен владеть любой разработчик. Всякий хороший плотник должен знать, что находится в его поясе для инструментов. И ни один строитель не может надеяться стать прорабом, не владея информацией об имеющихся технологиях.

Считайте эту книгу экспресс-курсом. Прочитав ее, вы научитесь колотить молотком, сверлить отверстия дрелью, работать с гвоздезабивным пистолетом и в конце концов сумеете построить нечто куда более серьезное, чем скворечник. Ну а теперь без дальнейших предисловий приступим к изучению нашей первой базы данных: PostgreSQL.



ГЛАВА 2.

PostgreSQL

PostgreSQL – молоток в мире баз данных. Ее хорошо знают, она легко доступна, стабильна и при должном старании и умении способна решать на удивление разнообразные задачи. Нельзя рассчитывать стать опытным строителем, не овладев этим самым распространенным инструментом.

PostgreSQL – реляционная система управления базами данных, то есть основана на теории множеств, реализована в виде двумерных таблиц, где данные хранятся по строкам, и строго контролирует типы столбцов. Несмотря на растущий интерес к новым тенденциям в области баз данных, реляционный стиль является самым популярным и, вероятно, останется таким еще довольно долго.

Преобладание реляционных СУБД объясняется не только встроенным в них обширным инструментарием (триггеры, хранимые процедуры, развитые индексы), безопасностью данных (благодаря свойствам транзакционности ACID¹), количеством специалистов (многие программисты говорят и думают в реляционных терминах), но и гибкостью формулирования запросов. В отличие от некоторых других хранилищ данных, не требуется заранее планировать, как будут использоваться данные. Если реляционная схема нормализована, то можно предъявлять практически произвольные запросы. PostgreSQL – прекрасный пример системы с открытым исходным кодом, следующей традициям РСУБД.

2.1. Произносится Post-greS-Q-L

PostgreSQL – самая старая и проверенная временем СУБД из всех рассматриваемых в этой книге. К ней прилагаются подключаемые модули для разбора запросов на естественном языке, для построения многомерных индексов, для запросов к географическим данным,

¹ Атомарность, непротиворечивость, изолированность, долговечность. *Прим. перев.*

для создания собственных типов данных и для многого другого. В ней реализованы хитроумные механизмы обработки транзакций, она позволяет писать хранимые процедуры на десятке языков и работает на самых разных платформах. В PostgreSQL встроена поддержка Unicode, последовательностей, наследования таблиц, подзапросов, а реализация SQL точнее следует стандарту ANSI, чем любая другая из представленных на рынке. СУБД является быстрой и надежной, способна хранить терабайты данных и доказала работоспособность в таких высоконагруженных проектах, как Skure, французская Национальная касса по выплате пособий многодетным семьям (Caisse Nationale d'Allocations Familiales – CNAF) и Федеральное управление гражданской авиации США.

Что в имени тебе моем?

Проект PostgreSQL существует в современном воплощении с 1995 года, но его корни гораздо глубже. Первоначально проект был написан в университете Беркли в начале 1970-х годов и назывался Interactive Graphics Retrieval System (диалоговая система графического поиска), сокращенно «Ingres». В 1980-х годах была выпущена улучшенная версия post-Ingres, сокращенно Postgres. В 1993 году Беркли прекратил работу над проектом, но ее продолжило сообщество, выпустив систему с открытым исходным кодом Postgres95. В 1996 году проект был переименован в PostgreSQL, чтобы подчеркнуть поддержку новых возможностей SQL и с тех пор так и называется.

Установить PostgreSQL можно разными способами – в зависимости от операционной системы². Помимо базовой системы, нам понадобятся следующие дополнительные пакеты: `tablefunc`, `dict_xsyn`, `fuzzystrmatch`, `pg_trgm` и `cube`. Инструкции по установке приведены на сайте³.

Установив Postgres, создайте схему `book`, выполнив следующую команду:

```
$ createdb book
```

Схема `book` будет использоваться до конца этой главы. Затем выполните команду, которая проверит корректность установки дополнительных пакетов:

```
$ psql book -c "SELECT '1'::cube;"
```

Если будет выдано сообщение об ошибке, обратитесь к онлайн-вой документации.

² <http://www.postgresql.org/download/>

³ <http://www.postgresql.org/docs/9.0/static/contrib.html>

2.2. День 1: отношения, операции CRUD и соединения

Не рассчитывая встретить в вас эксперта по реляционным базам данных, мы все же предполагаем, что с одной-другой базой вам доводилось сталкиваться. Более чем вероятно, что эта база была реляционной. Мы начнем с создания и заполнения схемы. Затем мы познакомимся с запросами и, наконец, обсудим аспект, который выделяет реляционные базы данных среди прочих: соединение таблиц.

Как и большинство баз данных, о которых мы будем говорить, Postgres включает сервер, выполняющий всю работу, и командную оболочку, позволяющую подключиться к серверу. По умолчанию сервер прослушивает порт 5432, к которому можно подключиться из оболочки `psql`.

```
$ psql book
```

PostgreSQL выводит приглашение, состоящее из имени базы данных, за которой следует знак решетки, если вы работаете от имени администратора, или знак доллара – если от имени обычного пользователя. В оболочку встроена самая лучшая документация, которую можно получить на консоли. Набрав `\h`, вы получите информацию о командах SQL, а набрав `\?` – информацию о специфичных для `psql` командах, начинающихся со знака обратной косой черты. Подробные сведения о конкретной команде SQL можно получить следующим образом:

```
book=# \h CREATE INDEX
Command: CREATE INDEX
Description: define a new index
Syntax:
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table [ USING method ]
( { column | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | ...
[ WITH ( storage_parameter = value [, ... ] ) ]
[ TABLESPACE tablespace ]
[ WHERE predicate ]
```

Перед тем как погружаться в недра Postgres, было бы полезно поближе познакомиться с этим инструментом. Имеет смысл изучить (или освежить в памяти) несколько наиболее употребительных команд, например `SELECT` или `CREATE TABLE`.

Введение в SQL

В PostgreSQL используется принятое в SQL соглашение о том, что отношения называются таблицами (TABLE), атрибуты – столбцами (COLUMN), а кортежи – строками (ROW). Мы будем последовательно придерживаться этой терминологии, хотя в специализированной литературе можно встретить и строгие математические термины: *отношения, атрибуты, кортежи*. Дополнительные сведения об этих понятиях см. на врезке «Математические отношения».

Работа с таблицами

Принадлежа к реляционным СУБД, PostgreSQL нуждается в предварительном проектировании схемы. Сначала создается схема, а затем вводятся данные, совместимые с определением схемы.

Для создания схемы необходимо задать ее имя и список столбцов с указанием типов и необязательных ограничений. В каждой таблице желательно также завести столбец, содержащий уникальный идентификатор, который позволяет отличить данную строку от всех прочих. Этот идентификатор называется первичным ключом (PRIMARY KEY). Команда SQL для создания таблицы `countries` выглядит следующим образом:

```
CREATE TABLE countries (  
  country_code char(2) PRIMARY KEY,  
  country_name text UNIQUE  
);
```

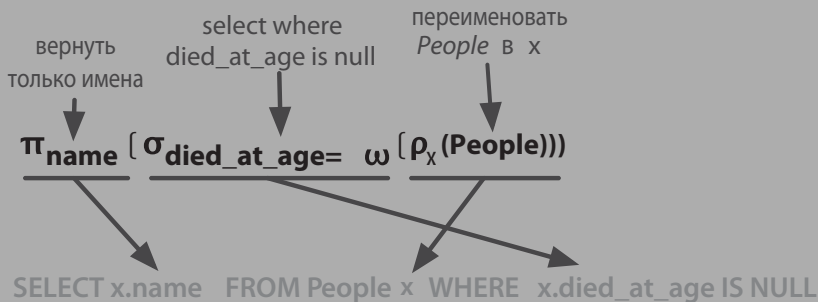
Математические отношения

Реляционные базы данных получили свое название от слова *relation* (отношения, иначе говоря таблицы). Отношение состоит из множества кортежей (*tuple*), или строк, которые сопоставляют *атрибутам* (*attribute*) атомарные значения (например, {name: 'Genghis Khan', p.died_at_age: 65}). Состав допустимых атрибутов определяется *заглавным* кортежем, которые отображается на некоторую область определения (*domain*), или ограничивающий тип (то есть на набор столбцов, например {name: string, age: int}). Это и есть существо реляционной структуры в двух словах.

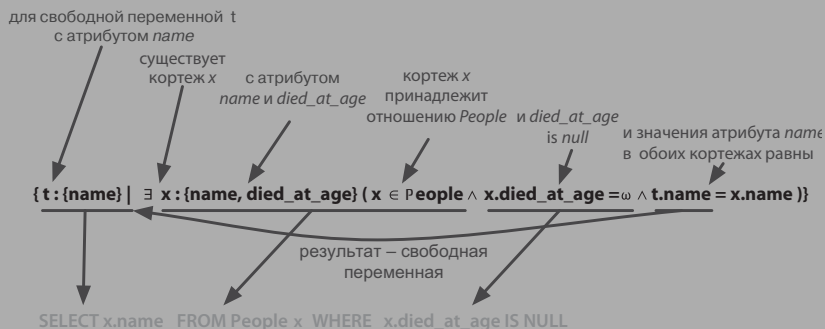
В реализациях применяется гораздо более прагматичная фразеология. Так зачем вообще упоминать эти термины? Затем, чтобы показать, что реляционные базы данных получили свое название благодаря слову *relation* в его математическом смысле. А вовсе не потому, что таблицы «соотносятся» друг с другом посредством внешних ключей, – существуют такие ограничения в действительности или нет, к делу совершенно не относится.

Хотя математическая подоплека в значительной мере скрыта от пользователя, своей мощью модель, безусловно, обязана математике. Именно она

позволяет формулировать сложные запросы, а затем поручать системе оптимизировать их выполнение, применяя определенные алгоритмы. РСУБД строятся на базе раздела теории множеств, называемого *реляционной алгеброй*, в которой определены операции выборки (*WHERE ...*), проецирования (*SELECT ...*), декартова произведения (*JOIN ...*) и другие. Определения устроены, как показано на рисунке ниже.



Интерпретация отношения как физической таблицы (массива массивов, как неизменно повторяют на начальных занятиях по базам данных) может привести к недоразумениям на практике, например попытке написать код обхода всех строк. Реляционные запросы формулируются гораздо более декларативно – на основе раздела математики, который называется *реляционное исчисление*. Можно показать, что реляционное исчисление эквивалентно реляционной алгебре. PostgreSQL и другие РСУБД оптимизируют запросы, выполняя сведение одного к другому и применяя упрощающие преобразования. Легко видеть, что представление команды SQL на следующем рисунке эквивалентно приведенному выше.



Новая таблица будет содержать набор строк, каждая из которых идентифицируется двузначным кодом, причем все имена уникальны. На оба столбца наложены *ограничения*. Ограничение *PRIMARY KEY* на столбец *country_code* запрещает появление одинаковых кодов

стран. В таблице может существовать только одна строка с кодом `us` и только одна строка с кодом `gb`. Аналогичное ограничение уникальности налагается на столбец `country_name`, хотя он и не является первичным ключом. Вставим в таблицу `countries` несколько строк.

```
INSERT INTO countries (country_code, country_name)
VALUES ('us','United States'), ('mx','Mexico'), ('au','Australia'),
      ('gb','United Kingdom'), ('de','Germany'), ('ll','Loompaland');
```

Проверим, как работает ограничение уникальности. Попытка добавить строку с повторяющимся значением в столбце `country_name` нарушает ограничение уникальности, поэтому вставка не выполняется. Именно с помощью ограничений реляционные базы данных – и PostgreSQL в том числе – гарантируют корректность данных.

```
INSERT INTO countries
VALUES ('uk','United Kingdom');
```

```
ERROR: duplicate key value violates unique constraint "countries_country_name_key"
DETAIL: Key (country_name)=(United Kingdom) already exists.
```

Чтобы проверить, те ли строки были вставлены, мы можем прочитать их с помощью команды `SELECT...FROM table`.

```
SELECT *
FROM countries;
```

```
country_code | country_name
-----+-----
us           | United States
mx           | Mexico
au           | Australia
gb           | United Kingdom
de           | Germany
ll           | Loompaland
(6 rows)
```

Ни на какой карте вы не найдете страны `Loompaland`, поэтому давайте удалим ее из таблицы. Чтобы указать, какую строку удалять, мы используем фразу `WHERE`. Следующая команда удалит строку, в которой столбец `country_code` содержит значение `ll`.

```
DELETE FROM countries
WHERE country_code = 'll';
```

Теперь, когда в таблице `countries` остались только реально существующие страны, создадим таблицу `cities`. Чтобы быть уверенными, что код страны `country_code` в любой вставляемой строке при-

существует в таблице `countries`, добавим ключевое слово `REFERENCES`. Поскольку столбец `country_code` ссылается на ключ в другой таблице, такое ограничение называется *ограничением внешнего ключа*.

```
CREATE TABLE cities (  
    name text NOT NULL,  
    postal_code varchar(9) CHECK (postal_code <> ''),  
    country_code char(2) REFERENCES countries,  
    PRIMARY KEY (country_code, postal_code)  
);
```

Об операциях CRUD

CRUD – это мнемоническая аббревиатура для запоминания основных операций работы с данными: *Create, Read, Update, Delete* (создание, чтение, обновление, удаление). Все операции, кроме вставки новых записей (*create*), модификации существующих (*update*) и удаления ненужных (*delete*), – сколько бы хитроумными ни были соответствующие запросы – называются *операциями чтения* (*read*). Имея в своем распоряжении операции CRUD, вы можете делать с данными всё что угодно.

На этот раз мы наложили ограничение на столбец `name` в таблице `cities`, запретив значения `NULL`. На столбец `postal_code` наложено ограничение, проверяющее, что строка не пуста (<> означает *не равно*). Далее, поскольку первичный ключ (`PRIMARY KEY`) уникально идентифицирует строку, мы создали составной ключ: `country_code` + `postal_code`. В сочетании эти два поля (код страны и почтовый индекс) определяют строку однозначно.

Postgres располагает широким набором типов данных. Выше вы видели три разных представления строк: `text` (строка произвольной длины), `varchar(9)` (строка переменной длины, не превышающей 9 символов) и `char(2)` (строка, состоящая ровно из двух символов). Подготовив схему, попробуем вставить строку, описывающую город Торонто в Канаде:

```
INSERT INTO cities  
VALUES ('Toronto', 'M4C1B5', 'ca');
```

```
ERROR: insert or update on table "cities" violates foreign key constraint  
"cities_country_code_fkey"  
DETAIL: Key (country_code)=(ca) is not present in table "countries".
```

Не получилось – и это правильно! Поскольку столбец `country_code` ссылается (`REFERENCES`) на таблицу `countries`, то в этой таблице должна существовать строка с указанным значением `country_code`. Этот механизм, называемый *поддержанием ссылочной целостности*,

проиллюстрирован на рис. 1. Он гарантирует правильность данных. Стоит отметить, что NULL в столбце `cities.country_code` допускается, так как ключевое слово NULL обозначает отсутствие какого-либо значения. Чтобы запретить NULL-ссылки в столбце `country_code`, нужно было бы определить его следующим образом:

```
country_code char(2) REFERENCES countries NOT NULL.
```

Теперь попробуем вставить город в США:

```
INSERT INTO cities
VALUES ('Portland', '87200', 'us');
```

```
INSERT 0 1
```

Эта операция завершилась успешно. Но по ошибке мы ввели неправильное значение в поле `postal_code`. Почтовый индекс Портленда на самом деле равен `97205`. Вместо того чтобы удалять и снова вставлять строку, мы можем обновить ее на месте:

```
UPDATE cities
SET postal_code = '97205'
WHERE name = 'Portland';
```

Итак, мы продемонстрировали все операции CRUD – создание, чтение, обновление и удаление строк таблицы – в действии.

name	postal_code	country_code	country_code	country_name
Portland	97205	us	us	United States
		mx		Mexico
		au		Australia
		uk		United Kingdom
		de		Germany

Рис. 1. Ключевое слово REFERENCES говорит, что значение поля в одной таблице должно совпадать с первичным ключом какой-то строки другой таблицы

Чтение с соединением

Все базы данных, о которых мы расскажем в этой книге, поддерживают операции CRUD. Но что выделяет реляционные СУБД типа PostgreSQL из общего ряда – так это возможность соединять таблицы при чтении. Соединением называется операция, которая принимает на входе две таблицы и, комбинируя их определенным об-

разом, возвращает новую таблицу. Что-то вроде составления новых слов из существующих в игре «скрэбл».

Наиболее распространенной является операция *внутреннего соединения*. В простейшем случае с помощью ключевого слова `ON` задаются два столбца (по одному из каждой таблицы), значения которых должны совпадать:

```
SELECT cities.*, country_name
FROM cities INNER JOIN countries
    ON cities.country_code = countries.country_code;
```

country_code	name	postal_code	country_name
us	Portland	97205	United States

Соединение возвращает одну таблицу, в которой присутствуют значения всех столбцов из таблицы `cities` плюс значения столбцов из той строки таблицы `countries`, в которой значение `country_name` такое же, как в соответствующей строке таблицы `cities`.

Можно также выполнить соединение с таблицей типа `cities`, в которой первичный ключ составной. Для проверки составного соединения создадим еще одну таблицу, в которой будет храниться список мест проведения мероприятий.

Место проведения мероприятия существует в конкретной *стране* на территории с конкретным *почтовым индексом*. Поэтому в состав *внешнего ключа* должны входить два столбца, ссылающиеся на оба столбца, образующих *первичный ключ* таблицы `cities`. Ограничение `MATCH FULL` гарантирует, что оба значения существуют или оба содержат `NULL`.

```
CREATE TABLE venues (
    venue_id SERIAL PRIMARY KEY,
    name varchar(255),
    street_address text,
    type char(7) CHECK ( type in ('public','private') ) DEFAULT 'public',
    postal_code varchar(9),
    country_code char(2),
    FOREIGN KEY (country_code, postal_code)
        REFERENCES cities (country_code, postal_code) MATCH FULL
);
```

Столбец `venue_id` – пример часто встречающегося способа определения первичного ключа: автоматически увеличивающиеся целые числа (1, 2, 3, 4 ...). Для задания такого идентификатора употребляется ключевое слово `SERIAL` (в MySQL аналогичная конструкция обозначается ключевым словом `AUTO_INCREMENT`).

```
INSERT INTO venues (name, postal_code, country_code)
VALUES ('Crystal Ballroom', '97205', 'us');
```

Мы не задавали значение `venue_id` явно, но при создании строки оно было сгенерировано. Вернемся, однако, к составному соединению. Для соединения таблицы `venues` с таблицей `cities` необходимо задать *оба* столбца, образующих составной ключ. Чтобы меньше набирать, мы можем назначить именам таблиц псевдонимы, указав их после настоящих имен с необязательным ключевым словом `AS` (например, `venues v` или `venues AS v`).

```
SELECT v.venue_id, v.name, c.name
FROM venues v INNER JOIN cities c
    ON v.postal_code=c.postal_code AND v.country_code=c.country_code;
```

venue_id	name	name
1	Crystal Ballroom	Portland

Дополнительно можно потребовать, чтобы PostgreSQL вернула значение столбца после вставки, для этого нужно добавить в конец запроса фразу `RETURNING`.

```
INSERT INTO venues (name, postal_code, country_code)
VALUES ('Voodoo Donuts', '97205', 'us') RETURNING venue_id;
```

id
2

Таким образом, мы получаем новое значение `venue_id`, не выполняя дополнительный запрос.

Внешние соединения

Помимо внутренних соединений, PostgreSQL умеет выполнять *внешние соединения*. При таком способе должны быть возвращены все строки одной таблицы, даже если им нет соответствия в другой.

Проиллюстрировать это проще всего на примере, но для этого нам придется создать новую таблицу `events`. Оставляем это вам в качестве упражнения. Таблица `events` должна содержать следующие столбцы: целочисленный последовательный (`SERIAL`) идентификатор мероприятия `event_id`, название `title`, время начала `starts` и время окончания `ends` (типа *timestamp*) и идентификатор места проведения `venue_id` (внешний ключ, ссылающийся на таблицу `venues`). На диаграмме схемы (рис. 2) изображены все созданные к этому моменту таблицы.

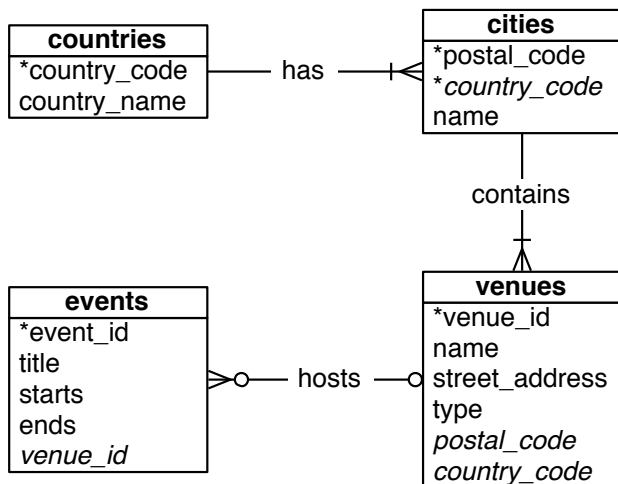


Рис. 2. Диаграмма сущность-связь

Создав таблицу `events`, вставим в нее строки (временные метки записываются в виде строк вида `2012-02-15 17:30`), описывающие два праздника и собрание клуба.

Title	starts	ends	venue_id	event_id
LARP Club	2012-02-15 17:30:00	2012-02-15 19:30:00	2	1
April Fools Day	2012-04-01 00:00:00	2012-04-01 23:59:00		2
Christmas Day	2012-12-25 00:00:00	2012-12-25 23:59:00		3

Сначала напишем запрос, который возвращает название мероприятия и места его проведения с помощью внутреннего соединения (слово `INNER` в `INNER JOIN` необязательно, поэтому опустим его).

```
SELECT e.title, v.name
FROM events e JOIN venues v
  ON e.venue_id = v.venue_id;
```

title	name
LARP Club	Voodoo Donuts

`INNER JOIN` возвращает только те строки, для которых значения столбцов совпадают. Поскольку в таблице `venues` не существует строк, в которых столбец `venue_id` содержит `NULL`, то две строки, в которых `events.venue_id` равно `NULL`, ни на что не ссылаются. Чтобы получить все мероприятия, даже те, для которых место прове-

дения неизвестно, нужно воспользоваться оператором `LEFT OUTER JOIN` (его можно сократить до `LEFT JOIN`).

```
SELECT e.title, v.name
FROM events e LEFT JOIN venues v
    ON e.venue_id = v.venue_id;
```

Title	name
LARP Club	Voodoo Donuts
April Fools Day	
Christmas Day	

Чтобы наоборот получить все места проведения и только соответствующие им мероприятия, следует воспользоваться оператором `RIGHT JOIN`. Наконец, существует также оператор `FULL JOIN` — объединение `LEFT` и `RIGHT`; он возвращает все строки из обеих таблиц вне зависимости от того, есть соответствие между столбцами или нет.

Быстрый поиск с применением индексов

Быстродействие PostgreSQL (или любой другой ПСУБД) обеспечивается эффективным механизмом управления блоками данных, который позволяет сократить количество операций чтения диска, оптимизацией запросов и другими приемами. Но их возможности по ускорению извлечения данных все же ограничены. Чтобы выбрать мероприятие *Christmas Day* из таблицы `events`, алгоритм должен просмотреть все строки и найти те, что удовлетворяют условиям запроса. Без *индекса* пришлось бы прочитать с диска каждую строку, иначе было бы невозможно понять, возвращать ее или нет. Это иллюстрируется на следующем примере.

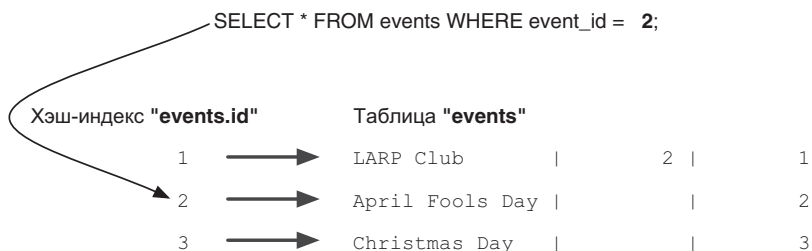
соответствует "Christmas Day"? Нет.	→ LARP Club		2		1
соответствует "Christmas Day"? Нет.	→ April Fools Day				2
соответствует "Christmas Day"? Да.	→ Christmas Day				3

Индекс — это специальная структура данных, которая позволяет избежать полного сканирования таблицы при выполнении запроса. Возможно, вы обратили внимание, что при выполнении команды `CREATE TABLE` печатается сообщение вида:

```
CREATE TABLE / PRIMARY KEY will create implicit index "events_pkey" \
for table "events"
```

PostgreSQL автоматически создает индекс по первичному ключу; его ключом является значение первичного ключа, а значением — указатель на строку на диске, как показано на рисунке ниже.

Другой способ принудительно создать индекс по столбцу таблицы – воспользоваться ключевым словом `UNIQUE`.



Можно и явно добавить хеш-индекс с помощью команды `CREATE INDEX`; отметим, что в хеш-индексе все значения должны быть уникальны (как в хеш-таблице или словаре).

```
CREATE INDEX events_title
ON events USING hash (title);
```

Если требуется сравнивать столбцы с помощью операторов меньше/больше/равно, то хеш-индекса недостаточно, необходима более гибкая структура – В-дерево (см. рис. 3). Рассмотрим запрос для поиска всех мероприятий с датой проведения *не ранее 1 апреля*.

```
SELECT *
FROM events
WHERE starts >= '2012-04-01';
```

Для такого запроса идеальной структурой данных является дерево. Чтобы построить В-дерево по столбцу `starts`, выполним следующую команду:

```
CREATE INDEX events_starts
ON events USING btree (starts);
```

Теперь при выполнении запроса по диапазону дат можно будет обойтись без полного сканирования таблицы. Если требуется просмотреть миллионы или миллиарды строк, то разница будет очень заметна.

Мы можем посмотреть на результат своей работы, выведя список всех определенных в схеме индексов:

```
book=# \di
```

Следует отметить, что при использовании ограничения `FOREIGN KEY` PostgreSQL автоматически создает индекс по указанному в

нем столбцу (или столбцам). Даже если вам не нравятся ограничения базы данных (да-да, это в ваш адрес, разработчики на платформе Ruby on Rails), создавать индексы по столбцам, участвующим в операциях соединения, полезно, так как это увеличивает скорость их выполнения.

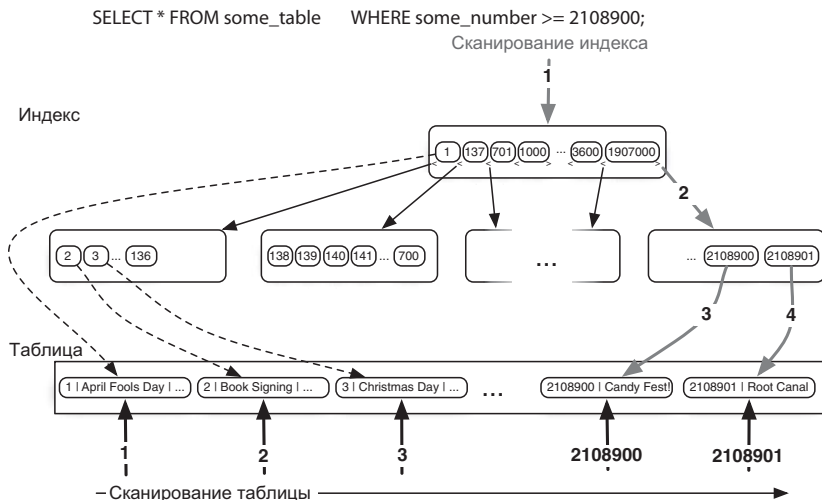


Рис. 3. Индекс типа В-дерево может использоваться для выполнения запросов по диапазону

День 1: итоги

Сегодня мы прошли немалый путь и ввели целый ряд терминов. Напомним их.

Термин	Определение
Столбец	Множество значений определенного типа; иногда называется также <i>атрибутом</i>
Строка	Объект, состоящий из набора значений столбцов; иногда называется также <i>кортежем</i>
Таблица	Множество строк с одинаковыми столбцами; иногда называется также <i>отношением</i>
Первичный ключ	Уникальное значение, определяющее конкретную строку
CRUD	Create, Read, Update, Delete

Термин	Определение
SQL	Структурированный язык запросов (Structured Query Language), лингва франка реляционных баз данных
Соединение	Комбинирование двух таблиц с получением одной путем сопоставления столбцов
Левое соединение	Способ комбинирования двух таблиц с получением одной, при котором строки из левой таблицы включаются, даже если им нет соответствия в правой; при этом вместо значений тех столбцов правой таблицы, которым не найдено соответствие, возвращается <code>NULL</code>
Индекс	Структура данных, предназначенная для оптимизации выборки по значениям определенного набора столбцов
В-дерево	Хороший стандартный индекс; значения хранятся в сбалансированном дереве; обладает высокой гибкостью

Вот уже сорок лет как реляционные базы данных де факто являются стратегией управления данными – многие из нас начинали свою карьеру в середине этого периода. Мы познакомились с некоторыми ключевыми понятиями реляционной модели на примере простых SQL-запросов. Завтра мы обогатим свои знания.

День 1: домашнее задание

Информационный поиск

1. Поставьте в браузере закладки на FAQ и документы по PostgreSQL.
2. Разберитесь с тем, что выводится на консоль при задании в командной строке флагов `\?` и `\h`.
3. Найдите в документации, что означает часть `MATCH FULL` в определении внешнего ключа `FOREIGN KEY`.

Задачи

1. Выберите из таблицы `pg_class` все таблицы, которые мы создали (и только их).
2. Напишите запрос для поиска названия страны, соответствующей мероприятию `LARP Club`.
3. Измените таблицу `venues`, добавив в нее булевский столбец `active` со значением по умолчанию `TRUE`.

2.3. День 2: более сложные запросы, код и правила

Вчера мы видели, как определить схему, наполнить ее данными, обновлять и удалять строки и выполнять простые операции чтения. Сегодня мы более глубоко рассмотрим бесконечное разнообразие способов запроса данных в PostgreSQL. Мы покажем, как группировать похожие значения, как выполнять код на сервере и как создавать специализированные интерфейсы с помощью *представлений* и *правил*. И закончим день, продемонстрировав использование одного из дополнительных пакетов PostgreSQL, который позволяет поставить таблицы с ног на голову.

Агрегатные функции

Агрегатный запрос группирует результаты, извлеченные из нескольких строк, по некоторому общему критерию. Это может быть, например, простой подсчет числа строк в таблице или вычисление среднего значения некоторого числового столбца. Это мощные средства SQL, использование которых способно доставить немало удовольствия.

Мы поэкспериментируем с некоторыми агрегатными функциями, но сначала добавим в нашу базу еще немного данных. Вставьте свою страну в таблицу `countries`, свой город в таблицу `cities` и свой адрес в качестве адреса проведения мероприятия (мы просто ввели строку *My Place*). Затем добавьте несколько записей в таблицу `events`.

Небольшой совет: вместо того чтобы задавать значение `venue_id` явно, можно вернуть его в качестве результата подзапроса – так будет понятнее человеку. Например, если *Moby* дает концерт в *Crystal Ballroom*, то `venue_id` можно задать так:

```
INSERT INTO events (title, starts, ends, venue_id)
VALUES ('Moby', '2012-02-06 21:00', '2012-02-06 23:00', (
    SELECT venue_id
    FROM venues
    WHERE name = 'Crystal Ballroom'
));
```

Вставьте в таблицу `events` следующие данные (в PostgreSQL для ввода строки *Valentine's Day* один апостроф заменяется двумя, например, *Heaven's Gate*):

title	starts	ends	venue
-----+-----	-----+-----	-----+-----	-----+-----
Wedding	2012-02-26 21:00:00	2012-02-26 23:00:00	Voodoo Donuts
Dinner with Mom	2012-02-26 18:00:00	2012-02-26 20:30:00	My Place
Valentine's Day	2012-02-14 00:00:00	2012-02-14 23:59:00	

Подготовив данные, выполним несколько агрегатных запросов. Самая простая агрегатная функция — `count()` — вряд ли нуждается в пояснениях. Запросив, сколько названий содержат слово *Day* (примечание: знак `%` употребляется как метасимвол в запросах с оператором `LIKE`), мы получим 3.

```
SELECT count(title)
FROM events
WHERE title LIKE '%Day%';
```

Чтобы получить самое раннее время начала и самое позднее время окончания мероприятий в зале Crystal Ballroom, нужно воспользоваться функциями `min()` (возвращает наименьшее значение) и `max()` (возвращает наибольшее значение).

```
SELECT min(starts), max(ends)
FROM events INNER JOIN venues
ON events.venue_id = venues.venue_id
WHERE venues.name = 'Crystal Ballroom';
```

min	max
-----+-----	-----+-----
2012-02-06 21:00:00	2012-02-06 23:00:00

Агрегатные функции полезны, но сами по себе ограничены. Чтобы подсчитать число мероприятий в каждом месте проведения, можно было бы, конечно, написать запросы для каждого идентификатора места проведения:

```
SELECT count(*) FROM events WHERE venue_id = 1;
SELECT count(*) FROM events WHERE venue_id = 2;
SELECT count(*) FROM events WHERE venue_id = 3;
SELECT count(*) FROM events WHERE venue_id IS NULL;
```

Но, согласитесь, это становится утомительно (и даже вряд ли возможно), когда количество строк растет. На помощь приходит команда `GROUP BY`.

Группировка

Фраза `GROUP BY` позволяет одним махом выполнить все приведенные выше запросы. С ее помощью вы просите PostgreSQL сгруппировать строки, а затем применить к отдельным группам ту или иную агрегатную функцию (например, `count()`).

```
SELECT venue_id, count(*)
FROM events
GROUP BY venue_id;
```

venue_id	count
1	1
2	2
3	1
	3

Совсем неплохо, но можно ли отфильтровать этот список по значениям функции `count()`? А как же! В паре с фразой `GROUP BY` идет ключевое слово `HAVING`, которое работает, как `WHERE`, но применяется для фильтрации результатов агрегатных функций (`WHERE` для этой цели не годится).

В следующем запросе отбираются наиболее популярные места проведения – те, которым соответствует два или более мероприятий:

```
SELECT venue_id
FROM events
GROUP BY venue_id
HAVING count(*) >= 2 AND venue_id IS NOT NULL;
```

venue_id	count
2	2

Можно использовать `GROUP BY` вообще без агрегатных функций. Если написать `SELECT...FROM...GROUP BY имя одного столбца`, то будут возвращены все уникальные значения в этом столбце.

```
SELECT venue_id FROM events GROUP BY venue_id;
```

Подобная группировка встречается настолько часто, что в `SQL` для нее предусмотрено специальное ключевое слово `DISTINCT`.

```
SELECT DISTINCT venue_id FROM events;
```

Результаты обоих запросов одинаковы.

GROUP BY в MySQL

Попытавшись выполнить в `MySQL` запрос, в котором во фразе `GROUP BY` не упоминаются столбцы, присутствующие в `SELECT`, вы с удивлением обнаружите, что он работает. Поначалу этот факт заставил нас усомниться в необходимости оконных функций. Но внимательно изучив, что именно возвращает `MySQL`, мы поняли, что выбирается только случайная строка данных вместе со счетчиком, а не все результаты. Вообще говоря, это бесполезно (а иногда и весьма опасно).

Оконные функции

Если вам доводилось работать над производственным проектом, в котором использовалась реляционная база данных, то с агрегатными запросами вы, скорее всего, знакомы. Это типичный для SQL механизм. С другой стороны, *оконные функции* далеко не так распространены (PostgreSQL – одна из немногих СУБД с открытым исходным кодом, в которой они реализованы).

Оконные функции похожи на запросы с `GROUP BY` в том смысле, что позволяют применять агрегатные функции к нескольким строкам. Различие же в том, что они позволяют использовать встроенные агрегатные функции, не требуя, чтобы каждое поле включалось только в одну строку сгруппированного результата.

Попытка выбрать в следующем запросе столбец `title`, не группируя по нему, должна привести к ошибке:

```
SELECT title, venue_id, count(*)
FROM events
GROUP BY venue_id;
```

```
ERROR: column "events.title" must appear in the GROUP BY clause or \
      be used in an aggregate function
```

Мы подсчитываем строки с одинаковым значением `venue_id`, но у мероприятий с названиями (`title`) *LARP Club* и *Wedding* значение `venue_id` одно и то же. PostgreSQL не знает, *какое* из них отображать.

Если `GROUP BY` возвращает по одной записи на каждое группируемое значение, то оконная функция может вернуть несколько записей для каждой строки. Наглядно это представлено на рис. 4. Рассмотрим пример, демонстрирующий, для чего могут быть полезны оконные функции.

Оконная функция возвращает все удовлетворяющие запросу строки, реплицируя результаты агрегатных функций.

```
SELECT title, count(*) OVER (PARTITION BY venue_id) FROM events;
```

Мы предпочитаем рассматривать фразу `PARTITION BY` как близкий аналог `GROUP BY`, но только вместо группировки результатов по столбцам, указанным в `SELECT` вне агрегатных функций (что приводит к уменьшению общего количества строк в результирующем множестве) она возвращает результаты вычисления агрегатных функций, как любое другое поле (то есть вычисление производится, но результат считается просто еще одним атрибутом). В терминологии SQL оконная функция возвращает результаты применения агрегатной функции к *разделу* (`OVER PARTITION`) результирующего множества.

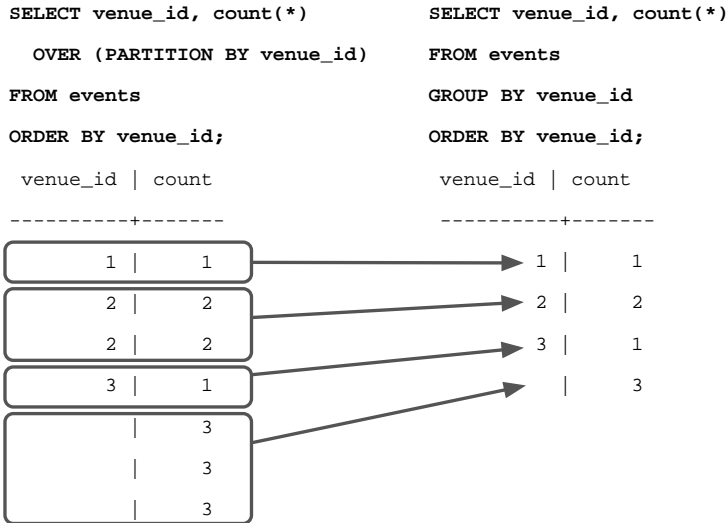


Рис. 4. Оконная функция не свертывает результаты, принадлежащие одной группе

Транзакции

Транзакции – это основа непротиворечивости в реляционных базах данных. *Все или ничего* – вот девиз транзакций. Транзакция гарантирует выполнение всех команд, входящих в группу. Если хотя бы одна завершается с ошибкой, то все команды откатываются, как будто они вообще не выполнялись.

В PostgreSQL транзакции обладают свойствами ACID, то есть являются атомарными (либо успешно выполнены все операции, либо не выполнена ни одна), непротиворечивыми (данные всегда находятся в корректном состоянии – противоречивые состояния не видны), изолированными (одна транзакция не влияет на работу другой) и долговечными (результаты зафиксированной транзакции сохраняются даже после аварийного отказа сервера). Отметим, что слово *consistency* в аббревиатурах ACID и CAP⁴ (см. приложение 2 «Теорема CAP») означает разные вещи.

Любую транзакцию можно заключить в блок `BEGIN TRANSACTION`. Чтобы убедиться в атомарности, отменим транзакцию командой `ROLLBACK`.

⁴ Consistency, Availability, partition Tolerance (согласованность, доступность, устойчивость к потере связности). *Прим. перев.*

Неявные транзакции

Все команды, которых мы до сих пор выполняли в `psql`, были частью неявной транзакции. Если, например, было начато выполнение команды `DELETE FROM account WHERE total < 20;`, и база данных, не довершив удаление до конца, «грохнулась», то вы не останетесь с наполовину стертой таблицей. После перезапуска сервера команда будет откатена.

```
BEGIN TRANSACTION;  
DELETE FROM events;  
ROLLBACK;  
SELECT * FROM events;
```

Все мероприятия остались на месте. Транзакции полезны, когда модифицируются две или более таблицы, которые должны быть согласованы между собой. Классический пример – система перевода денег с одного счета на другой в банке.

```
BEGIN TRANSACTION;  
UPDATE account SET total=total+5000.0 WHERE account_id=1337;  
UPDATE account SET total=total-5000.0 WHERE account_id=45887;  
END;
```

Если между двумя операциями обновления что-то случится с сервером, то банк потеряет пять тысяч. Если же поместить их в одну транзакцию, то первая операция будет просто откатена.

Хранимые процедуры

Все рассмотренные до сих пор команды были декларативными, но иногда желательно выполнить какой-то код. И тут надо решить, где его выполнять – на стороне клиента или на стороне базы данных.

Хранимые процедуры могут предложить радикальный выигрыш в производительности в обмен на не менее радикальный архитектурный компромисс. Можно избежать передачи клиенту тысяч строк, если вы готовы намертво связать код приложения с конкретной базой данных. Решение использовать хранимые процедуры не следует принимать с легким сердцем.

Но оставим в стороне предостережения и создадим процедуру (`FUNCTION`), которая упростит вставку нового события, ассоциированного с некоторым местом проведения, не требуя знания `venue_id`. Если место проведения еще не существует, процедура создаст его и сошлется на него из нового мероприятия. В качестве дополнительного удобства для пользователя мы будем возвращать булево значение, показывающее, было добавлено новое место проведения или нет.

postgres/add_event.sql

```
CREATE OR REPLACE FUNCTION add_event( title text, starts timestamp,
  ends timestamp, venue text, postal varchar(9), country char(2) )
RETURNS boolean AS $$
DECLARE
  did_insert boolean := false;
  found_count integer;
  the_venue_id integer;
BEGIN
  SELECT venue_id INTO the_venue_id
  FROM venues v
  WHERE v.postal_code=postal AND v.country_code=country AND
        v.name ILIKE venue
  LIMIT 1;

  IF the_venue_id IS NULL THEN
    INSERT INTO venues (name, postal_code, country_code)
    VALUES (venue, postal, country)
    RETURNING venue_id INTO the_venue_id;

    did_insert := true;
  END IF;

  -- Примечание: не "error", как в некоторых языках программирования
  RAISE NOTICE 'Venue found %', the_venue_id;

  INSERT INTO events (title, starts, ends, venue_id)
  VALUES (title, starts, ends, the_venue_id);

  RETURN did_insert;
END;
$$ LANGUAGE plpgsql;
```

О зависимости от поставщика

Когда реляционные базы данных находились в зените славы, они стали буквально швейцарским ножом, сочетающим все технологии. Хранить можно было едва ли не всё что угодно, – даже целые программные проекты (например, в Microsoft Access). Немногие компании, которые поставляли такое программное обеспечение, всячески продвигали нестандартные различия, а затем пожинали плоды зависимости корпораций от своих продуктов, назначая непомерную плату за лицензии и консультативные услуги. То была пора проклинаемой *зависимости от поставщика*, от которой пытались избавиться более поздние методологии программирования, появившиеся в конце 1990-х – начале 2000-х годов.

Однако стремление умерить аппетиты поставщиков породило такие максимы, как «никакой логики в базе данных». Но это же позор, ведь в реляционные базы данных встроено столько возможностей по управлению данными.

Зависимость от поставщика не исчезла. Многие рассматриваемые в этой книге действия сильно зависят от конкретной реализации. Однако имеет смысл сначала познакомиться с тем, на что способна база данных, а только потом принимать решение об отказе от таких средств, как хранимые процедуры.

Этот внешний файл можно импортировать в текущую схему, задав его имя в командной строке (если вы, конечно, не собираетесь набирать весь приведенный выше код).

```
book=# \i add_event.sql
```

Следующая команда должна вернуть значение `t` (`true`), поскольку в таблице `venues` раньше не было места проведения *Run's House*. Таким образом, мы вместо двух обращений к базе данных со стороны клиента (`select`, затем `insert`) обошлись только одним.

```
SELECT add_event('House Party', '2012-05-03 23:00',  
                '2012-05-04 02:00', 'Run''s House', '97205', 'us');
```

Эта процедура написана на языке PL/pgSQL (Procedural Language/PostgreSQL). Детальное его рассмотрение выходит за рамки этой книги, но в онлайн-овой документации по PostgreSQL он описан очень подробно⁵.

Помимо PL/pgSQL, ядро Postgres поддерживает написание процедур на языках Tcl, Perl и Python. Однако энтузиасты написали расширения еще для десятка языков, включая Ruby, Java, PHP, Scheme и другие, перечисленные в документации. Попробуйте выполнить такую команду в оболочке:

```
$ createlang book --list
```

Она напечатает список языков, установленных вместе с вашей базой данных. Команда `createlang` используется также для добавления новых языков, которые можно найти в сети⁶.

Триггеры

Триггер автоматически запускает хранимую процедуру, когда происходят определенные события, например вставка или обновление. Это позволяет базе данных гарантировать требуемое поведение в ответ на изменение данных.

Давайте напишем на языке PL/pgSQL еще одну процедуру, которая будет протоколировать факт изменения мероприятия (мы не хотим, чтобы кто-то мог внести изменение в описание мероприятия,

5 <http://www.postgresql.org/docs/9.0/static/plpgsql.html>

6 <http://www.postgresql.org/docs/9.0/static/app-createlang.html>

а потом говорить, что этого не делал). Для начала создадим таблицу `logs` для хранения информации об изменении мероприятий. Первичный ключ в данном случае не нужен, так как это всего лишь журнал.

```
CREATE TABLE logs (
    event_id integer,
    old_title varchar(255),
    old_starts timestamp,
    old_ends timestamp,
    logged_at timestamp DEFAULT current_timestamp
);
```

Затем напишем процедуру, которая будет вставлять в журнал старые данные. Переменная `OLD` представляет состояние строки до изменения (а переменная `NEW`, как мы вскоре увидим, – состояние строки после изменения). Перед возвратом выведем на консоль сообщение, содержащее `event_id`.

Где выполнять код?

Это первое из многих мест в этой книге, где приходится принимать трудное решение: должен ли код быть частью приложения или базы данных? Единого рецепта нет, ответ зависит от конкретного приложения.

Преимущество хранимых процедур в том, что они могут повысить производительность на порядок. Например, предположим, что в приложении нужно произвести сложное вычисление, требующее написания кода. Если в вычислении участвует много строк, то реализация его в виде хранимой процедуры позволит передавать клиенту лишь результат, а не тысячи исходных строк. Но за это придется заплатить – разделением приложения, кода и тестов на части, подразумевающие различные программные парадигмы.

postgres/log_event.sql

```
CREATE OR REPLACE FUNCTION log_event() RETURNS trigger AS $$
DECLARE
BEGIN
    INSERT INTO logs (event_id, old_title, old_starts, old_ends)
    VALUES (OLD.event_id, OLD.title, OLD.starts, OLD.ends);
    RAISE NOTICE 'Someone just changed event #%', OLD.event_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

И напоследок создадим триггер, который запροтоколирует изменения после обновления любой строки.

```
CREATE TRIGGER log_events
AFTER UPDATE ON events
FOR EACH ROW EXECUTE PROCEDURE log_event();
```

Выясняется, что вечер в Run's House заканчивается раньше, чем мы рассчитывали. Изменим описание мероприятия.

```
UPDATE events
SET ends='2012-05-04 01:00:00'
WHERE title='House Party';
```

NOTICE: Someone just changed event #9

Старое время окончания запротоколировано.

```
SELECT event_id, old_title, old_ends, logged_at
FROM logs;
```

event_id	old_title	old_ends	logged_at
9	House Party	2012-05-04 02:00:00	2011-02-26 15:50:31.939

Можно также создавать триггеры, которые срабатывают до обновления или до либо после вставки⁷.

Представление о мире

А неплохо было бы, если бы могли использовать результаты сложного запроса просто как еще одну таблицу? Именно для этого и предназначены представления (VIEW). Но в отличие от хранимых процедур это не исполняемые функции, а псевдонимы запросов.

В нашей базе данных названия всех праздников содержат слово *Day*, а место проведения для них не задано.

postgres/holiday_view_1.sql

```
CREATE VIEW holidays AS
SELECT event_id AS holiday_id, title AS name, starts AS date
FROM events
WHERE title LIKE '%Day%' AND venue_id IS NULL;
```

Как видите, создать представление просто – нужно лишь записать сам запрос и добавить в начало фразу CREATE VIEW имя_представления AS. Теперь представление holidays можно опрашивать как таблицу. Под капотом это все та же таблица events. Чтобы убедиться, добавьте в таблицу events запись о празднике *Valentine's Day* с датой *2012-02-14* и опросите представление holidays.

```
SELECT name, to_char(date, 'Month DD, YYYY') AS date
FROM holidays
WHERE date <= '2012-04-01';
```

⁷ <http://www.postgresql.org/docs/9.0/static/triggers.html>

Name	date
-----+-----	
April Fools Day	April 01, 2012
Valentine's Day	February 14, 2012

Представления очень удобны, когда нужно простым способом получить доступ к результатам сложного запроса. Сколь бы изощренным ни был подспудный запрос, на поверхности мы увидим только таблицу.

Любой добавляемый в представление столбец должен присутствовать в базовой таблице. Изменим таблицу `events`, добавив в нее массив ассоциированных цветов:

```
ALTER TABLE events
ADD colors text ARRAY;
```

Поскольку с праздниками должны быть ассоциированы цвета, изменим определяющий представление запрос, включив массив `colors`:

```
CREATE OR REPLACE VIEW holidays AS
SELECT event_id AS holiday_id, title AS name, starts AS date, colors
FROM events
WHERE title LIKE '%Day%' AND venue_id IS NULL;
```

Теперь осталось только задать для праздника массив строк, содержащих цвета. К сожалению, обновить представление непосредственно невозможно.

```
UPDATE holidays SET colors = '{"red","green"}' where name = 'Christmas Day';
```

```
ERROR: cannot update a view
HINT: You need an unconditional ON UPDATE DO INSTEAD rule.
```

Похоже, нам не обойтись без правила (RULE).

Правила

Правило RULE – это описание способа изменения разобранного *дерева запроса*. Всякий раз как Postgres выполняет команду SQL, порождается дерево запроса (в общем случае его называют *абстрактным синтаксическим деревом*).

Операторы и значения становятся соответственно ветвями и листьями дерева. Перед началом выполнения запроса над деревом можно произвести операции обхода, отсечения ветвей и другие. До отправки планировщику запросов (который также трансформирует дерево с целью оптимизации выполнения) и затем на исполнение дерево мо-

жет быть переписано путем применения правил Postgres (см. рис. 5). Интересно, что любое представление, в частности `holidays`, – это правило.

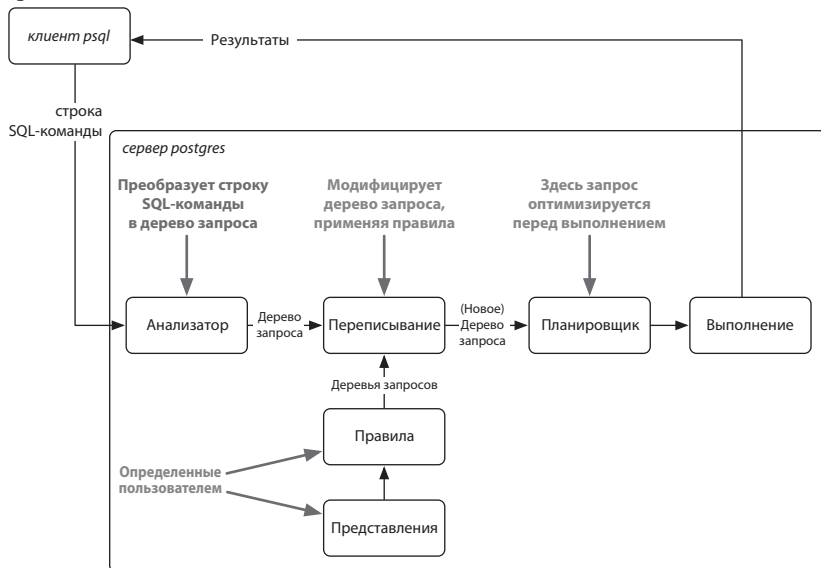


Рис. 5. Как в PostgreSQL выполняется команда SQL

Убедиться в этом можно, взглянув на план выполнения представления `holidays` с помощью команды `EXPLAIN` (отметим, что *Filter* – это фраза `WHERE`, а *Output* – список столбцов).

```
EXPLAIN VERBOSE
SELECT *
FROM holidays;
```

QUERY PLAN

```
-----
Seq Scan on public.events (cost=0.00..1.04 rows=1 width=57)
  Output: events.event_id, events.title, events.starts, events.colors
  Filter: ((events.venue_id IS NULL) AND ((events.title)::text ~ '%Day% '::text))
```

А теперь сравним это с результатом выполнения `EXPLAIN VERBOSE` для запроса, положенного в основу представления `holidays`. Функционально они идентичны.

```
EXPLAIN VERBOSE
SELECT event_id AS holiday_id,
       title AS name, starts AS date, colors
FROM events
```

```
WHERE title LIKE '%Day%' AND venue_id IS NULL;
```

QUERY PLAN

```
-----
Seq Scan on public.events (cost=0.00..1.04 rows=1 width=57)
  Output: event_id, title, starts, colors
  Filter: ((events.venue_id IS NULL) AND ((events.title)::text ~ '%Day% '::text))
```

Таким образом, чтобы разрешить обновление представления `holidays`, мы должны написать правило, которое объяснит Postgres, что делать с командой `UPDATE`. Наше правило будет запоминать все обновления, затребованные для представления `holidays`, и применять их к таблице `events`, получая значения из псевдоотношений `NEW` и `OLD`. Функционально `NEW` интерпретируется как отношение, содержащее новые значения, а `OLD` — как отношение, содержащее значения до обновления.

postgres/create_rule.sql

```
CREATE RULE update_holidays AS ON UPDATE TO holidays DO INSTEAD
  UPDATE events
  SET title = NEW.name,
      starts = NEW.date,
      colors = NEW.colors
  WHERE title = OLD.name;
```

Имея такое правило, мы можем обновить представление `holidays` непосредственно.

```
UPDATE holidays SET colors = '{"red","green"}' where name = 'Christmas Day';
```

Теперь попробуем вставить в `holidays` праздник *New Years Day* с датой *2013-01-01*. Как и следовало ожидать, для этого тоже нужно правило. Не проблема.

```
CREATE RULE insert_holidays AS ON INSERT TO holidays DO INSTEAD
  INSERT INTO ...
```

Мы собираемся двинуться дальше, но если вам хочется еще поэкспериментировать с правилами, можете добавить правило `DELETE RULE`.

Создание сводных таблиц с помощью `crosstab()`

Последнее на сегодня упражнение будет заключаться в создании календаря мероприятий, где для каждого месяца в году будет подсчитано количество мероприятий в этом месяце. Такого рода задачи обыч-

но решаются с помощью *сводных таблиц* (pivot table), позволяющих «свести» сгруппированные данные относительно результата какой-то другой операции – в нашем случае списка месяцев. Мы построим сводную таблицу с помощью функции `crosstab()`.

Сначала напомним запрос, который будет подсчитывать количество мероприятий в каждом месяце по годам. В PostgreSQL имеется функция `extract()`, которая выделяет компонент даты или временной метки; она пригодится для группировки.

```
SELECT extract(year from starts) as year,  
       extract(month from starts) as month, count(*)  
FROM events  
GROUP BY year, month;
```

Чтобы можно было воспользоваться функцией `crosstab()`, запрос должен возвращать три столбца: идентификатор строки, категория и значение. В качестве идентификатора строки мы возьмем год, в качестве категории – месяц, а в качестве значения – счетчик.

Функции `crosstab()` необходим еще один набор значений для представления месяцев. Именно таким образом функция узнает, сколько необходимо столбцов. Эти значения станут столбцами (таблицей, относительно которой производится сведение). Создадим временную таблицу для хранения списка чисел.

```
CREATE TEMPORARY TABLE month_count(month INT);  
INSERT INTO month_count VALUES (1), (2), (3), (4), (5), (6), (7), (8), (9),  
                                (10), (11), (12);
```

Вот теперь всё готово для вызова функции `crosstab()`, которой нужно передать два запроса.

```
SELECT * FROM crosstab(  
    'SELECT extract(year from starts) as year,  
      extract(month from starts) as month, count(*)  
    FROM events  
    GROUP BY year, month',  
    'SELECT * FROM month_count'  
);
```

ERROR: a column definition list is required for functions returning "record"

Ой. Ошибка.

Загадочный текст на самом деле означает, что функция возвращает набор записей (строк), но не знает, как их пометить. Собственно, она даже не знает типы соответствующих данных.

Напомним, что в сводной таблице категориями служат месяцы, но месяцы – это просто числа. Вот и определим их как числа:

```
SELECT * FROM crosstab(
    'SELECT extract(year from starts) as year,
        extract(month from starts) as month, count(*)
    FROM events
    GROUP BY year, month',
    'SELECT * FROM month_count'
) AS (
    year int,
    jan int, feb int, mar int, apr int, may int, jun int,
    jul int, aug int, sep int, oct int, nov int, dec int
) ORDER BY YEAR;
```

У нас имеется столбец `year` (идентификатор строки) и еще двенадцать столбцов, представляющих месяцы.

year	jan	feb	mar	apr	may	jun	jul	aug	sep	oct	nov	dec
2012		5		1	1							1

Добавьте еще несколько мероприятий с датами в каком-нибудь другом году, чтобы посмотреть, как отображаются мероприятия за разные годы. Снова выполните запрос и насладитесь календарем.

День 2: итоги

Сегодня мы закончили обзор основных возможностей PostgreSQL. Вы уже, наверное, понимаете, что Postgres – не просто сервер для хранения данных примитивных типов и предъявления к ним запросов; это система управления данными, которая умеет переформатировать данные на выходе, хранить данные экзотических типов, например массивы, исполнять код и переписывать входные запросы.

День 2: домашнее задание

Информационный поиск

1. Найдите в документации по PostgreSQL информацию об агрегатных функциях.
2. Найдите клиентскую программу для PostgreSQL с графическим интерфейсом, например Navicat.

Задачи

1. Напишите правило, которое перехватывает команды DELETE, обращенные к таблице `venues` и вместо удаления устанавливает флаг `active` (добавленный в домашнем задании для первого дня) в FALSE.

2. Временная таблица – не лучший способ реализовать сводную таблицу, представляющую календарь. Функция `generate_series(a, b)` возвращает последовательность целых чисел от `a` до `b` в виде набора записей. Воспользуйтесь ей вместо выборки из таблицы `month_count`.
3. Постройте сводную таблицу, в которой представлены все числа одного месяца, причем строки соответствуют неделям, а столбцы – дням (начиная с воскресенья и кончая субботой), как в обычном календаре. Для каждого дня значение должно содержать количество событий, запланированных на эту дату, или `NULL`, если ни одного события не запланировано.

2.4. День 3: полнотекстовый поиск и многомерные кубы

Третий день мы посвятим исследованию разнообразных имеющихся в нашем распоряжении инструментов для построения системы поиска фильмов. Начнем с нечеткого поиска по имени актера или названию фильма – PostgreSQL позволяет делать это разными способами. Затем изучим пакет `cube` на примере системы recommendations фильмов, основанной на фильмах похожих жанров, которые уже признаны хорошими. Поскольку все рассматриваемые здесь пакеты дополнительные, то реализация является спецификой PostgreSQL и не имеет никакого отношения к стандарту SQL.

Обычно проектирование схемы реляционной базы данных начинают с рисования диаграммы сущностей. На рис. 6 изображена модель нашей системы рекомендаций фильмов, которая ведет учет фильмам, их жанрам и играющим в них актерам.

Напомним, что в первый день мы установили несколько дополнительных пакетов. Сегодня все они нам понадобятся. На всякий случай перечислим, что должно быть установлено: `tablefunc`, `dict_xsyn`, `fuzzystrmatch`, `pg_trgm`, `cube`.

Первым делом создадим базу данных. Всегда полезно строить индексы по внешним ключам, так как это ускоряет выполнение некоторых запросов (например, найти фильмы, в которых снимался данный актер). Кроме того, следует задавать ограничение `UNIQUE` для связующих таблиц типа `movies_actors`, чтобы избежать появления дубликатов в соединениях.

```
postgres/create_movies.sql
CREATE TABLE genres (
```

```

        name text UNIQUE,
        position integer
    );
CREATE TABLE movies (
    movie_id SERIAL PRIMARY KEY,
    title text,
    genre cube
);
CREATE TABLE actors (
    actor_id SERIAL PRIMARY KEY,
    name text
);
CREATE TABLE movies_actors (
    movie_id integer REFERENCES movies NOT NULL,
    actor_id integer REFERENCES actors NOT NULL,
    UNIQUE (movie_id, actor_id)
);
CREATE INDEX movies_actors_movie_id ON movies_actors (movie_id);
CREATE INDEX movies_actors_actor_id ON movies_actors (actor_id);
CREATE INDEX movies_genres_cube ON movies USING gist (genre);

```

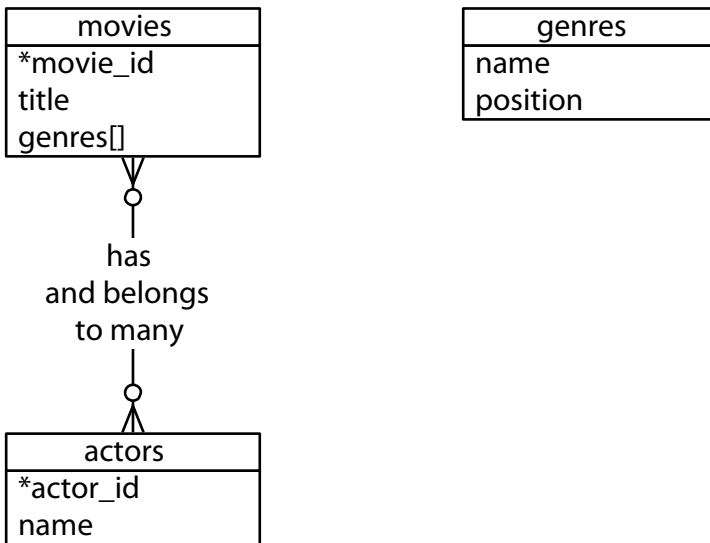


Рис. 6. Наша система рекомендаций фильмов

Вы можете скачать файл `movies_data.sql` с сопроводительно-го сайта книги и, подав его на стандартный вход утилиты `psql`, заполнить таблицы данными. Ответы на вопросы, касающиеся строки `genre cube`, будут даны ниже.

Нечеткий поиск

Разрешить в системе полнотекстовый поиск – значит позволить вводить неточные данные. Следует быть готовым к вводу названия «Brid of Frankstein» вместо «Bride of Frankenstein»⁸. Кто-то не сможет вспомнить полное имя Джулии Робертс и напишет «J. Roberts». А кто-то и вовсе не знает, как правильно пишется имя Бена Аффлека и введет «Benn Aflek» вместо «Ben Affleck». Мы рассмотрим несколько пакетов для PostgreSQL, которые упрощают поиск в таких случаях. Стоит сразу отметить, что такой поиск строк размывает грань между реляционными запросами и поисковыми системами типа Lucene⁹. Некоторые считают, что функции вроде полнотекстового поиска следует рассматривать как часть приложения, но включение этих пакетов в СУБД – туда, где находятся сами данные, – может дать заметные преимущества в части производительности и удобства администрирования.

Поиск строк в стандарте SQL

В PostgreSQL есть много способов поиска в тексте, но основными, присутствующими по умолчанию являются оператор LIKE и регулярные выражения.

Операторы LIKE и ILIKE

Простейшие виды текстового поиска реализуются операторами LIKE и ILIKE (вариант LIKE без учета регистра). Они имеются практически во всех реляционных базах данных. LIKE сопоставляет значения в столбце с указанным образцом. Символы % и _ являются метасимволами, причем % сопоставляется с произвольным числом символов, а _ – ровно с одним.

```
SELECT title FROM movies WHERE title ILIKE 'stardust%';
```

```

      title
-----
Stardust
Stardust Memories
```

Если требуется, чтобы подстрока *stardust* находилась не в конце строки, то можно применить нехитрый трюк – добавить в образец символ _:

```
SELECT title FROM movies WHERE title ILIKE 'stardust_%';
```

⁸ «Невеста Франкенштейна» – фильм Джеймса Уэйла, 1935 год. *Прим. перев.*

⁹ <http://lucene.apache.org/>



```
title
-----
Stardust Memories
```

В простых случаях оператор LIKE может быть полезен, но только если двух указанных метасимволов достаточно.

Регулярные выражения

Более мощным механизмом сопоставления с образцом являются *регулярные выражения*. Они часто будут встречаться на страницах этой книги, потому что поддерживаются многими базами данных. Написанию регулярных выражений посвящены целые книги – эта тема слишком сложна и обширна, чтобы рассматривать ее здесь во всей полноте. Postgres придерживается синтаксиса регулярных выражений, описанного в стандарте POSIX (по большей части).

В Postgres для сопоставления с регулярным выражением применяется оператор ~ с необязательным предшествующим знаком ! (который означает *несоответствие*) и последующим знаком * (без учета регистра). Таким образом, следующий запрос подсчитывает количество фильмов, названия которых *не* начинаются словом *the* без учета регистра:

```
SELECT COUNT(*) FROM movies WHERE title !~* '^the.*';
```

Строка, заключенная в кавычки, – регулярное выражение.

Текстовые столбцы, участвующие в такого рода запросах, можно проиндексировать, создав индекс операторного класса `text_pattern_ops` при условии, что значения представлены в нижнем регистре:

```
CREATE INDEX movies_title_pattern ON movies (lower(title) text_pattern_ops);
```

Мы указали класс `text_pattern_ops`, потому что поле `title` имеет тип `text`. Для индексирования столбцов типа `varchar`, `char` или `name` служат соответственно классы `varchar_pattern_ops`, `bpchar_pattern_ops` и `name_pattern_ops`.

Расстояние Левенштейна

Расстояние Левенштейна – это мера похожести двух строк, оно определяется как количество *шагов*, необходимое для преобразования одной строки в другую. Шагом считается замена, добавление и удаление одного символа. В PostgreSQL функция `levenshtein()` входит в дополнительный пакет `fuzzystrmatch`. Пусть имеются строки *bat* и *fads*. Тогда запрос


```
SELECT levenshtein('bat', 'fads');
```

вернет 3, потому что для перехода от *bat* нужно две буквы заменить ($b \Rightarrow f$, $t \Rightarrow d$) и одну добавить ($+s$). Каждое изменение увеличивает расстояние на единицу. Изменение расстояния можно наблюдать, подойдя поближе (образно говоря). Расстояние уменьшается, пока не достигнет нуля (в этом случае строки совпадают):

```
SELECT levenshtein('bat', 'fad') fad,
       levenshtein('bat', 'fat') fat,
       levenshtein('bat', 'bat') bat;
```

```
fad | fat | bat
-----+-----+-----
  2 |   1 |   0
```

Изменение регистра также считается шагом, поэтому при формулировании запроса лучше преобразовать все строки к одному регистру.

```
SELECT movie_id, title FROM movies
WHERE levenshtein(lower(title), lower('a hard day nght')) <= 3;
```

```
movie_id | title
-----+-----
    245 | A Hard Day's Night
```

При таком подходе несущественные различия не будут приводить к увеличению расстояния.

Триграммы

Триграммой называется группа из трех последовательных символов в строке. Дополнительный модуль `pg_trgm` выделяет из строки все возможные триграммы.

```
SELECT show_trgm('Avatar');

show_trgm
-----
{" a"," av"," ar ","ata,ava,tar,vat}
```

Поиск наиболее подходящей строки сводится к подсчету совпадающих триграмм. Чем больше совпадений, тем более похожи строки. Этот метод полезен, когда нужно искать строку, игнорируя мелкие расхождения в орфографии или даже короткие пропущенные слова. Чем длиннее строка, тем больше в ней триграмм и тем вероятнее соответствие. Для поиска фильмов по названиям отлично подходит, так как длины названий примерно одинаковы.



Сначала мы создадим индекс по названиям фильмов типа Generalized Index Search Tree (GIST) (обобщенное индексное дерево поиска). Такой тип индексов включен в ядро PostgreSQL по умолчанию.

```
CREATE INDEX movies_title_trigram ON movies
USING gist (title gist_trgm_ops);
```

Теперь, даже если искомая строка содержит опечатки, мы все равно получим приемлемые результаты.

```
SELECT *
FROM movies
WHERE title % 'Avatre';
```

```
title
-----
Avatar
```

Триграммы стоит рекомендовать, когда нужно получить от пользователя входную информацию, не обременяя его сложностями семантики метасимволов.

Полнотекстовый поиск

Далее мы хотим дать пользователю возможность производить полнотекстовый поиск без учета форм множественного числа. Если пользователь хочет найти фильм, зная только некоторые слова в его названии, то Postgres сможет помочь, так как поддерживает простые средства обработки естественного языка.

TSVector и TSQuery

Давайте поищем фильмы, в названиях которых есть слова *night* и *day*¹⁰. Это идеальная задача для полнотекстового движка — путем использования оператора запроса @@.

```
SELECT title
FROM movies
WHERE title @@ 'night & day';
```

```
title
-----
A Hard Day's Night
Six Days Seven Nights
Long Day's Journey Into Night
```

¹⁰ Далее перечисляются фильмы, которые в российском прокате называются «Вечер трудного дня», «Шесть дней, семь ночей», «Долгое путешествие в ночь». *Прим. пер.*

Этот запрос возвращает название *A Hard Day's Night* несмотря на то, что слово *Day* записано в притяжательном падеже и два слова переставлены. Оператор @@ преобразует поле названия в тип `tsvector`, а сам запрос – в тип `tsquery`.

Тип данных `tsvector` представляет строку в виде массива (или *вектора*) лексем, которые сравниваются со строкой, указанной в запросе, а тип `tsquery` представляет запрос на некотором естественном языке, например, английском или французском. Языку соответствует словарь (о котором мы будем говорить чуть ниже). Так, показанный выше запрос эквивалентен следующему (в предположении, что система настроена на английский язык):

```
SELECT title
FROM movies
WHERE to_tsvector(title) @@ to_tsquery('english', 'night & day');
```

Посмотреть, как разбиваются на компоненты вектор и запрос, можно, выполнив функции преобразования непосредственно:

```
SELECT to_tsvector('A Hard Day's Night'),
       to_tsquery('english', 'night & day');
```

```

           to_tsvector           |           to_tsquery
-----+-----
'day':3 'hard':2 'night':5 | 'night' & 'day'
```

С лексемами, образующими `tsvector`, связаны позиции в исходной фразе.

Возможно, вы обратили внимание, что `tsvector` для строки *A Hard Day's Night* не содержит лексемы *a*. Более того, простые английские слова вроде артикля *a* при поиске вообще игнорируются.

```
SELECT *
FROM movies
WHERE title @@ to_tsquery('english', 'a');
```

```
NOTICE: text-search query contains only stop words or doesn't \
contain lexemes, ignored
```

Часто встречающиеся слова, в частности *a*, называются *стоп-словами*; как правило, искать по ним не имеет смысла. Для нормализации поискового запроса, то есть выделения из него значимых компонентов, анализатор использует словарь английского языка. Следующая команда выводит на консоль список стоп-слов для английского языка, который хранится в файле в каталоге `tsearch_data`:

```
cat `pg_config --sharedir`/tsearch_data/english.stop
```

При желании можно удалить слово *a* из списка или использовать другой словарь, например `simple`, который просто разбивает строку на слова и преобразует их в нижний регистр. Сравните следующие два вектора:

```
SELECT to_tsvector('english', 'A Hard Day''s Night');
```

```
          to_tsvector
-----
'day':3 'hard':2 'night':5
```

```
SELECT to_tsvector('simple', 'A Hard Day''s Night');
```

```
          to_tsvector
-----
'a':1 'day':3 'hard':2 'night':5 's':4
```

Используя словарь `simple`, можно найти любой фильм, в названии которого встречается лексема *a*.

Другие языки

Понятно, что обработка естественного языка зависит от заданных параметров. Все установленные конфигурации можно увидеть, выполнив команду

```
book=# \dF
```

Для вычисления вектора лексем Postgres использует словари, а также списки стоп-слов и другие не рассматриваемые здесь правила, которые называются *анализаторами* и *шаблонами*. Просмотреть список установленных в системе словарей и других языковых компонентов позволяет команда

```
book=# \dFd
```

Любой словарь можно протестировать непосредственно, вызвав функцию `ts_lexize()`. Ниже мы получаем основу строки *Day's*.

```
SELECT ts_lexize('english_stem', 'Day''s');
```

```
ts_lexize
-----
{day}
```

Описанные выше команды полнотекстового поиска работают и для других языков. Если в системе установлен немецкий язык, попробуйте такую команду:

```
SELECT to_tsvector('german', 'was machst du gerade?');
```

```
to_tsvector
```

```
-----
'gerad':4 'mach':2
```

Поскольку слова *was* (что) и *du* (ты) встречаются очень часто, то в немецком словаре они объявлены стоп-словами, а из слов *machst* (делаешь) и *gerade* (сейчас) выделены основы.

Индексирование лексем

Полнотекстовый поиск – могучая штука. Но если таблицы не индексируются, то работать он будет медленно. Команда `EXPLAIN` позволяет узнать детали плана выполнения запроса.

```
EXPLAIN
SELECT *
FROM movies
WHERE title @@ 'night & day';
```

```
QUERY PLAN
```

```
-----
Seq Scan on movies (cost=10000000000.00..10000000001.12 rows=1 width=68)
  Filter: (title @@ 'night & day'::text)
```

Обратите внимание на строку *Seq Scan on movies*. Это плохой знак, потому что свидетельствует о полном сканировании таблицы, то есть СУБД должна будет прочитать каждую строку. А, значит, необходим подходящий индекс.

Индекс по значениям лексем имеет тип Generalized Inverted iNdex (GIN, обобщенный инвертированный индекс). Как и GIST, этот тип встроен в ядро. Слова *инвертированный индекс* должны быть знакомы тем, кому доводилось работать с поисковыми системами, например Lucene или Sphinx. Это стандартная структура данных, применяемая для полнотекстового поиска.

```
CREATE INDEX movies_title_searchable ON movies
USING gin(to_tsvector('english', title));
```

Построив индекс, попробуем снова выполнить поиск.

```
EXPLAIN
SELECT *
FROM movies
WHERE title @@ 'night & day';
```

```
QUERY PLAN
```

```
-----
Seq Scan on movies (cost=10000000000.00..10000000001.12 rows=1 width=68)
  Filter: (title @@ 'night & day'::text)
```

Что же мы видим? Да ничего. Индекс есть, но Postgres его не использует. Всё дело в том, что при построении GIN-индекса мы указали английский язык (`english`), но в запросе об этом ничего не сказали. Необходимо указать язык во фразе `WHERE`.

```
EXPLAIN
SELECT *
FROM movies
WHERE to_tsvector('english',title) @@ 'night & day';
```

Команда `EXPLAIN` способна оказать неоценимую помощь, когда нужно убедиться, что индексы используются в соответствии с ожиданиями. Если это не так, то индекс – просто лишняя нагрузка на систему.

Метафоны

Мы шаг за шагом продвигаемся к своей цели – искать по неточно заданным критериям. Оператор `LIKE` и регулярные выражения требуют задания специально составленных образцов для сопоставления со строками. Расстояние Левенштейна позволяет находить строки с незначительными орфографическими ошибками, но все же искомая строка должна быть очень близка к заданной в запросе. Триграммы хорошо приспособлены для поиска с разумными орфографическими отклонениями. Наконец, полнотекстовый поиск учитывает некоторые особенности естественного языка, в частности умеет игнорировать малозначащие слова типа артиклей *a* и *the* и понимает формы множественного числа. Но иногда мы вообще не знаем, как слово пишется, зато знаем, как оно произносится.

Мы любим Брюса Уиллиса (Bruce Willis) и хотели бы посмотреть фильмы с его участием. Увы, мы никак не можем вспомнить, как правильно пишутся его имя и фамилия, поэтому пытаемся максимально точно воспроизвести ее фонетическое звучание.

```
SELECT *
FROM actors
WHERE name = 'Broos Wlis';
```

Тут даже триграммы не помогут (для их использования знак `=` нужно заменить на `%`):

```
SELECT *
FROM actors
WHERE name % 'Broos Wlis';
```

Тогда на помощь приходят метафоны, то есть алгоритмы для создания строкового представления звучания слова. Вы можете ука-

зять, сколько символов должно быть в выходной строке. Например, *ARNKHRT* – семизначный метафон имени Aaron Eckhart.

Чтобы найти все фильмы, в которых играет актер с именем, звучащим как Broos Wils, можно предъявить запрос к результатам вычисления метафонов. Отметим, что оператор `NATURAL JOIN` – это `INNER JOIN`, в котором соединение автоматически производится по столбцам с одинаковыми именами (например, `movies.actor_id=movies_actors.actor_id`).¹¹

```
SELECT title
FROM movies NATURAL JOIN movies_actors NATURAL JOIN actors
WHERE metaphone(name, 6) = metaphone('Broos Wils', 6);
```

```

           title
-----
The Fifth Element
Twelve Monkeys
Armageddon
Die Hard
Pulp Fiction
The Sixth Sense
:
```

Заглянув в онлайнную документацию, вы обнаружите, что в модуле *fuzzystrmatch* есть и другие функции: `dmetaphone()` (двойной метафон), `dmetaphone_alt()` (для альтернативного произношения имени) и `soundex()` (очень старый алгоритм, который был придуман в 1880-х годах для сравнения типичных американских фамилий при проведении переписей в США). Нетрудно увидеть, что именно возвращают эти функции.

```
SELECT name, dmetaphone(name), dmetaphone_alt(name),
       metaphone(name, 8), soundex(name)
FROM actors;
```

name	dmetaphone	dmetaphone_alt	metaphone	soundex
50 Cent	SNT	SNT	SNT	C530
Aaron Eckhart	ARNK	ARNK	ARNKHRT	A652
Agatha Hurler	AK0R	AKTR	AK0HRL	A236

:

Ни одна из этих функций не лучше всех прочих; какую выбрать, зависит от конкретного набора данных.

¹¹ Перечисленные ниже фильмы в российском прокате называются соответственно «Пятый элемент», «Двенадцать обезьян», «Армагеддон», «Крепкий орешек», «Криминальное чтиво», «Шестое чувство». *Прим. перев.*

Комбинирование разных способов поиска

Выстроив в шеренгу все вышеперечисленные методы поиска строк, мы можем приступить к интересным экспериментам с их комбинированием.

Одно из самых полезных свойств метафонов заключается в том, что они возвращают строковые значения, а, значит, мы можем использовать их в сочетании с другими методами поиска строк. Например, строку, возвращенную функцией `metaphone()` можно разложить на триграммы, а затем упорядочить результаты по наименьшему расстоянию Левенштейна. Иными словами, запросить у системы имена, которые произносятся похоже на *Robin Williams*, и расположить их в порядке убывания похожести.

```
SELECT * FROM actors
WHERE metaphone(name,8) % metaphone('Robin Williams',8)
ORDER BY levenshtein(lower('Robin Williams'), lower(name));
```

actor_id	name
2442	John Williams
4090	Robin Shou
4093	Robin Williams
4479	Steven Williams

Отметим, что результат получился не идеальным. Robin Williams идет под номером 3. Бесконтрольное использование предоставляемой гибкости может давать неожиданные результаты, так что будьте осторожны.

```
SELECT * FROM actors WHERE dmetaphone(name) % dmetaphone('Ron');
```

actor_id	name
3911	Renji Ishibashi
3913	Renje Zellweger

Вариантов комбинирования множество, ограничены они только вашей фантазией.

Жанры как многомерный куб

И последний дополнительный пакет, который мы рассмотрим, — это `cube`. Мы воспользуемся типом данных `cube` для отображения жанров фильмов на многомерный вектор. А затем применим методы, позволяющие эффективно находить ближайшие точки в пределах гиперкуба, для получения списка похожих фильмов.

Вы, наверное, помните, что в начале третьего дня мы создали в таблице столбец `genres` типа `cube`. Каждое значение является точкой в 18-мерном пространстве, в котором оси координат соответствуют жанрам. Зачем представлять киножанры в виде точек в n -мерном пространстве? Жанровое деление – не точная наука, и многие фильмы нельзя назвать ни стопроцентной комедией, ни стопроцентной трагедией – они находятся где-то посередине.

В нашей системе каждому жанру присваивается рейтинг (произвольное число) от 0 до 10 в зависимости от того, насколько точно фильм можно отнести к этому жанру. При этом 0 означает наименьшее, а 10 – наибольшее соответствие.

Вектор жанров для фильма *Star Wars*¹² имеет вид (0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0). В таблице `genres` описываются *позиции* каждого измерения этого вектора. Чтобы понять, что означает вектор жанров, нужно вычислить функцию `cube_ur_coord(vector,dimension)`, используя `genres.position`. Поясним это на примере, отфильтровав жанры с рейтингом 0.

```
SELECT name,
cube_ur_coord('(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)', position) as score
FROM genres g
WHERE cube_ur_coord('(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)', position) > 0;
```

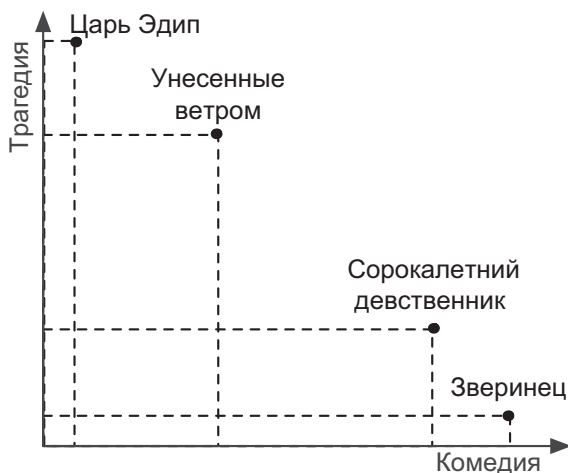
name	score
Adventure	7
Fantasy	7
SciFi	10

Мы будем искать похожие фильмы, отыскивая ближайшие точки. Чтобы понять, почему это работает, представим два фильма на двумерном графике жанров, как на рисунке ниже. Если ваш любимый фильм – *Зверинец*, то вы скорее захотите посмотреть *Сорокалетний девственник*, чем *Царь Эдип* – последний к комедиям никак не отнестись. В нашей двумерной вселенной для поиска вероятных соответствий нужно просто найти ближайшего соседа.

Этот результат можно обобщить на большее число измерений, то есть жанров – 2, 3 или все 18. Принцип один и тот же: ближайший сосед в пространстве жанров дает наилучшее соответствие по жанру.

Ближайшего соседа для заданного вектора жанров можно найти с помощью функции `cube_distance(point1, point2)`. В примере ниже мы выводим расстояния в пространстве жанров от фильма *Star Wars* до всех остальных, сортируя их в порядке возрастания.

¹² Звездные войны. Прим. перев.



```
SELECT *,
       cube_distance(genre, '(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)')
       dist
FROM movies
ORDER BY dist;
```

Ранее при создании таблиц мы построили индекс `movies_genres_cube`. Но даже при наличии этого индекса запрос выполняется сравнительно медленно, так как требует полного сканирования таблицы. Сервер вычисляет расстояние для каждой строки, а затем сортирует результаты.

Вместо того чтобы вычислять расстояния до каждой точки, мы можем оставить только наиболее вероятные, воспользовавшись *ограничивающим кубом*. Идея проста – искать пять ближайших городов на карте штата быстрее, чем на карте всего мира, поскольку, наложив ограничение, мы уменьшаем количество просматриваемых точек.

Функция `cube_enlarge(cube, radius, dimensions)` строит 18-мерный куб с центром в данной точке и стороной заданной длины.

Рассмотрим более простой пример – двумерный квадрат с центром в точке (1,1) и «радиусом» 1. Его левый нижний угол находится в точке (0,0), а правый верхний – в точке (2,2).

```
SELECT cube_enlarge('(1,1)',1,2);

cube_enlarge
-----
(0, 0), (2, 2)
```

Тот же принцип применим к любому числу измерений. Имея ограничивающий гиперкуб, мы можем воспользоваться специальным оператором @>, который означает *содержит*. Следующий запрос вычисляет расстояния от *Star Wars* до всех точек в пространстве жанров, которые заключены внутри куба со стороной 5¹³.

```
SELECT title, cube_distance(genre, '(0,7,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)') dist
FROM movies
WHERE cubeEnlarge('(0,7,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)')::cube, 5, 18) @> genre
ORDER BY dist;
```

title	dist
Star Wars	0
Star Wars: Episode V - The Empire Strikes Back	2
Avatar	5
Explorers	5.74456264653803
Krull	6.48074069840786
E.T. The Extra-Terrestrial	7.61577310586391

Воспользовавшись подзапросом, мы сможем получить жанр по названию фильма и произвести для него вычисления, используя псевдоним таблицы.¹⁴

```
SELECT m.movie_id, m.title
FROM movies m, (SELECT genre, title FROM movies WHERE title = 'Mad Max') s
WHERE cubeEnlarge(s.genre, 5, 18) @> m.genre AND s.title <> m.title
ORDER BY cube_distance(m.genre, s.genre)
LIMIT 10;
```

movie_id	title
1405	Cyborg
1391	Escape from L.A.
1192	Mad Max Beyond Thunderdome
1189	Universal Soldier
1222	Soldier
1362	Johnny Mnemonic
946	Alive
418	Escape from New York
1877	The Last Starfighter
1445	The Rocketeer

13 Перечисленные ниже фильмы в российском прокате называются соответственно «Звездные войны», «Звездные войны – империя наносит ответный удар», «Аватар», «Исследователи», «Крулл», «Инопланетянин». *Прим. перев.*

14 Перечисленные ниже фильмы в российском прокате называются соответственно «Киборг», «Побег из Лос-Анджелеса», «Безумный Макс: под куполом грома», «Солдат», «Джонни-мнемоник», «Живые», «Побег из Нью-Йорка», «Последний звездный боец», «Ракетчик». *Прим. перев.*

Такой метод recommendations фильмов не идеален, но для начала вполне сойдет. В последующих главах мы встретимся и с другими многомерными запросами, в том числе с поиском на двумерной карте в MongoDB.

День 3: итоги

Сегодня мы совершили стремительное путешествие, исследовав гибкие возможности поиска строк в PostgreSQL, и воспользовались пакетом `cube` для многомерного поиска. Но гораздо важнее то, что мы получили представление о нестандартных расширениях, благодаря которым PostgreSQL занимает первое место среди РСУБД с открытым исходным кодом. К вашим услугам еще десятки (если не сотни) пакетов – от хранения географических данных до криптографических функций, от нестандартных типов данных до языковых расширений. Дополнительные пакеты в сочетании с мощью встроенного в ядро языка SQL – вот что выделяет PostgreSQL.

День 3: домашнее задание

Информационный поиск

1. Найдите в онлайн-официальной документации описание всех дополнительных пакетов, поставляемых в составе PostgreSQL.
2. Найдите документацию по регулярным выражениям POSIX (это пригодится в последующих главах)

Задачи

1. Напишите хранимую процедуру, которая принимает название фильма или имя актера и возвращает первые пять рекомендаций, в которые включены либо фильмы, где актер снимался в главной роли, либо фильмы похожих жанров.
2. Включите в базу данных о фильмах возможность хранить комментарии пользователей и извлекать из них ключевые слова (за вычетом английских стоп-слов). Составьте список перекрестных ссылок между ключевыми словами и фамилиями актеров и попытайтесь найти наиболее часто упоминаемых актеров.

2.5. Резюме

Если вы не слишком хорошо знакомы с реляционными базами данных, то мы настоятельно рекомендуем глубже изучить PostgreSQL

или какую-нибудь другую реляционную СУБД, прежде чем искать менее традиционные альтернативы. Реляционные СУБД находились в центре внимания академических кругов и коммерческих организаций более сорока лет, и PostgreSQL – одна из лучших РСУБД с открытым исходным кодом, в которой воплотились все достижения.

Сильные стороны PostgreSQL

Сильных сторон у PostgreSQL столько же, сколько у реляционной модели: многие годы исследований и промышленной эксплуатации практически во всех областях, где применяются компьютеры, гибкие средства запросов, высочайший уровень непротиворечивости и долговечности данных. Для Postgres написаны и проверены в деле драйверы большинства языков программирования. Многие программные модели, в том числе объектно-реляционное отображение (ORM), предполагают, что в основе лежит реляционная СУБД. Главный плюс – гибкость соединения. Не нужно заранее планировать, какие запросы будут предъявляться к модели, поскольку всегда можно воспользоваться соединением, фильтрацией, представлениями и индексами – более чем вероятно, что вы сумеете извлечь интересующие вас данные.

СУБД PostgreSQL широко известна своими «степфордскими данными» (название происходит от фильма «Степфордские жены», в котором рассказывается о городке, где почти все жители похожи друг на друга по форме и содержанию). Под этим понимается тот факт, что данные в высокой степени однородны и согласованы со структурированной схемой.

Кроме того, PostgreSQL выходит далеко за рамки обычной для РСУБД с открытым исходным кодом функциональности, например мощных механизмов ограничений на схему. Вы можете писать собственные языковые расширения, вводить новые типы индексов, создавать собственные типы данных и даже переписывать планы выполнения запросов. И, если для других СУБД с открытым исходным кодом имеются хитроумные лицензионные соглашения, то PostgreSQL – это квинтэссенция открытости. У кода вообще нет владельца. Любой может делать с проектом практически все, что его душе угодно (только не возлагая ответственность на авторов). Разработка и распространение поддерживаются сообществом и только им. Если вы – сторонник бесплатного (и свободно распространяемого) ПО или носите длинную косматую бороду, то должны проявить уважение к их

единодушному сопротивлению коммерциализации этого замечательного продукта.

Слабые стороны PostgreSQL

Хотя реляционные базы данных, без сомнения, добились наибольшего успеха по сравнению со всеми прочими видами СУБД, в некоторых случаях они – не лучшее решение. Сегментирование – не самая сильная сторона реляционных баз и PostgreSQL в том числе. Если требуется масштабирование по горизонтали, а не по вертикали (то есть несколько параллельных хранилищ данных, а не одна супермощная машина или кластер), то лучше поискать в другом месте. Если требования к данным настолько гибкие, что не укладываются в прокрустово ложе требований к схеме, или вас отпугивают издержки на эксплуатацию полномасштабной СУБД, или задача подразумевает очень большое количество операций чтения и записи значений ключа, или требуется хранить только большие двоичные объекты, то, быть может, лучше подойдет какое-нибудь другое хранилище.

Перед расставанием

Реляционная база данных – отличный выбор, и обусловлено это, прежде всего, гибкостью запросов. Хотя PostgreSQL требует проектировать схему заранее, она не делает никаких предположений о том, как данные будут использоваться. Если схема в достаточной степени нормализована, в частности, в ней не хранятся дубликаты или вычисляемые значения, то можете считать себя во всеоружии для предъявления любых запросов. А если включить подходящие модули, оптимально настроить ядро и построить нужные индексы, то система будет работать поразительно быстро на терабайтных объемах данных, потребляя при этом совсем немного ресурсов. И наконец, для тех, кто на первое место ставит безопасность, PostgreSQL предлагает транзакции – полностью атомарные, непротиворечивые, изолированные и долговечные.



ГЛАВА 3.

Riak

Любой человек, хоть немного знакомый со строительством, знает, что арматурный стержень – это стальная балка или прут, предназначенный для усиления бетонных конструкций. Причем одного стержня недостаточно; чтобы система получилась долговечной, необходимо много совместно работающих стержней. Точно так же устроена и база данных Riak (произносится «риак») – каждый компонент дешев и легко заменим, но при правильном использовании трудно найти более простой и в то же время прочный фундамент, на котором возводить все здание.

Riak – это распределенное хранилище ключей и значений, в котором значением может быть что угодно – простой текст, документ в формате JSON или XML, изображение или видеоклип. Для доступа к хранилищу предоставляется простой и единообразный HTTP-интерфейс. С какими бы данными вы ни работали, Riak сможет сохранить их.

Riak также может похвастаться отказоустойчивостью. Любой сервер может быть остановлен или запущен в любой момент, точки общего отказа не существует. Кластер продолжает работать при удалении, добавлении или аварийном отказе (избави Бог, конечно) серверов. Riak позволит отказаться от ночных дежурств, потому что отказ одного узла – не критическая ситуация и вполне может подождать до утра. Один из разработчиков ядра, Джастин Шихи (Justin Sheehy) как-то заметил: «[Команда Riak] усердно работала над такими вещами, как доступность для записи..., чтобы потом спокойно отправиться спать».

Однако подобная гибкость требует компромиссов. В Riak нет хорошей поддержки произвольных запросов, а хранилища ключей и значений, по самой своей природе, плохо связываются друг с другом (иными словами, понятие внешнего ключа отсутствует). Riak атакует эти проблемы с нескольких фронтов; как именно, мы будем разбираться в ближайшие дни.

3.1. Riak дружит с веб

Riak «говорит на языке веб» лучше, чем все остальные рассматриваемые в этой книге базы данных (на втором месте, ненамного отстав, стоит CouchDB). Вы предъявляете запрос с помощью URL, заголовков и глаголов HTTP, а Riak возвращает ресурсы и стандартные коды ответа HTTP.

Riak и cURL

Поскольку задача этой книги – познакомить читателя с семью базами данных и положенными в их основу идеями, а не научить программированию на конкретном языке, мы по возможности стараемся не использовать новые языки. Riak предоставляет REST-интерфейс поверх HTTP, поэтому мы будем обращаться к нему с помощью утилиты cURL. В производственной среде вы, скорее всего, будете пользоваться драйвером своего любимого языка. Использование cURL позволит нам изучить базовый API, не прибегая к конкретному драйверу или языку программирования.

Riak – отличный выбор для центров обработки данных – таких, как Amazon, – которые должны обслуживать много запросов с низкой задержкой. В условиях, когда каждая лишняя миллисекунда ожидания означает потерю потенциального клиента, составить конкуренцию Riak трудно. Она проста в настройке и администрировании и может расти вместе с требованиями. Если вам доводилось работать с веб-службами Amazon, например SimpleDB или S3, то вы легко заметите некоторое сходство в форме и функционировании. Это не случайное совпадение – в основу Riak легли идеи, описанные в статье Amazon о системе Dynamo¹.

В этой главе мы изучим, как Riak хранит и извлекает значения и как связать данные между собой с помощью ссылок (Links). Затем мы поговорим о технологии обработки данных mapreduce, с которой часто будем сталкиваться на страницах этой книги. Мы увидим, как организуется кластер серверов Riak и как он обрабатывает запросы даже в случае отказа отдельных серверов. Наконец, мы расскажем о том, как Riak разрешает конфликты, возникающие при записи на распределенные серверы, и познакомимся с некоторыми расширениями базового сервера.

¹ <http://allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>

3.2. День 1: CRUD, ссылки и типы MIME

Можно скачать и установить сборку Riak, предлагаемую компанией Basho², которая финансирует разработку, но мы предпочитаем собирать продукт сами, так как в исходном дистрибутиве имеются примеры некоторых готовых конфигураций. Если вам совсем не хочется заниматься сборкой, то можете установить готовую версию, а потом скачать исходный код и найти в нем примеры настройки сервера. Для запуска Riak понадобится также язык Erlang³ (версии не ниже R14B03).

Для сборки Riak из исходного кода необходимы три вещи: Erlang (он понадобится также при изучении CouchDB в главе 6), сам исходный код и стандартные средства сборки в Unix типа Make. Процедура установки Erlang несложна, хотя и занимает некоторое время. Исходный код мы брали из репозитория (ссылка имеется на сайте Basho); если на вашем компьютере не установлена клиентская часть системы Git или Mercurial, то можете скачать заархивированный пакет. Все примеры в этой главе тестировались для версии 1.0.2.

Создатели Riak выступили в роли Санта-Клауса, положив в чулок прелестную игрушку для новых пользователей. В том же каталоге, где собирался Riak, выполните такую команду:

```
$ make devrel
```

По ее завершении вы обнаружите три сконфигурированных сервера. Осталось только запустить их:

```
$ dev/dev1/bin/riak start
$ dev/dev2/bin/riak start
$ dev/dev3/bin/riak start
```

Если какой-нибудь сервер не запустится, потому что порт уже занят, не паникуйте. Просто измените номер порта, для чего откройте файл `etc/app.config` в каталоге неудачливого сервера и пропишите новый порт в строке вида:

```
{http, [ {"127.0.0.1", 8091 } ]}
```

Теперь у нас должно быть три процесса Erlang, в которых исполняется программа `beam.smp`. Каждый процесс представляет один узел Riak (экземпляр сервера), который ничего не знает о существовании

2 <http://www.basho.com/>

3 <http://www.erlang.org/>

других узлов. Чтобы создать кластер, узлы необходимо объединить, запустив утилиту `riak-admin` в каталоге каждого сервера и указав в команде `join` адрес любого другого узла.

```
$ dev/dev2/bin/riak-admin join dev1@127.0.0.1
```

Какой именно сервер указать, неважно — в Riak все узлы равноправны. Включив в кластер узлы `dev1` и `dev2`, мы можем указать на любой из них из `dev3`.

```
$ dev/dev3/bin/riak-admin join dev2@127.0.0.1
```

Убедимся, что все серверы работают, проверив их состояние в браузере: `http://localhost:8091/stats`. Эта команда может предложить скачать файл, который содержит разнообразную информацию о кластере. Выглядит она примерно так (файл отредактирован для удобства чтения):

```
{
  "vnode_gets":0,
  "vnode_puts":0,
  "vnode_index_reads":0,
  ...
  "connected_nodes":[
    "dev2@127.0.0.1",
    "dev3@127.0.0.1"
  ],
  ...
  "ring_members":[
    "dev1@127.0.0.1",
    "dev2@127.0.0.1",
    "dev3@127.0.0.1"
  ],
  "ring_num_partitions":64,
  "ring_ownership":
    "[{'dev3@127.0.0.1',21},{ 'dev2@127.0.0.1',21},{ 'dev1@127.0.0.1',22}]",
  ...
}
```

Убедиться, что все серверы — равноправные участники кольца (*ring*) можно, прозвонив порты других серверов: 8092 (`dev2`) и 8093 (`dev3`). А пока ограничимся статистикой, полученной от `dev1`.

Взгляните на свойство *ring_members* — оно содержит имена всех узлов и одинаково для любого сервера. Далее, в свойстве *connected_nodes* должен быть список других серверов в кольце.

Теперь остановите узел...

```
$ dev/dev2/bin/riak stop
```

... и снова перейдите по адресу `/stats`. Как видите, адрес `dev2@127.0.0.1` больше не присутствует в списке `connected_nodes`. Запустите `dev2`, и он самостоятельно присоединится к *кольцу Riak* (о том, что такое кольцо, мы поговорим на второй день).

Лучше REST может быть только REST (или как завивать локоны)⁴

Акроним REST означает REpresentational State Transfer (передача представимых состояний). Несмотря на жаргонное звучание, этот архитектурный стиль де-факто стал основой многих веб-приложений, поэтому его название стоит запомнить. REST – это рекомендация по отображению ресурсов на URL-адреса и взаимодействию с ресурсами путем использования глаголов, соответствующих операциям CRUD: POST (создание), GET (чтение), PUT (обновление) и DELETE (удаление).

Установите клиентскую программу `cURL` для работы по протоколу HTTP, если она еще не установлена. Мы будем использовать ее для отправки REST-запросов, так как она позволяет легко задавать глаголы (например, GET или PUT) и HTTP-заголовки (например, `Content-Type`). С помощью команды `curl` мы можем напрямую обращаться к REST-интерфейсу Riak, не пользуясь интерактивной консолью и драйвером какого-нибудь языка, например Ruby.

Убедиться, что `curl` работает, можно, прозвонив узел:

```
$ curl http://localhost:8091/ping
OK
```

Теперь отправим заведомо неправильный запрос. Флаг `-i` говорит `cURL`, что мы хотим видеть только заголовки ответа.

```
$ curl -I http://localhost:8091/riak/no_bucket/no_key
```

```
HTTP/1.1 404 Object Not Found
Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic)
Date: Thu, 04 Aug 2011 01:25:49 GMT
Content-Type: text/plain
Content-Length: 10
```

Так как работа Riak основана на URL-адресах и действиях, то используются заголовки и коды ошибок, определенные в протоколе HTTP. Ответ с кодом 404 означает, что страница не найдена – не на что смотреть. А, стало быть, пора что-то поместить в хранилище Riak, воспользовавшись глаголом PUT.

⁴ Игра слов: `curl` – по-английски «локон». Прим. перев.

Параметр `-X PUT` говорит сURL, что мы хотим выполнить HTTP-действие PUT, чтобы сохранить явно заданное значение ключа. Флаг `-H` означает, что следующая за ним строка должна быть передана в виде HTTP-заголовка. В данном случае мы устанавливаем MIME-тип содержимого HTML. Строка, следующая после флага `-d` (тело запроса), интерпретируется Riak как новое значение.

```
$ curl -v -X PUT http://localhost:8091/riak/favs/db \
-H "Content-Type: text/html" \
-d "<html><body><h1>My new favorite DB is RIAK</h1></body></html>"
```

Теперь, перейдя в браузере по адресу `http://localhost:8091/riak/favs/db`, вы увидите симпатичное сообщение от самого себя.

Сохранение нового значения с помощью глагола PUT

Riak – это хранилище ключей и значений, и, значит, чтобы получить значение, нужно задать ключ. Riak разбивает все множество ключей на *сегменты* (bucket), чтобы избежать коллизий. Например, ключ для *языка* java может сосуществовать с ключом для *напитка* java⁵.

Мы хотим создать систему для учета животных в гостинице для собак. Сначала создадим сегмент `animals`, в котором будут храниться сведения о лохматых постояльцах. URL-адрес устроен следующим образом:

```
http://SERVER:PORT/riak/BUCKET/KEY
```

Проще всего поместить данные в сегмент Riak, если ключ заранее известен. Первым мы добавим чудесного пса Тузика (*Ace*, *The Wonder Dog*), сопоставив ему ключ `ace` и значение `{"nickname" : "The Wonder Dog", "breed" : "German Shepherd"}`. Создавать сегмент явно необязательно – помещение первого же значения в новый сегмент приводит к его созданию.

```
$ curl -v -X PUT http://localhost:8091/riak/animals/ace \
-H "Content-Type: application/json" \
-d '{"nickname" : "The Wonder Dog", "breed" : "German Shepherd"}'
```

В ответ мы получаем код 204. Флаг `-v` (от слова *verbose* – подробно) приводит к выводу следующей строки с заголовком:

```
< HTTP/1.1 204 No Content
```

⁵ Словом java в Америке называют не только яванский кофе, но и кофе вообще. *Прим. перев.*

Теперь можно просмотреть список созданных нами сегментов.

```
$ curl -X GET http://localhost
{"buckets":["favs","animals"]}
```

При желании получить результат операции установки, добавив параметр `?returnbody=true`. Проверим это, добавив собачку Полли:

```
$ curl -v -X PUT http://localhost:8091/riak/animals/polly?returnbody=true \
-H "Content-Type: application/json" \
-d '{"nickname": "Sweet Polly Purebred", "breed": "Purebred"}'
```

На это раз мы получим ответ с кодом 200.

```
< HTTP/1.1 200 OK
```

Если нам все равно, как будет называться ключ, то Riak сгенерирует его самостоятельно, нужно только отправить POST-запрос.

```
$ curl -i -X POST http://localhost:8091/riak/animals \
-H "Content-Type: application/json" \
-d '{"nickname": "Sergeant Stubby", "breed": "Terrier"}'
```

Сгенерированный ключ будет возвращен в заголовке `Location`; обратите также внимание, что код ответа в этом случае равен 201.

```
HTTP/1.1 201 Created
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic)
Location: /riak/animals/6VZc2o7zKxq2B34kJrm1S0ma3PO
Date: Tue, 05 Apr 2011 07:45:33 GMT
Content-Type: application/json
Content-Length: 0
```

GET-запрос (отправляемый сURL по умолчанию, если метод запроса не задан) на указанный в заголовке `Location` адрес возвращает значение ключа.

```
$ curl http://localhost:8091/riak/animals/6VZc2o7zKxq2B34kJrm1S0ma3PO
```

Запрос методом `DELETE` удаляет ключ.

```
$ curl -i -X DELETE http://localhost:8091/riak/animals/6VZc2o7zKxq2B34kJrm1S0ma3PO
```

```
HTTP/1.1 204 No Content
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic)
Date: Mon, 11 Apr 2011 05:08:39 GMT
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
```

В ответ на запрос `DELETE` тело не возвращается, но в случае успешного удаления код ответа будет равен 204, иначе – 404, вполне ожидаемо.

Если мы забыли, какие ключи имеются в сегменте, то можем получить их список:

```
$ curl http://localhost:8091/riak/animals?keys=true
```

Можно также получить их в виде потока, задав параметр `keys=stream`, это удобнее, когда набор данных очень велик, — сервер просто будет посылать порциями объекты, содержащие массивы ключей, а в конце пошлет пустой массив.

Ссылки

Ссылками (link) называются метаданные, ассоциирующие один ключ с другими. Базовая структура имеет следующий вид:

```
Link: </riak/bucket/key>; riaktag=\"whatever\"
```

Ключ, на который указывает эта ссылка, заключен в угловые скобки (<...>), далее следует точка с запятой и тег, описывающий, как ссылка соотносится с данным значением (это может быть произвольная строка).

Следование по ссылкам

В нашей гостинице для собак имеется несколько клеток (просторных, комфортабельных и гуманных). С помощью ссылок мы будем отслеживать, в какой клетке находится каждое животное. Чтобы сказать, что клетка 1 *содержит* (contains) собачку Полли, мы свяжем клетку с ее ключом (попутно будет создан новый сегмент cages). Клетка (cage) находится в номере 101, это значение мы и укажем в виде JSON-данных.

```
$ curl -X PUT http://localhost:8091/riak/cages/1 \
-H "Content-Type: application/json" \
-H "Link: </riak/animals/polly>; riaktag=\"contains\"" \
-d '{"room" : 101}'
```

Отметим, что эта ссылка однонаправленная. Иными словами, только что созданная клетка «знает», что в ней живет Полли, но в запись о самой Полли мы не внесли никаких изменений. Убедиться в этом можно, запросив данные о Полли, — содержимое заголовка Link осталось прежним.

```
$ curl -i http://localhost:8091/riak/animals/polly
```

```
HTTP/1.1 200 OK
```

```
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA=
Vary: Accept-Encoding
```

```
Server: MochiWeb/1.1 WebMachine/1.9.0 (participate in the frantic)
Link: </riak/animals>; rel="up"
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT
ETag: "VD0ZAfOTsIHsgG5PM3YZW"
Date: Tue, 13 Dec 2011 17:54:51 GMT
Content-Type: application/json
Content-Length: 59
```

```
{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}
```

Мы можем включать любое число ссылок, разделив их запятыми. Поместим Тузика в клетку 2 и с помощью тега `next_to` укажем на клетку 1, чтобы знать, что она находится рядом.

```
$ curl -X PUT http://localhost:8091/riak/cages/2 \
-H "Content-Type: application/json" \
-H "Link:</riak/animals/ace>;riaktag=\"contains\",
  </riak/cages/1>;riaktag=\"next_to\"" \
-d '{"room" : 101}'
```

Ссылки в Riak интересны тем, что допускают *следование по ссылкам* (и более мощный механизм связанных запросов `mapreduce`, о котором мы поговорим завтра). Для получения связанных данных мы добавляем в URL *спецификацию ссылки*, устроенную следующим образом: `/_/_/.` Знаки подчеркивания в URL представляют собой метасимволы для каждого из параметров ссылки: сегмент, тег и признак «оставлять». Что эти термины означают, мы объясним чуть позже. А сейчас выберем все ссылки, ведущие из клетки 1.

```
$ curl http://localhost:8091/riak/cages/1/_/_/.
--4PYi9DW8iJK5aCvQQrrP7mh7jZs
Content-Type: multipart/mixed; boundary=Av1fawIA4WjypRlz5gHJtrRqk1D

--Av1fawIA4WjypRlz5gHJtrRqk1D
X-Riak-Vclock: a85hYGBgzGDKBVicypz/fvrde/U5gymRMY+VwZw35gRfFgA=
Location: /riak/animals/polly
Content-Type: application/json
Link: </riak/animals>; rel="up"
ETag: VD0ZAfOTsIHsgG5PM3YZW
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT

{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}
--Av1fawIA4WjypRlz5gHJtrRqk1D--

--4PYi9DW8iJK5aCvQQrrP7mh7jZs--
```

Сервер вернул ответ с содержимым типа `multipart/mixed`, в который входят заголовки и тела, включающие все связанные ключи и значения. Смотреть на это неприятно. Завтра мы покажем более

удобный способ получить связанные ссылками данные, который, кстати, представляет результаты в удобочитаемом виде. А сегодня все же разберемся с этим синтаксисом.

Для тех, кто не знает, как читать данные MIME-типа `multipart/mixed`, поясним, что в заголовке `Content-Type` указана разграничительная строка (`boundary`), которая обозначает начало и конец секции заголовков и тела.

```
--BcOdSWMLuhkisryp0GidDLqeA64
HTTP-заголовки и тело
--BcOdSWMLuhkisryp0GidDLqeA64--
```

В нашем случае данные в теле – это то, на что указывает клетка 1: Polly Purebred. Обратите внимание, что в возвращенных заголовках нет информации о самой ссылке. Это нормально, данные никуда не делись и хранятся под ключом, на который ведет ссылка.

При следовании по ссылкам мы можем заменить подчерки в спецификации ссылки, чтобы оставить только интересующие нас значения. Из клетки 2 ведут две ссылки, поэтому выполнение запроса, указанного в спецификации ссылки, вернет информацию о проживающем в этой клетке Тузике и о находящейся рядом клетке 1. Чтобы оставить только информацию о сегменте `animals`, замените первый подчёрк именем сегмента.

```
$ curl http://localhost:8091/riak/cages/2/animals,_,_
```

А чтобы узнать о клетках рядом (*next to*) с данной, задайте параметр `tag`.

```
$ curl http://localhost:8091/riak/cages/2/_ ,next_to, _
```

Последний подчёрк – `keep` (оставлять) – принимает значения 1 или 0. Он полезен при следовании по ссылкам второго порядка, то есть ведущим из объекта, на который ведет первая ссылка. Для этого нужно лишь добавить в URL еще одну спецификацию ссылки. Давайте сначала проследуем по ссылке `next_to`, ведущей из клетки 2. Это даст клетку 1. Затем перейдем к животным, на которых ведут ссылки из клетки 1. Поскольку мы задали `keep` равным 0, Riak не станет возвращать промежуточный результат (данные о клетке 1), а вернет только сведения о Полли, которая живет в клетке по соседству с Тузиком.

```
$ curl http://localhost:8091/riak/cages/2/_ ,next_to,0/animals,_,_
--6mBdsboQ8kTt6MlUhg0rgvbLhzd
```



```
Content-Type: multipart/mixed; boundary=EZYdVz9Ox4xzR4jx1I2ugUFFiZh
```

```
--EZYdVz9Ox4xzR4jx1I2ugUFFiZh
```

```
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA=
```

```
Location: /riak/animals/polly
```

```
Content-Type: application/json
```

```
Link: </riak/animals>; rel="up"
```

```
Etag: VD0ZAfOTsIHsgG5PM3YZW
```

```
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT
```

```
{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}
```

```
--EZYdVz9Ox4xzR4jx1I2ugUFFiZh--
```

```
--6mBdsboQ8kTT6MlUHg0rgvbLhzd--
```

Если нас интересует информация как о Полли, так и о клетке 1, то нужно задать кеер равным 1.

```
$ curl http://localhost:8091/riak/cages/2/_ ,next_to,1/_ ,_ ,_
```

```
--PDVOEl7Rh1AP90jGlnlmhz7x8r9
```

```
Content-Type: multipart/mixed; boundary=YliPQ9LPNEoAnDeAMiRkAjCbmed
```

```
--YliPQ9LPNEoAnDeAMiRkAjCbmed
```

```
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRKY+VIYo35gRfFgA=
```

```
Location: /riak/cages/1
```

```
Content-Type: application/json
```

```
Link: </riak/animals/polly>; riaktag="contains", </riak/cages>; rel="up"
```

```
Etag: 6LYhRnMRrGIgsTmPE55PaU
```

```
Last-Modified: Tue, 13 Dec 2011 17:54:34 GMT
```

```
{"room" : 101}
```

```
--YliPQ9LPNEoAnDeAMiRkAjCbmed--
```

```
--PDVOEl7Rh1AP90jGlnlmhz7x8r9
```

```
Content-Type: multipart/mixed; boundary=GS9J6KQLsI8zzMxJluDITfwiUKA
```

```
--GS9J6KQLsI8zzMxJluDITfwiUKA
```

```
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA=
```

```
Location: /riak/animals/polly
```

```
Content-Type: application/json
```

```
Link: </riak/animals>; rel="up"
```

```
Etag: VD0ZAfOTsIHsgG5PM3YZW
```

```
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT
```

```
{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}
```

```
--GS9J6KQLsI8zzMxJluDITfwiUKA--
```

```
--PDVOEl7Rh1AP90jGlnlmhz7x8r9--
```

В ответ сервер вернет все объекты на пути к конечному результату. Иными словами, *оставит* промежуточные данные.

Ссылки – это еще не все

Помимо ссылок, можно сохранять произвольные метаданные, воспользовавшись заголовком с префиксом `X-Riak-Meta-`. Если мы хотим хранить цвет клетки, но не считаем эту информацию необходимой для повседневных задач управления клетками, то можем просто пометить, что клетка 1 розовая (`pink`). Запросив заголовки ответа (флаг `-I`), мы увидим, что сервер вернул имя и значение метаданных.

```
$ curl -X PUT http://localhost:8091/riak/cages/1 \  
-H "Content-Type: application/json" \  
-H "X-Riak-Meta-Color: Pink" \  
-H "Link: </riak/animals/polly>; riaktag=\"contains\" \" \" \  
-d '{"room" : 101}'
```

Типы MIME в Riak

Riak хранит все данные в двоичном виде, как принято в HTTP. Тип MIME сообщает, как интерпретировать двоичные данные; до сих пор мы имели дело только с простым текстом. Типы MIME хранятся на сервере Riak, но в действительности представляют собой всего лишь указание клиенту – чтобы, скачав данные, он знал, что с ними делать.

Мы хотим, чтобы в гостинице хранились фотографии постояльцев. Чтобы загрузить на сервер изображение и задать для него MIME-тип `image/jpeg` нужно всего лишь указать флаг `data-binary` в команде `curl`. И еще мы добавим обратную ссылку на ключ `/animals/polly`, чтобы знать, на кого мы смотрим.

Для начала создайте файл `polly_image.jpg` с изображением и поместите его в тот же каталог, из которого запускается команда `curl`.

```
$ curl -X PUT http://localhost:8091/riak/photos/polly.jpg \  
-H "Content-type: image/jpeg" \  
-H "Link: </riak/animals/polly>; riaktag=\"photo\" \" \" \  
--data-binary @polly_image.jpg
```

Теперь перейдите по этому URL-адресу в браузере и убедитесь, что файл возвращается и отображается именно так, как мы и ожидаем:

```
http://localhost:8091/riak/photos/polly.jpg
```

Поскольку мы указали, что изображение ссылается на `/animals/polly`, то можем проследовать от изображения к Полли, но не в обратном направлении. В отличие от реляционных баз данных, к ссылкам неприменимы правила «содержит» или «является». Ссылка явно

указывает направление перехода. Если вы считаете, что в этой задаче имеет смысл получать изображения, зная животное, то заведите ссылку в соответствующем объекте (или обе сразу).

День 1: итоги

Мы надеемся, что вы оценили потенциал Riak в качестве гибкого механизма хранения данных. Но пока мы рассмотрели лишь стандартные способы работы с ключами и значениями, сбоблив блюдо толикой ссылок. Проектирование схемы Riak находится где-то между системами кэширования и PostgreSQL. Данные разбиваются на несколько логических категорий (сегментов), а значения могут ссылаться друг на друга. Но нормализовать схему столь же строго, как в реляционных базах данных, не имеет смысла, потому что Riak не умеет выполнять соединения для воссоздания данных в исходной форме.

День 1: домашнее задание

Информационный поиск

1. Поставьте закладку на документацию по проекту Riak и найдите документацию по REST API.
2. Найдите максимально полный список типов MIME, поддерживаемых браузерами.
3. Изучите пример конфигурационного файла Riak `dev/dev1/etc/app.config` и сравните его с другими конфигурационными файлами в каталоге `dev`.

Задачи

1. Используя глагол PUT, измените запись `animals/polly`, добавив ссылку, указывающую на изображение `photos/polly.jpg`.
2. Методом POST отправьте файл с еще не рассматривавшимся типом MIME (например, `application/pdf`), найдите сгенерированный ключ и перейдите по соответствующему URL-адресу в браузере.
3. Создайте новый сегмент *medicines* (лекарства), методом PUT загрузите изображение в формате JPEG (указав правильный тип MIME), задав для него ключ *antibiotics*, и свяжите ссылкой с Тузиком (Ace) (бедный, больной щеночек).

3.3. День 2: mapreduce и кластеры серверов

Сегодняшний день мы посвятим каркасу mapreduce и выполним более сложные запросы, чем позволяет традиционная парадигма ключей и значений. Затем мы пойдем еще дальше, объединив mapreduce со следованием по ссылкам. И напоследок рассмотрим архитектуру Riak и поговорим о том, как в ней используются кластерные решения для обеспечения гибкой согласованности и доступности даже в условиях потери связности сети.

Скрипт для загрузки данных

В этом разделе нам понадобятся дополнительные данные. Для этого мы переключимся на другой тип гостиницы – для людей, а не для животных. Простенький скрипт на языке Ruby заполнит базу данными о гигантском отеле на 10000 номеров. Для тех, кто не знает, поясним, что Ruby – популярный универсальный язык программирования. Он очень полезен, когда нужно быстро написать простой и понятный скрипт. Почитать о Ruby можно в книге Дэйва Томаса и Энди Ханта «Programming Ruby: The Pragmatic Programmer's Guide» [TH01], а также в Сети⁶.

Еще нам понадобится менеджер Ruby-пакетов, который называется RubyGems⁷. Вслед за Ruby и RubyGems установите драйвер Riak⁸. Может также понадобится драйвер json, так что устанавливайте сразу оба пакета.

```
$ gem install riak-client json
```

Номера в нашем отеле имеют разную вместимость – от одноместных до восьмиместных – и разные типы – от односпального до апартаментов. То и другое скрипт генерирует случайным образом.

riak/hotel.rb

```
# генерировать тысячи номеров случайного типа и вместимости
require 'rubygems'
require 'riak'
STYLES = %w{single double queen king suite}
```

```
client = Riak::Client.new(:http_port => 8091)
```

6 <http://ruby-lang.org>

7 <http://rubygems.org>

8 <http://rubygems.org/gems/riak-client>

```
bucket = client.bucket('rooms')
# Создать здание в 100 этажей
for floor in 1..100
  current_rooms_block = floor * 100
  puts "Создаю номера #{current_rooms_block} - #{current_rooms_block + 100}"
  # На каждом этаже будет 100 номеров (немаленький отель!)
  for room in 1...100
    # Уникальный номер комнаты используется в качестве ключа
    ro = Riak::RObject.new(bucket, (current_rooms_block + room))
    # Выбрать случайный тип и вместимость номера
    style = STYLES[rand(STYLES.length)]
    capacity = rand(8) + 1
    # Сохранить информацию о номере в формате JSON
    ro.content_type = "application/json"
    ro.data = {'style' => style, 'capacity' => capacity}
    ro.store
  end
end

$ ruby hotel.rb
```

Итак, мы заселили отель, к которому собираемся применить `mapreduce`.

Введение в Mapreduce

Один из самых крупных и долговечных вкладов Google в информатику – популяризация алгоритмического каркаса `mapreduce` для параллельного выполнения задач на нескольких узлах. Он описан в основополагающей статье Google⁹ и стал ценным инструментом для исполнения запросов в целом классе хранилищ данных, устойчивых к потере связности.

При использовании `mapreduce` задача разбивается на две части. На первом шаге – распределения – список данных преобразуется в новый список посредством функции `map()`. На втором шаге – редукции – полученный список преобразуется в одно или несколько скалярных значений с помощью функции `reduce()`. Следование этому принципу позволяет системе разбить задачу на меньшие подзадачи и параллельно исполнять их на массивном кластере серверов. Чтобы подсчитать, сколько значений в базе Riak содержат строку `{country : 'CA'}`, мы могли бы сопоставить каждому подходящему документу строку `{count : 1}` и на этапе редукции вычислить сумму таких единичных счетчиков.

Если бы в нашем наборе данных было 5012 постояльцев из Канады, то результатом редукции был бы документ `{count : 5012}`.

⁹ <http://research.google.com/archive/mapreduce.html>

```
map = function(v) {
  var parsedData = JSON.parse(v.values[0].data);
  if(parsedData.country === 'CA')
    return [{count : 1}];
  else
    return [{count : 0}];
}

reduce = function(mappedVals) {
  var sums = {count : 0};
  for (var i in mappedVals) {
    sums[count] += mappedVals[i][count];
  }
  return [sums];
}
```

В каком-то смысле mapreduce – прямая противоположность привычному способу выполнения запросов. В системе Ruby on Rails данные можно было бы получить следующим способом (с помощью встроенного в нее ORM-интерфейса ActiveRecord):

```
# Конструируем хеш для хранения вместимости номеров, взяв в качестве
# ключа тип номера
capacity_by_style = {}
rooms = Room.all
for room in rooms
  total_count = capacity_by_style[room.style]
  capacity_by_style[room.style] = total_count.to_i + room.capacity
end
```

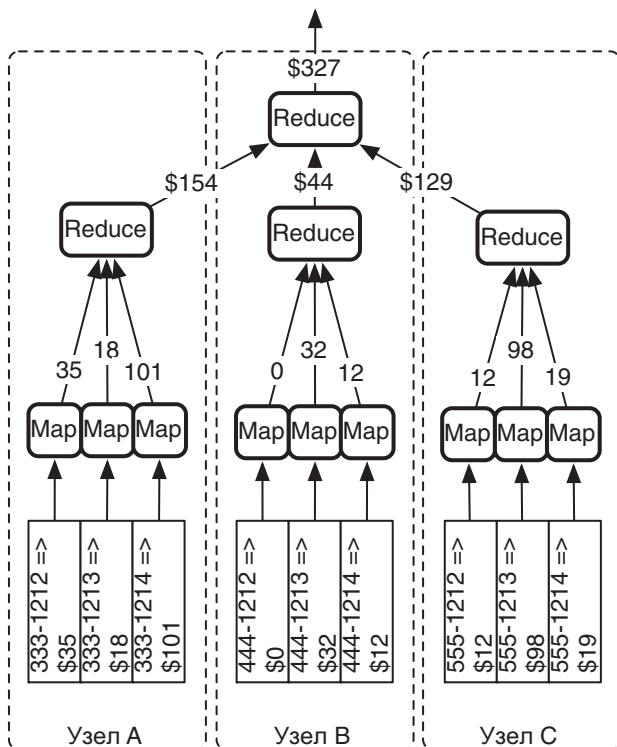
Метод `Room.all` отправляет базе данных SQL-запрос такого вида:

```
SELECT * FROM rooms;
```

База данных посылает все результаты серверу приложений, а тот производит с ними какие-то действия. В данном случае мы перебираем все номера отеля и подсчитываем суммарную вместимость номеров каждого типа (например, все апартаменты могут быть рассчитаны на 448 гостей). Для небольших наборов данных такой подход приемлем. Но с ростом числа номеров система начинает работать медленно, потому что база по-прежнему передает приложению сведения обо всех номерах.

Mapreduce делает всё наоборот. Вместо того чтобы получать данные от базы и обрабатывать их на стороне клиента (то есть сервера приложений), каркас mapreduce отправляет на все узлы, где работают серверы базы данных, алгоритм, который те должны выполнить и вернуть результат. *Каждому серверному объекту «распределяется»*

группа данных с общим ключом, а сервер «редуцирует» полученные данные в одно значение.



Функция *map* поставляет исходные данные редукторам, а вычисленные теми данные передаются редукторам следующего уровня

Рис. 7. Результаты функции *map*

В случае Riak это означает, что серверы базы данных отвечают за распределение и редукцию значений в каждом узле. Редуцированные значения передаются на следующий круг, где какой-то другой сервер (обычно отправивший запрос) редуцирует их дальше – до тех пор, пока не будет получен конечный результат, который передается запрашившему клиенту (или, например, серверу приложений на платформе Rails).

Такая инверсия потока выполнения – действенный способ исполнять сложные алгоритмы локально на каждом сервере и возвра-

щать вызывающему клиенту только небольшой по размеру результат. *Быстрее отправить алгоритм к данным, а затем данные к алгоритму.* На рис. 7 показано, как можно вычислить общую сумму телефонных счетов на трех серверах, каждому из которых распределяются номера с общим префиксом.

Результаты функций map поступают на вход функциям reduce первого уровня, а функциям reduce следующего уровня на вход подаются результаты map и функций reduce предыдущего уровня. Мы еще вернемся к этой идее в последующих главах, потому что это важный, хотя и непростой, аспект искусства написания эффективных mapreduce-запросов.

Mapreduce в Riak

Сейчас мы напишем функции mapreduce для набора данных Riak, которые будут работать, как рассмотренный выше подсчет суммарной вместимости номеров. У реализации mapreduce в Riak есть удобная особенность – функцию map() можно запускать автономно и смотреть, какие получаются промежуточные результаты (в предположении, что редукция вообще нужна). Не будем спешить и посмотрим только на результаты для номеров 101, 102 и 103.

Для настройки функции map необходимо задать язык программирования и исходный код; мы напишем функцию на JavaScript (код функции – это просто строка, поэтому необходимо правильно экранировать специальные символы).

Команда @- в cURL позволяет не закрывать стандартный ввод, пока не будет нажата клавиша CTRL+D. Введенные данные попадут в тело HTTP-запроса, который затем отправляется методом POST на адрес /mapred (обратите внимание – именно /mapred, а не /riak/mapred).

```
$ curl -X POST -H "content-type:application/json" \
http://localhost:8091/mapred --data @-
{
  "inputs":[
    ["rooms","101"],["rooms","102"],["rooms","103"]
  ],
  "query":[
    { "map":{
      "language":"javascript",
      "source":
        "function(v) {
          /* Достать данные из объекта Riak и разобрать их как JSON */
          var parsed_data = JSON.parse(v.values[0].data);
```



```

var data = {};
/* Словарь вмести́мостей с ключом "тип комнаты" */
data[parsed_data.style] = parsed_data.capacity;
return [data];
}
}
}
}
}

```

CTRL-D

По адресу `/mapred` сервер ожидает получить корректный JSON-документ, именно в нем мы определяем вид команд `mapreduce`. Мы задали три интересующих нас номера, присвоив элементу `inputs` массив, содержащий пары `[сегмент, ключ]`. Но самое главное – это элемент `query`, который должен содержать массив JSON-объектов, индексированных ключами `map`, `reduce` и `link` (о последнем мы поговорим подробнее ниже).

Наша функция `map` получает данные (`v.values[0].data`), разбирает строку как JSON-объект (`JSON.parse(...)`) и возвращает ассоциативный массив, в котором ключом является тип номера (`parsed_data.style`), а значением – вместимость (`parsed_data.capacity`). Результат будет получен в виде:

```
[{"suite":6}, {"single":1}, {"double":1}]
```

Это всего лишь данные, извлеченные из трех JSON-объектов, описывающих номера 101, 102 и 103. Но нам нужно не просто вывести данные в формате JSON. Значение ключа можно было бы преобразовать во что угодно. Мы проанализировали только тело запроса, но могли бы получить также метаданные, информацию о ссылках, ключ или данные. Возможно всё – лишь бы значению каждого ключа ставилось в соответствие какое-то другое значение.

Если хотите, можете распределить все 10000 номеров, заменив в параметре `inputs` массивы `[сегмент, ключ]` именем сегмента `rooms`:

```
"inputs": "rooms"
```

Предостережение: в ответ будет возвращено много данных. Наконец, стоит упомянуть, что, начиная с версии Riak 1.0, функции `mapreduce` обрабатываются подсистемой Riak Pipe. В более ранних версиях используется устаревшая подсистема `mapred_system`. На конечном пользователе это не отражается, но быстродействие и стабильность резко повысились.

Хранимые функции

Riak предоставляет также возможность хранить функцию map в качестве значения сегмента. Это еще один пример перемещения алгоритма в базу данных. По существу, это хранимая процедура, точнее, определенная пользователем функция – идея, близкая к той, что используется в реляционных базах данных уже много лет.

```
$ curl -X PUT -H "content-type:application/json" \
http://localhost:8091/riak/my_functions/map_capacity --data @-
function(v) {
  var parsed_data = JSON.parse(v.values[0].data);
  var data = {};
  data[parsed_data.style] = parsed_data.capacity;
  return [data];
}
```

Надежно сохранив свою функцию, мы можем выполнить ее, указав сегмент и ключ, где она находится.

```
$ curl -X POST -H "content-type:application/json" \
http://localhost:8091/mapred --data @-
{
  "inputs":[
    ["rooms","101"],["rooms","102"],["rooms","103"]
  ],
  "query":[
    { "map":{
      "language":"javascript",
      "bucket":"my_functions",
      "key":"map_capacity"
    }}
  ]
}
```

Результаты должны быть такими же, как при передаче JavaScript-функции прямо в запросе.

Встроенные функции

К вашим услугам ряд встроенных в Riak функций, присоединенных к JavaScript-объекту Riak. Если выполнить показанный ниже код, то сервер извлечет из объектов, описывающих номера, значения и вернет их в формате JSON. Именно это делает функция Riak.mapValuesJson.

```
curl -X POST http://localhost:8091/mapred \
-H "content-type:application/json" --data @-
{
  "inputs":[
```

```
[
  ["rooms", "101"], ["rooms", "102"], ["rooms", "103"]
],
"query": [
  { "map": {
    "language": "javascript",
    "name": "Riak.mapValuesJson"
  }}
]
}
```

Riak содержит и другие функции, все они находятся в файле `mapred_builtins.js`, который можно найти в Сети (или порывшись в коде). С помощью такого же синтаксиса можно вызывать и ваши собственные встроенные функции, но об этом мы будем говорить завтра.

Редукция

Распределение полезно и само по себе, но позволяет только преобразовывать одни значения в другие. Для анализа набора данных, даже такого простого, как подсчет количества записей, необходим еще один шаг. Тут-то и вступает в игру редукция.

В рассмотренном выше примере на SQL/Ruby мы видели, как можно перебрать все значения и подсчитать суммарную вместимость номеров каждого типа. Теперь сделаем то же самое в функции `reduce`, написанной на JavaScript.

По большей части команды, отправляемые на адрес `/mapred`, те же самые. Только теперь мы добавим еще функцию `reduce`.

```
$ curl -X POST -H "content-type:application/json" \
http://localhost:8091/mapred --data @-
{
  "inputs": "rooms",
  "query": [
    { "map": {
      "language": "javascript",
      "bucket": "my_functions",
      "key": "map_capacity"
    }},
    { "reduce": {
      "language": "javascript",
      "source":
        "function(v) {
          var totals = {};
          for (var i in v) {
            for(var style in v[i]) {
              if( totals[style] ) totals[style] += v[i][style];
              else
                totals[style] = v[i][style];
            }
          }
        }
      "
    }
  ]
}
```

```

    }
  }
  return [totals];
}"
  }}
]
}

```

Выполнив этот запрос для всех номеров, мы получим ассоциативный массив, в котором ключом является тип номера, а значением – суммарная вместимость номеров такого типа.

```
[{"single":7025,"queen":7123,"double":6855,"king":6733,"suite":7332}]
```

На вашей машине получатся другие результаты, потому что данные о номерах генерировались случайно.

Фильтры ключей

Сравнительно недавно в Riak было добавлено понятие фильтра ключей. Это набор команд, которые применяются для обработки каждого ключа перед подачей его на вход mapreduce. Фильтры позволяют сэкономить на загрузке ненужных данных. В следующем примере мы преобразуем ключ – номер комнаты – в целое число и проверяем, что оно меньше 1000 (то есть номер находится на одном из нижних десяти этажей; все номера, расположенные выше, игнорируются).

Паттерны редукторов

Функцию `reduce` проще написать, если следовать тому же паттерну, что при написании функции `map`. Иначе говоря, если одиночное значение отображается следующим образом:

```
[{name:'Eric', count:1}]
```

то результат редукции должен выглядеть так:

```
[{name:'Eric', count:105}, {name:'Jim', count:215}, ...]
```

Разумеется, это не обязательно – но практически целесообразно. Поскольку выход одних редукторов может подаваться на вход других, то вы не можете знать, откуда поступили входные значения для конкретной функции `reduce`: от `map`, от `reduce` или из обоих источников. Но если следовать описанному выше паттерну, то это и не важно – никакой разницы нет! В противном случае функции `reduce` пришлось бы каждый раз проверять тип полученных данных и принимать те или иные решения.

В примере, возвращающем суммарную вместимость номеров, замените `"inputs": "rooms"` следующим блоком (он должен заканчиваться запятой):

```

"inputs":{
  "bucket":"rooms",
  "key_filters":[["string_to_int"], ["less_than", 1000]]
},

```

Следует обратить внимание на две вещи: во-первых, запрос исполняется гораздо быстрее (так как мы обрабатывали лишь те значения, которые нужны), а, во-вторых, суммарные величины получились меньше (так как учтены только десять нижних этажей).

Каркас `mapreduce` – мощное средство агрегирования данных и выполнения их всестороннего анализа. Мы будем часто возвращаться к нему в этой книге, но базовая идея во всех случаях одна и та же. Riak, впрочем, добавляет к `mapreduce` одну дополнительную черту – ссылки.

Следование по ссылкам с помощью `mapreduce`

Вчера мы познакомились с механизмом следования по ссылкам. А сегодня посмотрим, как это делается с помощью `mapreduce`. В секции *query* наряду с параметрами `map` и `reduce` появляется еще один – `link`.

Вернемся к сегменту `cages` из вчерашнего примера с гостиницей для собак и напишем распределитель, который возвращает только клетку 2 (напомним, что в ней обитает Тузик).

```

$ curl -X POST -H "content-type:application/json" \
http://localhost:8091/mapred --data @-
{
  "inputs":{
    "bucket":"cages",
    "key_filters":[["eq", "2"]]
  },
  "query":{
    {"link":{
      "bucket":"animals",
      "keep":false
    }},
    {"map":{
      "language":"javascript",
      "source":
        "function(v) { return [v]; }"
    }}
  ]
}

```

Хотя `mapreduce`-запрос относится к сегменту `cages`, возвращается также информация о собаке Тузике, потому что на нее ведет ссылка из клетки 2.

```
[{
  "bucket": "animals",
  "key": "ace",
  "vclock": "a85hYGBgzmdKBVIsrDJPfTKYEhnzWBn6LfIP80GFWVZay0KF5yGE2ZqTGPmCLiJLZAEA",
  "values": [{
    "metadata": {
      "Links": [],
      "X-Riak-VTag": "4JV1DcEYRIKuyUhw8OUYJS",
      "content-type": "application/json",
      "X-Riak-Last-Modified": "Tue, 05 Apr 2011 06:54:22 GMT",
      "X-Riak-Meta": []},
    "data": "{ \"nickname\" : \"The Wonder Dog\",
      \"breed\" : \"German Shepherd\"}"
  ]
}]
```

В массиве `values` присутствуют данные и метаданные (которые обычно возвращаются в виде заголовков HTTP).

Если объединить функции `map` и `reduce`, следование по ссылкам и фильтры ключей, то можно будет исполнять произвольные запросы к широкому спектру ключей Riak. Это гораздо эффективнее, чем просматривать все данные на стороне клиента. Поскольку такие запросы обычно исполняются одновременно на нескольких серверах, то долго ждать не придется. Но если нетерпимо даже небольшое ожидание, то в запросе можно указать еще один параметр: `timeout`. Таймаут задается в миллисекундах (по умолчанию подразумевается `"timeout": 60000`, то есть 60 секунд); если выполнение запроса не успевает завершиться за указанное время, то запрос снимается.

О согласованности и долговечности

Серверная архитектура Riak устраняет точки общего отказа (все узлы равноправны) и позволяет по желанию наращивать или сокращать кластер. Это важно при реализации крупномасштабных проектов, так как база данных остается доступной, даже когда несколько узлов выходят из строя или по какой-то причине перестают отвечать.

Но распределению данных по нескольким серверам неизбежно сопутствует одна и та же проблема. Если база данных должна функционировать в условиях потери связности сети (когда некоторые сообщения пропадают), то необходимо идти на компромисс. Либо оставить базу данных *доступной* для запросов к серверу, либо принять меры по гарантированию *согласованности* данных. Невозможно создать распределенную базу данных, которая была бы полностью согласованной, доступной и устойчивой к потере связности. Можно обеспечить

выполнение только двух условий (устойчива к потере связности и согласована, устойчива к потере связности и доступна, согласована, доступна, но не является распределенной). Это утверждение известно под названием теоремы CAP (Consistency – согласованность, Availability – доступность, Partition tolerance – устойчивость к потере связности). Дополнительные сведения см. в приложении 2, но уже сейчас скажем, что это проблема проектирования системы.

Однако в теореме есть одна лазейка. На самом деле утверждается, что *в любой момент времени* нельзя обеспечить сразу согласованность, доступность и устойчивость к потере связности. Riak обращает этот факт себе на пользу, позволяя выбирать доступность или согласованность на уровне отдельных запросов. Сначала посмотрим, как в Riak организуется кластер серверов, а потом научимся настраивать операции чтения и записи в кластере.

Кольцо Riak

Riak разбивает множество серверов на секции, обозначаемые 160-разрядным числом (то есть от 0 до 2^{160}). Разработчики Riak любят представлять это огромное целое число в виде круга, который называют *кольцом*. Когда вычисляется хеш ключа, отображающий его на секцию, кольцо определяет, на каком сервере Riak будет храниться значение.

При настройке кластера Riak нужно первым делом решить, сколько в нем будет секций. Предположим, что имеется 64 секции (это значение подразумевается по умолчанию). Если разнести эти 64 секции по трем узлам (то есть серверам), то Riak выделит каждому узлу 21 или 22 секции ($64 / 3$). Каждая секция называется виртуальным узлом, или *v-узлом* (vnode). На этапе запуска все серверы Riak обходят кольцо, по очереди предъявляя заявки на секции до тех пор, пока не будут исчерпаны все v-узлы. Это показано на рис. 8.

Узел А управляет v-узлами 1, 4, 7, 10...63. Эти v-узлы отображаются на секции в 160-разрядном кольце. Опросив состояние трех серверов разработки (вспомните команду `curl -H "Accept: text/plain" http://localhost:8091/stats`, которую мы изучали вчера), вы увидите такой результат:

```
"ring_ownership": \
"[{'dev3@127.0.0.1',21},{ 'dev2@127.0.0.1',21},{ 'dev1@127.0.0.1',22}]"
```

Второе число в каждом объекте – это количество v-узлов, которыми владеет узел. В сумме получается 64 ($21 + 21 + 22$).

Каждый v-узел представляет диапазон хешированных ключей. При вставке данных о номере с ключом *101* хеш-значение ключа может попасть в диапазон v-узла 2, и тогда объект ключ-значение будет храниться в узле В. Достоинство такого решения в том, что когда нужно узнать, на каком сервере находится ключ, Riak просто вычисляет его хеш-значение и находит соответствующий v-узел. Точнее, Riak преобразует хеш-значение в список потенциальных v-узлов и выбирает из него первый элемент.

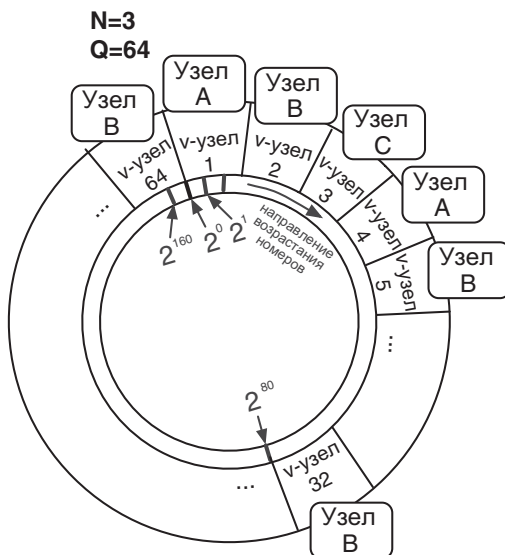


Рис. 8. Кольцо Riak с 64 v-узлами, распределенными по трем узлам

Узлы и операции чтения-записи

Riak позволяет управлять операциями чтения-записи в кластер с помощью трех параметров: N , W и R . N – это количество узлов, на которые в конечном счете будет реплицирована операция записи, – иначе говоря, количество копий данных в кластере. W – количество узлов, на которые данные должны быть фактически записаны перед отправкой ответа об успешном завершении операции. Если W меньше N , то запись будет считаться успешной, хотя Riak все еще продолжает реплицировать данные. Наконец, R – количество узлов, необходимое для успешного чтения значения. Если R больше количества доступных копий, то запрос на чтение завершится неудачно.

Рассмотрим эти параметры более подробно.

При записи объекта в базу данных Riak мы можем затребовать репликацию на несколько узлов. Плюс в том, что если один сервер выйдет из строя, то останется копия данных на другом сервере. В свойстве сегмента `n_val` хранится количество узлов, на которые следует реплицировать значение (величина N); по умолчанию оно равно 3. Изменить свойства узла можно, записав новое значение в объект `props`. Вот как установить `n_val` в сегменте `animals` равным 4:

```
$ curl -X PUT http://localhost:8091/riak/animals \
-H "Content-Type: application/json" \
-d '{"props":{"n_val":4}}'
```

N – это количество узлов, которые будут содержать правильное значение *в конечном счете*. Это не означает, что мы должны дожидаться, пока значение будет реплицировано на *все* узлы, и только потом вернуть управление. Иногда требуется вернуть результат клиенту немедленно и разрешить Riak продолжить репликацию в фоновом режиме. А иногда, наоборот, желательно дожидаться завершения репликации на все N узлов (безопасности ради).

Величина W определяет, на скольких узлах запись должна завершиться успешно, прежде чем можно будет считать успешной операцию в целом. В конечном счете значение будет записано на все четыре узла, но если сделать W равным 2, то операция записи вернет управление после успешной записи всего двух копий. Репликация на оставшиеся два узла произойдет в фоновом режиме.

```
curl -X PUT http://localhost:8091/riak/animals \
-H "Content-Type: application/json" \
-d '{"props":{"w":2}}'
```

Наконец, R – это количество узлов, откуда должно быть прочитано значение, чтобы операция чтения считалась успешной. Можно задать величину R по умолчанию, как мы выше поступили с `n_val` и `w`.

```
curl -X PUT http://localhost:8091/riak/animals \
-H "Content-Type: application/json" \
-d '{"props":{"r":3}}'
```

Но Riak предлагает и более гибкое решение. Мы можем указывать, со скольких узлов следует прочесть значение, задавая параметр `r` в конкретном запросе.

```
curl http://localhost:8091/riak/animals/ace?r=3
```

Возможно, вам непонятно, зачем вообще читать данные с нескольких узлов. Ведь записанное значение в конечном счете должно быть

реплицировано на N узлов, и читать можно с любого из них. Ответ, пожалуй, будет проще продемонстрировать на картинке.

Предположим, что тройка NRW задана следующим образом $\{“n_val”:3, “r”:2, “w”:1\}$, как показано на рис. 9. В этом случае система будет быстрее отзываться на операции записи, поскольку перед возвратом управления необходимо успешно завершить операцию только на одном узле. Однако не исключено, что какой-то другой процесс начнет операцию чтения до того, как все узлы синхронизируются. Даже при чтении с двух узлов есть шанс получить старое значение.

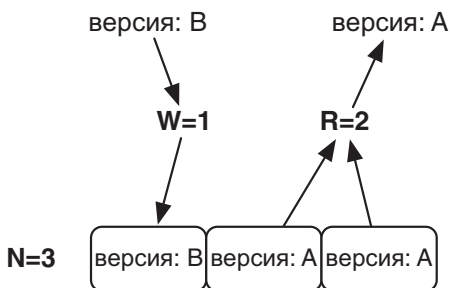


Рис. 9. Согласованность в конечном счете: $W+R \leq N$

Один из способов гарантировать получение актуального значения – установить $W=N$ и $R=1$: $\{“n_val”:3, “r”:1, “w”:3\}$ (см. рис. 10). То есть поступить точно так же, как делают реляционные СУБД, которые обеспечивают непротиворечивость, не возвращая управление, пока запись не будет завершена. Это, конечно, ускорит чтение, так как нам нужно обратиться только к одному узлу. Зато запись может замедлиться – и заметно.

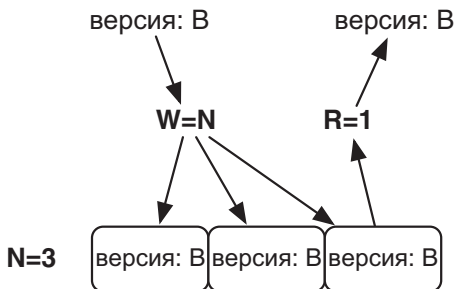


Рис. 10. Согласованность за счет записи: $W=N, R=1$

Можно вместо этого писать на один узел, а читать со всех, то есть задать $W=1, R=N$: $\{“n_val”:3, “r”:3, “w”:1\}$ (см. рис. 11). Хотя не

исключено, что среди них окажется несколько устаревших значений, но гарантируется, что последнее записанное также будет прочитано. Останется только выяснить, что это за значение (применяя алгоритм векторных часов, который мы опишем завтра). Разумеется, при этом возникает прямо противоположная проблема – медленное чтение.

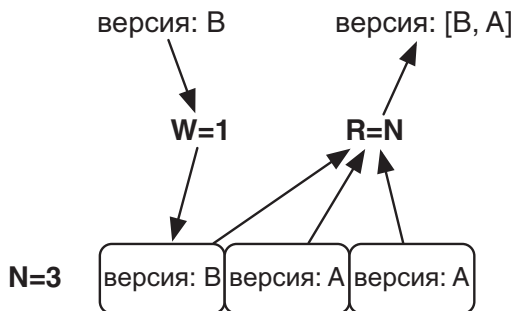


Рис. 11. Согласованность за счет чтения: $W=1, R=N$

Можно также положить $W=2$ и $R=2$: `{"n_val":3, "r":2, "w":2}` (см. рис. 12). В таком случае требуется завершить запись более чем на половине узлов и читать также более чем с половины узлов. При этом мы обеспечиваем согласованность, распределяя временные задержки поровну между операциями чтения и записи. Это называется *кворумом* и дает минимальное число операций, необходимое для поддержания согласованности данных.

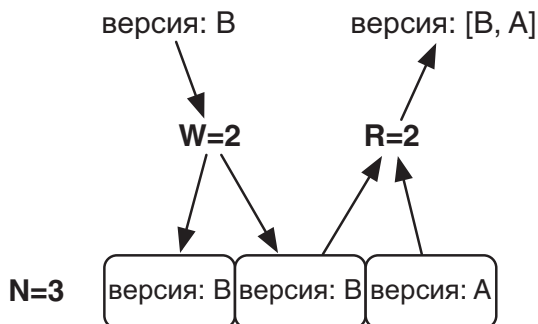


Рис. 12. Согласованность за счет кворума: $W+R>N$

Вы вправе выбрать для R и W любое значение от 1 до N , но в общем случае рекомендуется задавать один узел, все узлы или кворум. Эти настройки настолько типичны, что для R и W предусмотрены специальные представляющие их строковые значения.

Значение	Определение
one	Это просто 1. Присваивание такого значения параметру W или R означает, что операция будет считаться успешной после завершения хотя бы на одном узле.
all	Значение, совпадающее с N. Присваивание такого значения параметру W или R означает, что операция будет считаться успешной после того, как завершится на всех реплицированных узлах.
quorum	Равно $N/2+1$. Означает, что операция должна успешно завершиться на большинстве узлов.
по умолчанию	Значение W или R, заданное для сегмента. По умолчанию равно 3.

Перечисленные выше значения можно задавать как в качестве свойств сегмента, так и в параметрах запроса:

```
curl http://localhost:8091/riak/animals/ace?r=all
```

Требование читать со всех узлов таит в себе опасность: если какой-то узел выйдет из строя, Riak, возможно, не сумеет выполнить запрос. Проведем эксперимент – остановим сервер dev3.

```
$ dev/dev3/bin/riak stop
```

Теперь попытка читать со всех узлов вполне может завершиться неудачей (если это не так, попробуйте остановить также сервер dev2 или остановить dev1 и читать из порта 8092 либо 8093; мы не можем управлять тем, на какой v-узел пишет Riak).

```
$ curl -i http://localhost:8091/riak/animals/ace?r=all
HTTP/1.1 404 Object Not Found
Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic)
Date: Thu, 02 Jun 2011 17:18:18 GMT
Content-Type: text/plain
Content-Length: 10

not found
```

Если запрос не удалось выполнить, вы получаете ошибку 404 (Object Not Found), что в общем-то имеет смысл в данном контексте. Объект не найден, потому что для выполнения запроса было недостаточно копий. Разумеется, ничего хорошего в этом нет, поэтому в такой ситуации Riak выполняет *восстановление возможности чтения* (read repair): запрашивает N-кратную репликацию ключа на доступные серверы. Если вы попытаете снова обратиться к тому же URL,

то получите значение ключа, а не ошибку 404. В онлайн-официальной документации по Riak имеются отличные примеры¹⁰ на языке Erlang.

Но безопаснее просто затребовать кворум (данные с большинства, но не всех узлов):

```
curl http://localhost:8091/riak/animals/polly?r=quorum
```

Если выполнять запись в режиме кворума, который можно устанавливать на уровне отдельной операции записи, то операции чтения будут согласованы. На том же уровне можно задавать и величину W . Если вы не хотите ждать, пока Riak запишет данные хотя бы на один узел, то можете задать W равным нулю, что означает «я тебе доверяю, Riak, запишешь потом, а сейчас верни управление».

```
curl -X PUT http://localhost:8091/riak/animals/jean?w=0 \
-H "Content-Type: application/json" \
-d '{"nickname": "Jean", "breed": "Border Collie"}' \
```

Но несмотря на всю эту гибкость, обычно употребляются значения по умолчанию, если нет основательных причин поступить иначе. Задавать $W=0$ очень полезно для записи в журналы, а $W=N$, $R=1$ – для данных, которые записываются редко, но должны читаться очень быстро.

Запись и долговечная запись

Мы утаили от вас одну вещь. Для операций записи в Riak не гарантируется долговечность, то есть данные не пишутся на диск немедленно. Даже если запись на узел сочтена успешной, все равно не исключено, что в результате сбоя данные в этом узле будут потеряны – даже в случае, когда $W=N$. Перед записью на диск данные некоторое время хранятся в буфере в памяти, и вот эта-то доля миллисекунды и есть опасная зона.

Это дурные вести. Но есть и хорошие: в Riak имеется специальный параметр `dw` (*dw* – durable write, *долговечная запись*). В этом режиме операция производится медленнее, но риск снижается, так как Riak не возвращает управление, пока объект не будет физически записан на диск на заданном числе узлов. Как и в случае обычной записи, это свойство можно установить на уровне сегмента. Ниже мы присваиваем `dw` значение `one`, чтобы данные гарантированно были сохранены хотя бы на одном узле.

```
$ curl -X PUT http://localhost:8091/riak/animals \
-H "Content-Type: application/json" \
```

¹⁰ <http://wiki.basho.com/Replication.html>

```
-d '{"props":{"dw":"one"}}'
```

Если хотите, можете также переопределить это значение в конкретном запросе на запись, задав параметр `dw` в URL.

Замечание о временной передаче ответственности

Попытка записи на недоступные узлы тем не менее завершается успешно с кодом ответа «204 No Content». Объясняется это тем, что Riak записывает данные на расположенный рядом узел, где они хранятся до тех пор, пока не появится возможность перебросить их на ранее недоступный узел. Это прекрасная краткосрочная страховка, так как если один сервер выходит из строя, то его обязанности перехватывает какой-то другой узел Riak. Но, конечно, если все запросы, адресованные серверу А, передаются серверу В, то нагрузка на сервер В возрастает вдвое. Есть опасность, что в результате откажет и сервер В, тогда нагрузка ляжет на сервер С, сервер D и так далее. Такое явление, называемое *каскадным отказом*, встречается редко, но все же возможно. Считайте это предупреждением – не нагружайте серверы Riak до предела, поскольку заранее неизвестно, когда придется заметить выбывшего из строя бойца.

День 2: итоги

Сегодня мы рассмотрели две самых интересных особенности Riak: мощный механизм mapreduce и гибкие средства кластеризации. Технология mapreduce используется и во многих других обсуждаемых в этой книге базах данных, так что, если у вас остались вопросы, рекомендуем еще раз прочитать первую часть этого раздела, а также онлайн-документацию по Riak¹¹ и статьи в википедии¹².

День 2: домашнее задание

Информационный поиск

1. Прочитайте раздел о mapreduce в онлайн-документации по Riak.
2. Найдите репозиторий дополнительных функций Riak, в котором имеется много готовых mapreduce-функций.
3. Найдите в онлайн-документации полный перечень фильтров ключей, где имеется много полезных функций, например:

¹¹ <http://wiki.basho.com/MapReduce.html>

¹² <http://en.wikipedia.org/wiki/MapReduce>

преобразование строк в верхний регистр, сравнение числовых значений с границами диапазона, логические операции И/ИЛИ/НЕ и даже простая реализация сравнения строк с помощью расстояния Левенштейна.

Задачи

1. Напишите функции `map` и `reduce` для сегмента `rooms`, которые вычисляют общую вместимость номеров на каждом этаже.
2. Дополните написанные в предыдущем упражнении функции фильтром, который ограничивает вычисление только номерами на этажах 42 и 43.

3.4. День 3: разрешение конфликтов и расширение Riak

Сегодня мы будем говорить о менее очевидных аспектах Riak. Вы уже поняли, что Riak – простое хранилище ключей и значений, распределенное между кластером серверов. При наличии нескольких узлов возможны конфликты данных, и иногда их приходится разрешать. Riak предоставляет механизм определения того, какая операция записи имела место последней, – векторные часы и *одноуровневое разрешение* (sibling resolution).

Мы также увидим, как можно контролировать правильность входных данных с помощью точек подключения до и после фиксации. Мы расширим Riak, превратив его в персональную поисковую систему, воспользовавшись встроенными в Riak средствами поиска (с интерфейсом SOLR) и ускорения запросов за счет дополнительных индексов.

Разрешение конфликтов с помощью векторных часов

Векторные часы (vector clock)¹³ – это алгоритм, который в распределенных системах типа Riak используется для сохранения порядка конфликтующих обновлений пар ключ-значение. Следить за тем, в каком порядке производятся обновления, важно потому, что разные клиенты могут подключаться к разным серверам, и, значит, не исключено, что один клиент обновляет один сервер в то время, как другой клиент обновляет другой сервер (управлять тем, на какой сервер производится запись, вы не можете).

¹³ http://en.wikipedia.org/wiki/Vector_clock

Первое, что приходит в голову, – снабдить значения временными метками и отдавать предпочтение тому, у которого метка самая поздняя. Однако в кластере серверов это решение будет работать только, если все часы идеально синхронизированы. Riak такого требования не предъявляет, так как точная синхронизация часов в лучшем случае трудная, а часто просто невыполнимая задача. Использование централизованных часов противоречит всей философии Riak, поскольку такая система стала бы точкой общего отказа.

Векторные часы решают проблему, снабжая каждое событие (создание, обновление или удаление пары ключ-значение) меткой, содержащей информацию о том, какой клиент произвел изменение и в каком порядке. В результате сами клиенты или разработчик приложения могут решать, кто побеждает в случае конфликта. Читатель, знакомый с системами управления версиями типа Git или Subversion, увидит здесь аналогию с тем, как разрешаются конфликты, когда два человека изменяют один и тот же файл.

Векторные часы в теории

Предположим, что дела в гостинице для собак идут так хорошо, что вы можете себе позволить отбирать клиентуру. Для принятия оптимальных решений вы собираете комиссию из трех экспертов по собакам, которые должны сказать, какие кандидаты находятся в хорошем состоянии. Каждой собаке выставляется оценка от 1 (в плохом состоянии) до 4 (идеальный кандидат). Эксперты – назовем их Боб, Джейн и Ракшит – должны прийти к единогласному решению.

На компьютере каждого эксперта установлен клиент, подключающийся к базе данных. Клиент снабжает каждый запрос меткой, содержащей его уникальный идентификатор. Идентификатор клиента используется как для построения векторных часов, так и для запоминания в заголовке объекта клиента, выполнившего последнее обновление. Сначала рассмотрим простой псевдокод, а потом попытаемся реализовать его в Riak.

Боб создает объект первым и проставляет достойную оценку 3 новому щенку по кличке Амбал. В векторные часы записывается его имя и номер версии 1.

```
vclock: bob[1]
value: {score : 3}
```

Джейн читает эту запись и дает Амбалу оценку 2. Ее обновление произведено после того, как это сделал Боб, поэтому ее версия 1 добавляется в конец вектора `vclock`.


```
vclock: bob[1], jane[1]
value: {score : 2}
```

Одновременно Ракшит читает запись, созданную Бобом, а не Джейн. Ему понравился Амбал, поэтому он ставит ему оценку 4. Как и в предыдущем случае, идентификатор его клиента дописывается в конец вектора часов с номером версии 1.

```
vclock: bob[1], rakshith[1]
value: {score : 4}
```

В тот же день, но позже, Джейн (председатель комиссии) проверяет оценки. Поскольку вектор обновления Ракшита создан не после обновления Джейн, а одновременно с ним, то имеет место конфликт, который необходимо разрешить. Джейн получила оба значения и вправе решать, какое выбрать.

```
vclock: bob[1], jane[1]
value: {score : 2}
---
vclock: bob[1], rakshith[1]
value: {score : 4}
```

Она выбирает среднее, то есть изменяет оценку на 3.

```
vclock: bob[1], rakshith[1], jane[2]
value: {score : 3}
```

После разрешения конфликта всякий, кто запрашивает данные, получит самое последнее значение.

Векторные часы на практике

Теперь воплотим рассмотренный только что пример в Riak.

Мы хотим видеть все конфликтующие версии, чтобы можно было разрешить их вручную. Установим режим сохранения нескольких версий, присвоив `true` свойству `allow_mult` в сегменте `animals`. Несколько значений одного ключа называются *братьями* (sibling).

```
$ curl -X PUT http://localhost:8091/riak/animals \
-H "Content-Type: application/json" \
-d '{"props":{"allow_mult":true}}'
```

В следующем запросе Боб заносит данные об Амбале (Bruiser) в систему с оценкой 3 и идентификатором клиента *bob*.

```
$ curl -i -X PUT http://localhost:8091/riak/animals/bruiser \
-H "X-Riak-ClientId: bob" \
-H "Content-Type: application/json" \
-d '{"score": 3}'
```

Джейн и Ракшит одновременно читают данные об Амбале, созданные Бобом (вы увидите гораздо больше заголовков, мы показываем лишь векторные часы). Обратите внимание, что Riak кодирует векторные часы Боба, но по существу эта строка содержит идентификатор клиента и номер версии (и еще временную метку, поэтому ваша строка будет отличаться от приведенной ниже).

```
$ curl -i http://localhost:8091/riak/animals/bruiser?return_body=true
X-Riak-Vclock: a85hYGBgzGDKBVIS7NtEXmUwJTLmsTI8FMs5zpcFAA==
{"score" : 3}
```

Джейн добавляет свою оценку 2 и включает вектор часов, полученный из версии Боба. Таким образом, Riak узнаёт, что ее значение обновляет значение, присвоенное Бобом.

```
$ curl -i -X PUT http://localhost:8091/riak/animals/bruiser \
-H "X-Riak-ClientId: jane" \
-H "X-Riak-Vclock: a85hYGBgzGDKBVIS7NtEXmUwJTLmsTI8FMs5zpcFAA==" \
-H "Content-Type: application/json" \
-d '{"score" : 2}'
```

Поскольку Ракшит читал данные Боба одновременно с Джейн, то он отправляет свое обновление (с оценкой 4) вместе с тем же самым вектором часов, полученным от Боба.

```
$ curl -i -X PUT http://localhost:8091/riak/animals/bruiser \
-H "X-Riak-ClientId: rakshith" \
-H "X-Riak-Vclock: a85hYGBgzGDKBVIS7NtEXmUwJTLmsTI8FMs5zpcFAA==" \
-H "Content-Type: application/json" \
-d '{"score" : 4}'
```

Проверяя оценки, Джейн видит не значение, как можно было бы ожидать, а HTTP-код, означающий наличие нескольких вариантов, и тело, содержащее значения двух братьев.

```
$ curl http://localhost:8091/riak/animals/bruiser?return_body=true
Siblings:
637aZSiKy6281x1YrstzH5
7F85FBAIW8eiD9ubsBAeVk
```

Riak хранит версии в многочастном формате, поэтому, чтобы получить весь объект, необходимо указать готовность принять этот тип MIME.

```
$ curl -i http://localhost:8091/riak/animals/bruiser?return_body=true \
-H "Accept: multipart/mixed"
```

```
HTTP/1.1 300 Multiple Choices
X-Riak-Vclock: a85hYGBgyWDBVHs20Re...OYn9XY4sskQUA
```

```
Content-Type: multipart/mixed; boundary=1QwWn1ntX3gZmYQVBG6mAZRVXlu
Content-Length: 409
```

```
--1QwWn1ntX3gZmYQVBG6mAZRVXlu
Content-Type: application/json
Etag: 637aZSiKy6281x1YrstzH5
```

```
{"score" : 4}
--1QwWn1ntX3gZmYQVBG6mAZRVXlu
Content-Type: application/json
Etag: 7F85FBaiW8eiD9ubsBAeVk
```

```
{"score" : 2}
--1QwWn1ntX3gZmYQVBG6mAZRVXlu--
```

Обратите внимание, что показанные выше братья – на самом деле значения тегов сущностей (Etag) HTTP (в Riak они называются v-тегами – vtag). Попутно отметим, что можно задать в URL параметр vtag, запросив тем самым одну конкретную версию. Так, запрос `curl http://localhost:8091/riak/animals/bruiser?vtag=7F85FBaiW8eiD9ubsBAeVk` вернет `{"score" : 2}`. Теперь задача Джейн – воспользоваться этой информацией, чтобы произвести разумное обновление. Она решает усреднить обе оценки, то есть задать значение 3, используя векторные часы для разрешения конфликта.

```
$ curl -i -X PUT http://localhost:8091/riak/animals/bruiser?return_body=true \
-H "X-Riak-ClientId: jane" \
-H "X-Riak-Vclock: a85hYGBgyWDBVHs20Re...OYn9XY4sskQUA" \
-H "Content-Type: application/json" \
-d '{"score" : 3}'
```

Если теперь Боб и Ракшит запросят информацию об Амбале, то получат бесконфликтную оценку.

```
$ curl -i http://localhost:8091/riak/animals/bruiser?return_body=true
HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgyWDBVHs20Re...CpQmAkonCcHFM4CAA==
{"score" : 3}
```

На все последующие запросы будет возвращена оценка 3.

Время монотонно возрастает

Вы, наверное, обратили внимание, что вектор часов растет по мере того, как к процессу обновления значения подключаются новые клиенты. Это фундаментальная проблема векторных часов, и разработчики Riak ее понимали. Вектор часов может со временем «обрезаться», чтобы не допускать бесконтрольного роста. Частота обрезания

вектора часов является свойством сегмента, и ее можно узнать, как любое другое свойство.

```
$ curl http://localhost:8091/riak/animals
```

Показанные ниже свойства управляют применяемым в Riak алгоритмом обрезания вектора часов.

```
"small_vclock":10,"big_vclock":50,"young_vclock":20,"old_vclock":86400
```

Свойства *small_vclock* и *big_vclock* определяют минимальную и максимальную длину вектора, а свойства *young_vclock* и *old_vclock* – минимальный и максимальный возраст вектора часов.

Подробнее о векторных часах и обрезании вектора можно прочитать в онлайн-официальной документации¹⁴.

Точки подключения до и после фиксации

Riak умеет преобразовывать данные до и после сохранения объекта; для этого предусмотрены специальные точки подключения пользовательского кода. В рассматриваемых здесь точках подключения вызываются функции, написанные на JavaScript или Erlang и предназначенные для обработки данных до и после фиксации. Функция, вызываемая до фиксации, может модифицировать объект (или даже принудительно завершать операции с ошибкой), а вызываемая после фиксации – как-то отреагировать на успешную фиксацию (например, записать в журнал или отправить сообщение по электронной почте).

У каждого сервера имеется свой файл `app.config`, в который можно поместить ссылки на местоположение пользовательского кода на JavaScript. Сначала откройте файл `dev/dev1/etc/app.config` для сервера `dev1` и найдите в нем строку `js_source_dir`. Пропишите в ней путь к любому каталогу по своему усмотрению (но имейте в виду, что эта строка может быть закомментирована, то есть начинаться символом `%`, поэтому не забудьте удалить его). На нашем компьютере эта строка выглядит так:

```
{js_source_dir, "~/riak/js_source"},
```

Это изменение придется произвести трижды – для каждого сервера разработки. Давайте напомним функцию-валидатор, которая вызывается перед фиксацией, анализирует входные данные и проверяет, что оценка присутствует и находится в диапазоне от 1 до 4. Если хотя бы одно условие не выполняется, то возбуждается исключение и валидатор возвращает JSON-объект вида `{"fail" : message}`,

¹⁴ <http://wiki.basho.com/Vector-Clocks.html>

где `message` – сообщение, которое мы хотим передать пользователю. Если же данные корректны, то функция просто возвращает полученный на входе объект, и Riak его сохраняет.

riak/my_validators.js

```
function good_score(object) {
  try {
    /* извлечь из объект Riak данные и разобрать их как JSON-объект */
    var data = JSON.parse( object.values[0].data );
    /* если свойство score отсутствует, известить об ошибке */
    if( !data.score || data.score === '' ) {
      throw( 'Score is required' );
    }
    /* если свойство score вне диапазона, известить об ошибке */
    if( data.score < 1 || data.score > 4 ) {
      throw( 'Score must be from 1 to 4' );
    }
  } catch( message ) {
    /* Riak ожидает получить в случае ошибки такой JSON-объект */
    return { "fail" : message };
  }
  /* Все нормально, продолжаем */
  return object;
}
```

Сохраните этот файл в каталоге, прописанном в строке `js_source_dir`. Поскольку мы изменяем поведение сервера, то необходимо перезагрузить все серверы разработки, указав в командной строке аргумент `restart`.

```
$ dev/dev1/bin/riak restart
$ dev/dev2/bin/riak restart
$ dev/dev3/bin/riak restart
```

Riak найдет все файлы с расширением `.js` и загрузит их в память. Теперь, чтобы JavaScript-функция вызывалась из точки подключения перед фиксацией, следует задать в свойстве сегмента `precommit` имя *функции* (не имя файла!).

```
curl -X PUT http://localhost:8091/riak/animals \
  -H "content-type:application/json" \
  -d '{"props":{"precommit":[{"name" : "good_score"}]}}'
```

Протестируем нашу функцию, задав оценку больше 4. Поскольку функция проверяет, что оценка попадает в диапазон от 1 до 4, то следующий запрос завершается ошибкой.

```
curl -i -X PUT http://localhost:8091/riak/animals/bruise \
  -H "Content-Type: application/json" -d '{"score" : 5}'
HTTP/1.1 403 Forbidden
```

```
Content-Type: text/plain
Content-Length: 25
```

```
Score must be 1 to 4
```

Мы получаем код 403 Forbidden, а также текстовое сообщение об ошибке, которое было задано в поле «fail». Отправив GET-запрос для получения значения `bruiser`, можно убедиться, что оценка осталась прежней – 3. Попробуйте изменить ее на 2 – тут вас ждет успех.

Точка подключения после фиксации аналогична. Но здесь мы не будем ее рассматривать, так как функции, вызываемые из этой точки, должны быть написаны на языке Erlang. Если вы владеете этим языком, то можете прочесть в онлайн-официальной документации, как устанавливать собственные модули. На самом деле, на Erlang можно писать и функции `mapreduce`. Но мы продолжим наше путешествие по Riak рассмотрением других готовых модулей и расширений.

Расширение Riak

В комплекте с Riak поставляется несколько расширений, которые по умолчанию отключены, но добавляют ряд полезных поведений.

Поиск в Riak

Подсистема поиска в Riak просматривает данные в кластере и строит по ним инвертированный индекс. Возможно, вы еще не забыли об *инвертированных индексах*, рассматривавшихся в главе, посвященной PostgreSQL. Как и GIN-индексы в PostgreSQL, индексы Riak предназначены для быстрого поиска строк, но в распределенной системе.

Для использования поиска в Riak необходимо активировать соответствующее расширение в файлах `app.config` следующим образом:

```
%% Riak Search Config
{riak_search, [
  %% Чтобы включить поиск, установите этот параметр в 'true'.
  {enabled, true}
]},
```

Если вы знакомы с такими системами поиска, как Lucene, то эта часть покажется вам проще простого. Если нет, то и тогда этот материал легко освоить.

Мы должны уведомлять подсистему поиска обо всех изменениях в базе данных и можем воспользоваться для этого точкой подключения до фиксации. Установить функцию `precommit` из написанного

на Erlang модуля `riak_search_kv_hook` в сегмент `animals` позволит следующая команда:

```
$ curl -X PUT http://localhost:8091/riak/animals \
-H "Content-Type: application/json" \
-d '{"props":{"precommit":
[{"mod": "riak_search_kv_hook", "fun": "precommit"}]}}'
```

Обратившись к URL `curl http://localhost:8091/riak/animals`, мы увидим, что свойство `precommit` сегмента `animals` действительно изменено. Теперь при добавлении в сегмент `animals` любых данных в формате JSON или XML Riak будет индексировать имена и значения полей. Давайте загрузим данные о нескольких животных.

```
$ curl -X PUT http://localhost:8091/riak/animals/dragon \
-H "Content-Type: application/json" \
-d '{"nickname": "Dragon", "breed": "Briard", "score": 1 }'
$ curl -X PUT http://localhost:8091/riak/animals/ace \
-H "Content-Type: application/json" \
-d '{"nickname": "The Wonder Dog", "breed": "German Shepherd", "score": 3 }'
$ curl -X PUT http://localhost:8091/riak/animals/rtt \
-H "Content-Type: application/json" \
-d '{"nickname": "Rin Tin Tin", "breed": "German Shepherd", "score": 4 }'
```

Существует несколько способов запросить эти данные, но мы воспользуемся встроенным в Riak интерфейсом Solr на базе HTTP (он реализует поисковый интерфейс Apache Solr¹⁵). Для поиска в сегменте `animals` мы указываем в запросе путь `/solr`, за которым следует имя сегмента `/animals` и команда `/select`. В параметрах задаются поисковые слова. В данном случае мы хотим найти все объекты, в которых ключ *breed* (порода) имеет значение *Shepherd* (овчарка).

```
$ curl http://localhost:8091/solr/animals/select?q=breed:Shepherd
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      <str name="indent">on</str>
      <str name="start">0</str>
      <str name="q">breed:Shepherd</str>
      <str name="q.op">or</str>
      <str name="df">value</str>
      <str name="wt">standard</str>
      <str name="version">1.1</str>
      <str name="rows">2</str>
    </lst>
```

¹⁵ <http://lucene.apache.org/solr/>

```
</lst>
<result name="response" numFound="2" start="0" maxScore="0.500000">
  <doc>
    <str name="id">ace</str>
    <str name="breed">German Shepherd</str>
    <str name="nickname">The Wonder Dog</str>
    <str name="score">3</str>
  </doc>
  <doc>
    <str name="id">rtt</str>
    <str name="breed">German Shepherd</str>
    <str name="nickname">Rin Tin Tin</str>
    <str name="score">4</str>
  </doc>
</result>
</response>
```

Если вы предпочитаете получать ответ в формате JSON, добавьте параметр `wt=json`. В одном запросе можно задать несколько поисковых критериев, разделенных пробелами (%20 в URL-кодированной форме), добавив еще параметр `q.op=and`, означающий, что условия объединяются связкой И. Чтобы найти овчарок, кличка которых содержит слово *rin*, нужно выполнить следующий запрос:

```
$ curl http://localhost:8091/solr/animals/select\
?wt=json&q=nickname:rin%20breed:shepherd&q.op=and
```

Подсистема поиска в Riak допускает и другие синтаксические конструкции, например метасимволы (* соответствует нескольким символам, ? — одному), правда, только в конце поискового термина. Запрос `nickname:Drag*` найдет собаку Dragon, а запрос `nickname:*ragon` — не найдет. Еще одна полезная возможность — поиск по диапазону:

```
nickname:[dog TO drag]
```

Поддерживаются также более сложные запросы, включающие булевские операторы, группировку и поиск с учетом близости. Кроме того, можно определять нестандартные кодировки данных, создавать нестандартные индексы и даже указывать, какой индекс использовать при поиске. Перечень параметров, допустимых в URL, приведен в таблице ниже.

Параметр	Описание	По умолчанию
q	Сам запрос в виде строки	
q.op	Термы запроса могут объединяться связкой and или or	or

Параметр	Описание	По умолчанию
sort	По какому полю сортировать	не сортировать
start	Порядковый номер первого найденного объекта, включаемого в список возвращаемых	0
Rows	Максимальное число возвращаемых результатов	20
wt	Формат ответа: xml или json	xml
Index	Какой индекс использовать	

О расширении поиска в Riak можно рассказывать еще много, гораздо больше, чем позволяет эта книга. Надеемся, что некоторое представление о его возможностях вы получили. Понятно, что это расширение полезно для реализации поиска в большом веб-приложении, но даже если речь идет о простых запросах в более-менее произвольной форме, к нему все равно стоит присмотреться.

Индексирование в Riak

Начиная с версии 1.0, Riak поддерживает дополнительные индексы. Они похожи на индексы в PostgreSQL, но имеют некоторые особенности. Индексируется не отдельный столбец или набор столбцов, а метаданные, присоединенные к заголовку объекта.

Для включения этого расширения также придется изменить файл `app.config`. В качестве подсистемы хранения укажите `eLevelDB` вместо `bitcask`, как показано ниже, и перезапустите серверы:

```
{riak_kv, [
  %% Параметр storage_backend определяет модуль Erlang, описывающий
  %% механизм хранения в данном узле.
  {storage_backend, riak_kv_eleveldb_backend},
```

`eLevelDB` – это реализация на Erlang придуманного Google хранилища ключей и значений под названием `LevelDB`¹⁶. Эта подсистема допускает построение дополнительных индексов в Riak.

Подготовив систему, мы можем проиндексировать любой объект с произвольным количеством специальных заголовков, которые называются *индексными записями* и определяют способ индексирования. Имена полей начинаются префиксом `x-riak-index-` и заканчиваются суффиксом `_int` или `_bin`, означающим «целое» и «двоичное» (любое, кроме целого) значение соответственно.

¹⁶ <http://code.google.com/p/leveldb/>

Добавляя бульдога Blue II, изображенного на эмблеме спортивной команды Butler Bulldogs Батлеровского университета, мы хотим проиндексировать его по названию университета (*butler*) и по номеру «версии» (Blue 2 – вторая эмблема с изображением бульдога).

```
$ curl -X PUT http://localhost:8098/riak/animals/blue
-H "x-riak-index-mascot_bin: butler"
-H "x-riak-index-version_int: 2"
-d '{"nickname" : "Blue II", "breed" : "English Bulldog"}'
```

Вероятно, вы заметили, что индексы не имеют никакого отношения к хранимому значению ключа. Это чрезвычайно полезная возможность, так как позволяет строить индексы, независимые от хранимых данных. Так можно проиндексировать даже видеоролики.

Получить значение по индексу несложно.

```
$ curl http://localhost:8098/riak/animals/index/mascot_bin/butler
```

Хотя дополнительные индексы в Riak – большой шаг в правильном направлении, простор для развития еще есть. Например, чтобы проиндексировать дату, необходимо сохранить ее в виде строки в формате, пригодном для сортировки – “YYYYMMDD”. Числа с плавающей точкой нужно сначала умножить на подходящую степень 10, а затем сохранить в виде целого – $1.45 * 100 == 145$. Такие преобразования возлагаются на клиента. Тем не менее, сочетание в Riak *mapreduce*, подсистемы поиска, а теперь и дополнительного индексирования существенно ослабляет ограничения, присущие классическим хранилищам ключей и значений, и дает куда более богатую функциональность.

День 3: итоги

Мы завершили знакомство с Riak рассмотрением более сложных механизмов: разрешение конфликтов с помощью векторных часов, а также проверка и модификация входных данных с помощью точек подключения при фиксации. Мы также обсудили два полезных расширения Riak: подсистему поиска и индексирование данных для придания дополнительной гибкости запросам. В сочетании с каркасом *mapreduce*, рассмотренным во второй день, и ссылками, рассмотренными в первый день, это позволяет создавать гибкие решения, далеко превосходящие всё, на что способны классические хранилища ключей и значений.

День 3: домашнее задание

Информационный поиск

1. Найдите репозиторий дополнительных функций для Riak (подсказка: он находится на сайте GitHub).
2. Ознакомьтесь с дополнительной литературой о векторных часах.
3. Прочитайте о том, как создавать собственную конфигурацию индексов.

Задачи

1. Создайте собственный индекс, определяющий схему `animals`. Точнее, укажите, что поле `score` – целое число и сформулируйте запрос по диапазону его значений.
2. Подготовьте небольшой кластер из трех серверов (например, ноутбуков или экземпляров EC2¹⁷) и на каждый установите Riak. Организуйте из серверов кластер. Установите набор данных Google stock с информацией об акциях (его можно найти на сайте компании Basho¹⁸).

3.5. Резюме

Riak – первая из рассмотренных нами баз данных NoSQL. Это распределенное реплицируемое хранилище ключей и значений с дополнительными функциями, не имеющее точки общего отказа.

Если прежде вы работали только с реляционными базами данных, то Riak поначалу может показаться странным созданием. В нем нет ни транзакций, ни SQL, ни схемы. Есть ключи, но связывание сегментов совсем не похоже на соединение таблиц. А технология `mapreduce` и вовсе кажется темным лесом.

Однако для решения определенного класса задач все эти компромиссы оправданы. Способность Riak масштабироваться путем увеличения количества серверов (а не наращивания мощности одного сервера) и простота использования – великолепный пример попытки решить специфические проблема масштабируемости в веб. И вместо того чтобы изобретать велосипед, Riak опирается на уже имеющуюся структуру HTTP, предоставляя максимальную гибкость для построения каркасов или систем с поддержкой веб.

¹⁷ <http://aws.amazon.com/ec2/>

¹⁸ <http://wiki.basho.com>Loading-Data-and-Running-MapReduce-Queries.html>

Сильные стороны Riak

Если вы собираетесь спроектировать крупномасштабную систему заказов наподобие Amazon или ставите на первое место высокую доступность, то Riak, безусловно, заслуживает внимания. Один из несомненных плюсов Riak – внимание к устранению точек общего отказа с целью обеспечить бесперебойную работу и наращивание (или сокращение) кластера в соответствии с изменяющимися требованиями. Если структура данных не очень сложна, то работать с Riak будет просто. Но при этом сохраняется возможность изощренных манипуляций с данными, если это необходимо. В настоящее время поддерживается десяток языков (полный перечень можно найти на сайте Riak), но, если вы готовы писать на Erlang, то можете расширять ядро в любую сторону. А если требуется большее быстродействие, чем способен обеспечить протокол HTTP, то можете попробовать обмениваться данными по более эффективному двоичному транспортному протоколу Protobuf¹⁹.

Слабые стороны Riak

Если требуются простые средства формулирования запросов, сложные структуры данных или жесткая схема или если нет нужды в горизонтальном масштабировании, то Riak вам, пожалуй, ни к чему. Один из основных недостатков Riak – отсутствие простых и надежных средств формулирования произвольных запросов, хотя движение в этом направлении, безусловно, есть. Технология mapreduce дает фантастическую функциональность, но мы бы хотели видеть больше встроенных действий, основанных на URL или PUT-запросах. Добавление индексирования было крупным шагом в правильном направлении, но нам бы хотелось бы, чтобы эти идеи получили дальнейшее развитие. Наконец, если вы не хотите писать на Erlang, то можете столкнуться с некоторыми ограничениями JavaScript – в частности, невозможности написать функции, вызываемые в точке подключения после фиксации, – и недостаточным быстродействием mapreduce. Однако команда Riak работает над устранением этих мелких шероховатостей.

Riak и теорема CAP

Riak остроумно обходит ограничения, налагаемые теоремой CAP на любую распределенную базу данных. И это поразительно, если срав-

¹⁹ <http://code.google.com/p/protobuf/>

нить с PostgreSQL, которая (по большей части) поддерживает только строгую согласованность записи. В Riak используется высказанная в статье Amazon о системе Дупато идея о том, что условия CAP можно задавать на уровне сегментов или запросов. Это большой шаг в сторону надежной и гибкой СУБД с открытым исходным кодом. Читая о других базах данных, вспоминайте о Riak – вас постоянно будет поражать его гибкость.

Перед расставанием

Если вам необходимо хранить гигантский каталог данных, то Riak может оказаться неплохим выбором. Хотя исследования в области реляционных баз данных ведутся уже более сорока лет, не каждая задача нуждается в свойствах ACID или жестком контроле схемы. Если требуется база данных, встраиваемая в устройство, или нужно обрабатывать финансовые транзакции, то Riak не подойдет. Если задача в том, чтобы обеспечить горизонтальное масштабирование или обслужить интенсивный поток запросов через веб, то к Riak стоит присмотреться.



ГЛАВА 4.

HBase

Система Apache HBase создана для серьезных работ, как строительный гвоздепистолет. Вам вряд ли придет в голову использовать HBase для хранения корпоративной сводной декларации, как не придет в голову с помощью гвоздепистолета сколачивать кукольный домик. Если объем данных не измеряется многими гигабайтами, то стоит взять что-нибудь попроще.

На первый взгляд, HBase очень похожа на реляционную базу данных – настолько, что, не зная, с чем имеете дело, можно спутать одно с другим. Самая сложная часть в изучении HBase – не технология; проблема в том, что многие термины, применяемые в HBase, обманчиво знакомы. Например, HBase хранит данные в контейнерах, которые называются *таблицами*. Таблицы же состоят из *ячеек*, находящихся на пересечении *строк* и *столбцов*. Пока всё хорошо, верно?

Неверно! Таблицы HBase ведут себя совсем не так, как отношения, строки ничуть не похожи на записи, а состав столбцов может быть переменным (схема его не контролирует). Проектирование схемы по-прежнему остается важным делом, поскольку она влияет на производительность системы, но порядок в доме она не наведет. HBase – это дьявольское отражение или, если хотите, Бизарро¹ РСУБД.

Тогда зачем же пользоваться этой базой данных? Даже если отвлечься от масштабируемости, есть несколько причин. Во-первых, в HBase встроен ряд функций, отсутствующих в других базах данных, например, версионирование, сжатие, сборка мусора (для данных с истекшим сроком хранения) и таблицы в памяти. Раз эти возможности присутствуют изначально, значит, вам придется писать меньше кода, когда они потребуются. Кроме того, HBase дает строгие гарантии непротиворечивости, что упрощает переход от реляционных баз данных.

1 Бизарро (англ. Bizarro) – вымышленный персонаж вселенной комиксов DC Comics. Персонаж был создан как зеркальное отражение Супермена. *Прим. перев.*

Благодаря всему этому HBase может быть краеугольным камнем систем оперативной аналитической обработки – в этой роли она просто сверкает. Отдельные операции могут выполняться медленнее, чем их эквиваленты в других базах данных, но просмотр гигантских наборов данных – дело, которым HBase занимается с особым удовольствием. Поэтому при обработке трудоемких запросов HBase часто обгоняет другие СУБД. Этим объясняется, почему HBase так часто применяется в крупных компаниях для реализации систем анализа журналов и поиска.

4.1. Введение в HBase

HBase – это *столбцовая* база данных, которая может похвастаться поддержанием непротиворечивости и горизонтальной масштабируемостью. Она устроена по образцу BigTable, высокопроизводительной закрытой базы данных, которая разработана Google и описана в статье 2006 года «Bigtable: A Distributed Storage System for Structured Data»². Первоначально HBase создавалась для обработки естественных языков и начинала свою жизнь как дополнительный пакет к проекту Apache Hadoop. Но с тех пор превратилась в проект Apache верхнего уровня.

С архитектурной точки зрения, в HBase изначально заложена отказоустойчивость. Отказ отдельной машины – событие относительно редкое, но в большом кластере отказы узлов являются нормой. Использование упреждающей записи в журнал и распределенной конфигурации позволяет HBase быстро восстанавливаться после отказов отдельных серверов.

К тому же, HBase обитает в экосистеме, которая сама по себе дает дополнительные преимущества. HBase построена на базе Hadoop – стабильной и масштабируемой вычислительной платформы, которая предоставляет распределенную файловую систему и средства mapreduce. Всюду, где есть HBase, вы найдете также Hadoop и другие инфраструктурные компоненты, которые можете задействовать и в собственных приложениях.

Эта база данных активно используется и разрабатывается рядом крупных компаний для решения задач, связанных с «большими данными». В частности, Facebook выбрала HBase как основной компонент своей инфраструктуры обмена сообщениями, анонсированной в ноябре 2010 года. На сайте StumbleUpon HBase уже несколько лет

² <http://research.google.com/archive/bigtable.html>

применяется в качестве хранилища данных, работающего в режиме реального времени, и обработки аналитических данных. Некоторые функции сайта берут данные непосредственно из HBase. В Twitter HBase также используется для самых разных целей – от генерации данных (для таких приложений, как поиск людей) до хранения результатов мониторинга и данных о производительности. Среди громких имен, использующих HBase, можно назвать также eBay, Meetup, Ning, Yahoo! и многие другие.

При такой активности неудивительно, что новые версии HBase выходят весьма часто. На момент написания этой книги текущая стабильная версия имела номер 0.90.3, с ней мы и будем работать. Итак, скачивайте HBase и начнем.

4.2. День 1: операции CRUD и администрирование таблиц

Наша сегодняшняя цель – изучить составные части HBase. Сначала мы запустим локальный экземпляр HBase в автономном режиме, а потом покажем, как в оболочке HBase создавать и изменять таблицы, вставлять и модифицировать данные. Затем мы посмотрим, как некоторые из этих операций программируются с помощью HBase Java API на языке JRuby. Попутно мы расскажем об архитектурных концепциях HBase, в том числе связях между строками, семействах столбцов, столбцах и значениях.

Полностью работоспособный кластер HBase промышленного качества должен включать по меньшей мере пять узлов – по крайней мере, таково общепринятое мнение. Но для наших целей такая конфигурация избыточна. К счастью, HBase поддерживает три режима работы:

- автономный режим с одной-единственной машиной;
- псевдораспределенный режим с одним узлом, притворяющимся кластером;
- полностью распределенный режим с кластером совместно работающих узлов.

В этой главе мы по большей части будем запускать HBase в автономном режиме. Но даже это не совсем тривиально, поэтому мы не сможем рассмотреть все аспекты установки и администрирования, но в некоторых местах будет отмечать возможные проблемы.

Конфигурирование HBase

Прежде всего HBase необходимо сконфигурировать. Конфигурационные параметры хранятся в файле `hbase-site.xml`, который находится в каталоге `${HBASE_HOME}/conf/`. Переменная окружения `HBASE_HOME` должна указывать на каталог, в который установлена HBase.

Первоначально этот файл содержит всего лишь пустой тег `<configuration>`. Но в него можно добавлять определения свойств в следующем формате:

```
<property>
  <name>some.property.name</name>
  <value>A property value</value>
</property>
```

Полный перечень поддерживаемых свойств вместе со значениями по умолчанию и описаниями находится в файле `hbase-default.xml` в каталоге `${HBASE_HOME}/src/main/resources`. По умолчанию HBase хранит файлы данных во временном каталоге. Это означает, что *все данные будут потеряны*, когда операционная система решит освободить место на диске.

Чтобы не потерять данные, следует указать какое-нибудь не столь уязвимое место для хранения файлов. Запишите в свойство `hbase.rootdir` путь к подходящему каталогу:

```
<property>
  <name>hbase.rootdir</name>
  <value>file:///path/to/hbase</value>
</property>
```

Чтобы запустить HBase, откройте терминал (окно команд) и выполните следующую команду:

```
${HBASE_HOME}/bin/start-hbase.sh
```

Для останова HBase выполните команду `stop-hbase.sh` в том же каталоге. Если что-то пойдет не так, взгляните на недавно модифицированные файлы в каталоге `${HBASE_HOME}/logs`. В системах *nix следующая команда выводит на консоль записываемые в журналы данные по мере их поступления.

```
cd ${HBASE_HOME}
find ./logs -name "hbase-*.log" -exec tail -f {} \;
```

Оболочка HBase

Оболочка HBase – это написанная на JRuby командная программа для интерактивной работы с HBase. В оболочке можно добавлять и удалять таблицы, изменять схему таблицы, добавлять и удалять данные и выполнять целый ряд других операций. Позже мы рассмотрим другие способы подключения к HBase, а пока нашим родным домом станет оболочка. Убедившись, что HBase работает, откройте терминал и запустите оболочку:

```
${HBASE_HOME}/bin/hbase shell
```

Чтобы убедиться, что все функционирует нормально, запросите номер версии.

```
hbase> version
0.90.3, r1100350, Sat May 7 13:31:12 PDT 2011
```

В любой момент можно ввести команду `help` и получить список доступных команд или сведения об использовании конкретной команды.

Затем выполните команду `status`, которая возвращает краткие сведения о состоянии сервера HBase.

```
hbase> status
1 servers, 0 dead, 2.0000 average load
```

Если в какой-нибудь команде произойдет ошибка или оболочка зависнет, то, возможно, имеет место проблема с подключением. HBase делает всё возможное, чтобы автоматически сконфигурировать свои службы в соответствии с настройками сети, но иногда ошибается. Если вы наблюдаете подобные симптомы, обратитесь к врезке «Сетевые настройки HBase».

Создание таблицы

Словарем (`map`) называется набор пар ключ-значение. Это аналог хеша в Ruby или структуры данных `HashMap` в Java. Таблица в HBase по существу представляет собой огромный словарь. Или, если быть точным, словарь словарей.

Сетевые настройки HBase

По умолчанию HBase пытается сделать свои службы доступными внешним клиентам, но в нашем случае требуется подключаться с той же машины, где работает сервер. Поэтому имеет смысл добавить некоторые или все пере-

численные ниже свойства в файл `hbase-site.xml` (хотя всё зависит от конкретной конфигурации). Отметим, что показанные значения будут полезны, только если вы планируете подключаться локально, а не удаленно.

Имя свойства	Значение
<code>hbase.master.dns.interface</code>	<code>Lo</code>
<code>hbase.master.info.bindAddress</code>	<code>127.0.0.1</code>
<code>hbase.regionserver.info.bindAddress</code>	<code>127.0.0.1</code>
<code>hbase.regionserver.dns.interface</code>	<code>Lo</code>
<code>hbase.zookeeper.dns.interface</code>	<code>Lo</code>

Эти свойства говорят HBase, как устанавливать соединения с главным сервером и с региональными серверами (о тех и других мы еще будем говорить ниже), а также со службой конфигурирования `zookeeper`. Строка «`Lo`» обозначает так называемый возвратный интерфейс. В системах `*nix` возвратный интерфейс не является настоящим сетевым интерфейсом (как сетевые карты Ethernet или беспроводной связи), а реализован программно, чтобы компьютер мог установить соединение сам с собой. Свойства `bindAddress` говорят HBase, какой IP-адрес прослушивать.

В таблице HBase ключи – это произвольные текстовые строки, каждая из которых отображается на *строку* данных. Строка данных сама является словарем, в котором ключи называются *столбцами*, а значения интерпретируются как массивы байтов. Столбцы группируются в *семейства столбцов*, так что полное имя столбца состоит из двух частей: имя семейства и *квалификатор столбца*. Часто они сцепляются вместе с двоеточием в качестве разделителя (например, `'family:qualifier'`).

Все эти понятия иллюстрируются на рис. 13. Здесь изображена гипотетическая таблица с двумя семействами столбцов: `color` и `shape`. В таблице есть две строки – обозначены пунктирными прямоугольниками, которые идентифицируются своими ключами: `first` и `second`. В строке `first` мы видим три столбца из семейства `color` (с квалификаторами `red`, `blue` и `yellow`) и один столбец из семейства `shape` (`square`). Комбинация ключа строки и имени столбца (включающего семейство и квалификатор) дает адрес данных. В этом примере кортеж `first/color:red` указывает на значение `'#F00'`.

А теперь воспользуемся всем, что узнали о структуре таблицы, и займемся кое-чем интересным – мы собираемся разработать вики!

С вики можно ассоциировать много разной информации, но мы ограничимся абсолютным минимумом. Вики-сайт состоит из страниц, и с каждой страницей связаны уникальное название и текст какой-то статьи.

	ключи строк	семейство столбцов "color"	семейство столбцов "shape"
строка	"first"	"red": "#F00" "blue": "#00F" "yellow": "#FF0"	"square": "4"
строка	"second"		"triangle": "3" "square": "4"

Рис. 13. Таблицы в HBase состоят из строк, ключей, семейств столбцов, столбцов и значений

Для создания таблицы мы воспользуемся командой `create`:

```
hbase> create 'wiki', 'text'
0 row(s) in 1.2160 seconds
```

Здесь создается таблица с именем `wiki` с единственным семейством столбцов `text`. В настоящий момент таблица пуста, в ней нет строк, а, значит, нет и столбцов. В отличие от реляционных баз данных, в HBase столбец – неотъемлемая принадлежность содержащей его строки. Когда мы начнем добавлять строки, то одновременно будем добавлять и столбцы для хранения данных.

Структура нашей таблицы наглядно представлена на рис. 14. Мы ожидаем, что в каждой строке будет ровно один столбец из семейства `text`, квалифицированный пустой строкой (' '). Таким образом, полное имя столбца, содержащего текст страницы, будет равно `'text: '`.

Разумеется, таблица вики полезна, только если в ней есть какое-то содержимое. Так добавим его!

Вставка, обновление и выборка данных

Нашему вики-сайту нужна начальная страница, с нее и начнем. Для добавления данных в таблицу HBase служит команда `put`:

```
hbase> put 'wiki', 'Home', 'text:', 'Welcome to the wiki!'
```

Она вставляет в таблицу `wiki` новую строку с ключом `'Home'` и помещает текст `'Welcome to the wiki!'` в столбец `'text: '`.

	ключи строк (названия страниц вики)	семейство столбцов "text"
строка (страница)	"название первой страницы"	"". "текст первой страницы"
строка (страница)	"название второй страницы"	"". "текст второй страницы"

Рис. 14. Таблица wiki содержит только одно семейство столбцов

Запросить данные из строки 'Home' можно командой `get`, которая требует двух параметров: имя таблицы и ключ строки. Дополнительно можно задать список возвращаемых столбцов.

```
hbase> get 'wiki', 'Home', 'text:'
COLUMN CELL
text: timestamp=1295774833226, value=Welcome to the wiki!
1 row(s) in 0.0590 seconds
```

Обратите внимание на поле `timestamp` в возвращенных данных. HBase хранит вместе с каждым значением данных целочисленную временную метку – количество миллисекунд, прошедших с момента «эпохи» (00:00:00 UTC 1 января 1970). Когда в ячейку записывается новое значение, старое не стирается, а остается вместе со своей временной меткой. Это потрясающе удобная возможность. В большинстве баз данных за сохранение исторических данных отвечает программист, а в HBase версионирование уже встроено!

Пример: индексная таблица сообщений в Facebook

В Facebook база данных HBase является основным компонентом инфраструктуры сообщений, где используется как для хранения самих данных сообщений, так и для поддержания инвертированного индекса, предназначенного для поиска. Индексная таблица устроена следующим образом.

- Строки ключей – это идентификаторы пользователей.
- Квалификаторы столбцов – слова, встречающиеся в сообщениях пользователей.
- Временные метки выступают в роли идентификаторов сообщений, содержащих слово.

Поскольку сообщения после ввода уже не изменяются, то и записи о сообщениях в индексе тоже статичны. Версионирование в данном случае ни к чему. Но Facebook использует временные метки в качестве идентификаторов сообщений, бесплатно получая таким образом поле для хранения данных.

Команды put и get

Команды `put` и `get` позволяют явно задавать временную метку. Если количество миллисекунд с начала эпохи вас чем-то не устраивает, можете выбрать другое целое значение. Это дает дополнительное поле, если оно вам нужно. Если временная метка не задана, то HBase по умолчанию использует текущее время в момент вставки, а при чтении возвращает последнюю версию.

Пример изобретательного использования временной метки описан на врезке «Пример: индексная таблица сообщений в Facebook». Далее в этой главе мы будем интерпретировать временную метку, как принято по умолчанию.

Изменение таблиц

Пока что в схеме таблицы `wiki` есть только названия страниц, их тексты и интегрированная история версий – больше ничего. Предъявим новые требования:

- страница уникально идентифицируется своим названием;
- у страницы может быть неограниченное количество редакций;
- каждая редакция идентифицируется своей временной меткой;
- редакция содержит текст и необязательный комментарий, задаваемый при сохранении;
- у редакции есть автор, идентифицируемый своим именем.

Эти требования наглядно представлены на рис. 15. Как видим, у каждой редакции есть автор, комментарий при сохранении, текст статьи и временная метка. Название страницы не является частью редакции, оно идентифицирует все редакции, относящиеся к одной и той же странице.

После переложения этих требований на язык таблиц HBase получается картина, изображенная на рис. 16. В нашей таблице `wiki` название выступает в роли ключа строки, а прочие данные о странице группируются в два семейства столбцов: `text` и `revision`. Семейство столбцов `text` не изменилось; мы ожидаем, что в каждой строке есть ровно один столбец, квалифицированной пустой строкой (‘’),

в котором будет храниться содержимое статьи. Семейство столбцов `revision` предназначено для всей остальной информации о редакции, например, автора и комментария при сохранении.

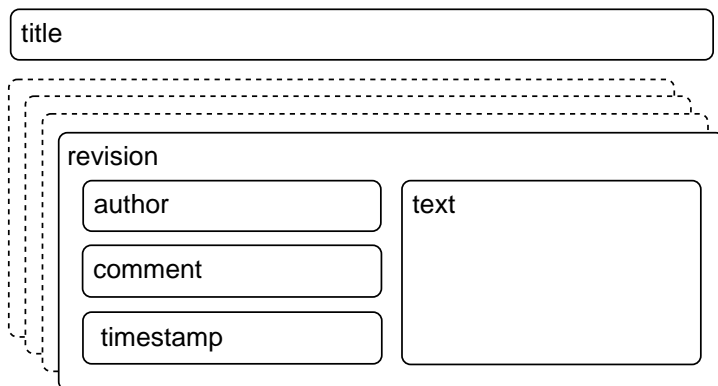


Рис. 15. Требования к таблице `wiki` (включая временную метку)

	ключи (title)	семейство "text"	семейство "revision"
строка (страница)	"первая страница"	"" : "..." "" : "..."	"author": "..." "comment": "..."
строка (страница)	"вторая страница"	"" : "..." "" : "..."	"author": "..." "comment": "..."

Рис. 16. Измененная структура таблицы `wiki`
(временная метка не показана)

Значения по умолчанию

Поскольку мы не задавали для таблицы `wiki` никаких специальных параметров, всюду используются принятые в HBase значения по умолчанию. Одно из них говорит, что сохранять следует только три версии значений в столбцах. Увеличим его. Чтобы внести изменения в схему, необходимо сначала перевести таблицу в автономный режим командой `disable`.

```
hbase> disable 'wiki'
0 row(s) in 1.0930 seconds
```

Теперь можно модифицировать характеристики семейства столбцов командой `alter`.

```
hbase> alter 'wiki', { NAME => 'text', VERSIONS =>
hbase* org.apache.hadoop.hbase.HConstants::ALL_VERSIONS }
0 row(s) in 0.0430 seconds
```

Здесь мы просим HBase изменить атрибут `VERSIONS` семейства столбцов `text`. Существуют и другие допускающие установку атрибуты, некоторые из них мы рассмотрим завтра. Обозначение `hbase*` говорит, что эта строка является продолжением предыдущей.

Изменение таблицы

Операции, изменяющие характеристики семейства столбцов, могут обходиться очень дорого, потому что HBase должна создать новое семейство, а затем скопировать данные. В производственной системе это может повлечь за собой длительный простой. Поэтому лучше правильно описывать семейство столбцов с самого начала.

Пока таблица `wiki` находится в автономном режиме, добавим семейство столбцов `revision`, снова воспользовавшись командой `alter`:

```
hbase> alter 'wiki', { NAME => 'revision', VERSIONS =>
hbase* org.apache.hadoop.hbase.HConstants::ALL_VERSIONS }
0 row(s) in 0.0660 seconds
```

Как и в случае `text`, мы добавляем в схему только *семейство столбцов* `revision`, а не отдельные *столбцы*. Мы, конечно, ожидаем, что в конечном итоге каждая строка будет содержать столбцы `revision:author` и `revision:comment`, но удовлетворить эти ожидания — обязанность клиента; в формальной схеме такие требования отсутствуют. Если кто-нибудь захочет добавить столбец `revision:foo`, HBase возражать не будет.

Продолжаем

Внеся эти изменения, мы можем снова активировать `wiki`:

```
hbase> enable 'wiki'
0 row(s) in 0.0550 seconds
```

Теперь наша таблица поддерживает расширенные требования, и можно приступить к заполнению столбцов из семейства `revision` данными.

Добавление данных из программы

Как мы только что видели, оболочка HBase прекрасно справляется с манипулированием структурой таблиц. К сожалению, о поддержке вставки данных такого не скажешь. Команда `put` позволяет за один раз задать значение только одного столбца, а нам необходимо одновременно добавить несколько значений, чтобы у всех была одинаковая временная метка. Поэтому придется написать скрипт.

Приведенный ниже скрипт можно выполнить непосредственно в оболочке HBase, потому что она является также интерпретатором языка JRuby. Скрипт добавляет новую версию текста для страницы Home, одновременно заполняя поля автора и комментария. JRuby исполняется виртуальной машиной Java (JVM), предоставляя ей доступ к написанному на Java коду HBase. Этот и следующие примеры не будут работать с версиями Ruby, реализованными не в JVM.

hbase/put_multiple_columns.rb

```
import 'org.apache.hadoop.hbase.client.HTable'
import 'org.apache.hadoop.hbase.client.Put'

def jbytes( *args )
  args.map { |arg| arg.to_s.to_java_bytes }
end

table = HTable.new( @hbase.configuration, "wiki" )

p = Put.new( *jbytes( "Home" ) )

p.add( *jbytes( "text", "", "Hello world" ) )
p.add( *jbytes( "revision", "author", "jimbo" ) )
p.add( *jbytes( "revision", "comment", "my first edit" ) )

table.put( p )
```

Предложения `import` включают в оболочку ссылки на полезные классы HBase. Это избавляет от необходимости в дальнейшем записывать полные пространства имен. Функция `jbytes()` принимает произвольное число аргументов и возвращает массив байтовых массивов Java, ожидаемый методами HBase API.

Затем мы создаем локальную переменную `table`, указывающую на нашу таблицу `wiki`, для чего пользуемся административным объектом `@hbase`, получая от него конфигурационную информацию.

Далее мы создаем и настраиваем новый экземпляр команды `Put`, который принимает подлежащую изменению строку. В данном слу-

чае нас интересует страница Home. Наконец, в экземпляре Put методом `add()` добавляются свойства, после чего мы просим объект `table` выполнить подготовленную операцию `put`. У метода `add()` есть несколько вариантов, мы воспользовались вариантом с тремя аргументами: `add(column_family, column_qualifier, value)`.

Зачем нужны семейства столбцов?

Может возникнуть искушение построить всю инфраструктуру вообще без семейств столбцов; почему бы не хранить все данные строки в одном семействе? Да, такое решение реализовать проще. Но у него есть и недостатки – и, прежде всего, невозможность тонкой настройки производительности. Для каждого семейства столбцов параметры, определяющие такие характеристики, как скорость чтения и записи и объем занятого места на диске, задаются независимо.

Все операции в HBase атомарны *на уровне строк*. Сколько бы столбцов ни было в строке, любая операция видит согласованное представление той строки, которую читает или модифицирует. Такое проектное решение позволяет клиентам делать осмысленные предположения о данных.

Наша операция `put` затрагивает несколько столбцов и не обращается явно к полю `timestamp`, поэтому у значений во всех столбцах будет одна и та же временная метка (текущее время в миллисекундах). Убедимся в этом, выполнив операцию `get`.

```
hbase> get 'wiki', 'Home'
COLUMN      CELL
revision:   author timestamp=1296462042029, value=jimbo
revision:   comment timestamp=1296462042029, value=my first edit
text:       timestamp=1296462042029, value=Hello world
3 row(s) in 0.0300 seconds
```

Как видите, значение `timestamp` во всех перечисленных выше столбцах действительно одинаково.

День 1: итоги

Сегодня мы получили первое представление о запуске сервера HBase. Мы научились конфигурировать его и пользоваться журналами для поиска и устранения неисправностей. С помощью оболочки HBase мы выполнили простейшие операции администрирования и манипуляции с данными. Моделируя схему вики-сайта, мы познакомились с проектированием схем в HBase. Мы узнали, как создавать таблицы и работать с семействами столбцов. Проектирование схемы в HBase

сводится к принятию решений о параметрах семейств столбцов и, что не менее важно, о семантической интерпретации временных меток и ключей строк.

Мы также сделали первый шаг в исследовании HBase Java API, выполнив в оболочке скрипт, написанный на JRuby. Завтра мы сделаем следующий шаг, воспользовавшись оболочкой для запуска скриптов, выполняющих такие длительные задания, как импорт данных.

Хочется надеяться, что вы уже начали расставаться с принятой в реляционных базах семантикой терминов *таблица*, *строка* и *столбец*. Различие между смыслом этих слов в HBase и в других системах станет еще более разительным, когда мы познакомимся с функциональностью HBase глубже.

День 1: домашнее задание

Онлайновую документацию HBase можно грубо разбить на две категории: сугубо техническая и отсутствующая. Это положение дел постепенно изменяется – появляются руководства типа «Приступая к работе», но будьте готовы к тому, что для поиска ответа на вопрос придется покопаться в документации, сгенерированной Javadoc, или в исходном коде.

Информационный поиск

1. Найдите, как выполнить в оболочке следующие операции:
 - удалить из строки значения, хранящиеся в указанных столбцах;
 - удалить всю строку.
2. Поставьте закладку на документацию HBase API для той версии HBase, которой пользуетесь.

Задачи

1. Напишите функцию `put_many()`, которая создает экземпляр `Put`, добавляет в него произвольное количество пар столбец-значение и сохраняет в таблице. Сигнатура функции должна выглядеть так:

```
def put_many( table_name, row, column_values )  
  # здесь должен быть ваш код  
end
```

2. Определите свою функцию `put_many()`, скопировав ее текст в оболочку HBase, а затем вызвав следующим образом:

```
hbase> put_many 'wiki', 'Some title', {  
hbase*   "text:" => "Some article text",
```

```
hbase* "revision:author" => "jschmoe",
hbase* "revision:comment" => "no comment" }
```

4.3. День 2: работа с «большими данными»

Освоив в первый день создание таблиц и манипуляции с данными, мы можем приступить к добавлению в таблицу `wiki` настоящих данных. Сегодня мы займемся написанием скрипта, который в конечном итоге закачает на наш вики-сайт содержимое википедии! Попутно мы расскажем о некоторых приемах ускорения импорта. И напоследок покопаемся во внутреннем устройстве HBase и разберемся, как данные разбиваются на регионы, чтобы повысить производительность и упростить аварийное восстановление.

Импорт данных, выполнение скриптов

Одна из типичных проблем, с которой сталкивается всякий, кто исследует новую СУБД, – как перенести в нее данные. Описанное в первый день задание статических строк в операции `Put` – это, конечно, хорошо, но есть способы и получше.

К счастью, копирование текста команды в оболочку – не единственный способ выполнить ее. При запуске оболочки HBase из командной строки можно указать имя JRuby-скрипта – HBase выполнит его, как если бы он был введен прямо в оболочке. Синтаксис выглядит следующим образом:

```
${HBASE_HOME}/bin/hbase shell <your_script> [<optional_arguments> ...]
```

Поскольку нас интересуют именно «большие данные», то напишем скрипт, который будет импортировать в нашу таблицу `wiki` статьи из википедии. Организация Wikimedia Foundation, которая курирует проекты Wikipedia, Wictionary и другие, периодически публикует выгруженные наборы данных, которыми мы вполне можем воспользоваться. Эти наборы представляют собой гигантские XML-файлы. Вот пример записи из англоязычной википедии:

```
<page>
<title>Anarchism</title>
<id>12</id>
<revision>
<id>408067712</id>
<timestamp>2011-01-15T19:28:25Z</timestamp>
```

```

<contributor>
  <username>RepublicanJacobite</username>
  <id>5223685</id>
</contributor>
<comment>Undid revision 408057615 by [[Special:Contributions...</comment>
<text xml:space="preserve">{{Redirect|Anarchist|the fictional character|
...
[[bat-smg:Anarkézm0s]]
</text>
</revision>
</page>

```

Мы предусмотрительно включили в свою схему всю присутствующую здесь информацию: название (ключ строки), текст, временную метку и автора. Поэтому скрипт импорта редакций не должен быть слишком сложным.

Потоковая загрузка XML

Будем действовать по порядку. Нам необходимо разобрать огромный XML-файл, рассматривая его как поток данных (с помощью интерфейса SAX), с этого и начнем. Базовая структура JRuby-программы для последовательного разбора XML-файла выглядит примерно так:

hbase/basic_xml_parsing.rb

```

import 'javax.xml.stream.XMLStreamConstants'

factory = javax.xml.stream.XMLInputFactory.newInstance
reader = factory.createXMLStreamReader(java.lang.System.in)

while reader.has_next

  type = reader.next

  if type == XMLStreamConstants::START_ELEMENT
    tag = reader.local_name
    # сделать что-то с тегом
  elsif type == XMLStreamConstants::CHARACTERS
    text = reader.text
    # сделать что-то с текстом
  elsif type == XMLStreamConstants::END_ELEMENT
    # то же, что для START_ELEMENT
  end
end

```

Здесь следует обратить внимание на несколько моментов. Сначала мы создаем экземпляр `XMLStreamReader` и связываем его с объектом `java.lang.System.in`, говоря тем самым, что собираемся читать из стандартного ввода.

Затем в цикле `while` мы считываем лексемы из потока XML, пока не дойдем до конца. В теле цикла производится обработка лексем. Действия зависят от того, является ли текущая лексема открывающим тегом, закрывающим тегом или текстом между тегами.

Загрузка википедии

Теперь можно объединить логику обработки XML с изученными ранее классами `HTable` и `Put`. Окончательный скрипт приведен ниже. В основном, здесь все уже вам знакомо, а новые вещи мы обсудим подробнее.

hbase/import_from_wikipedia.rb

```
require 'time'

import 'org.apache.hadoop.hbase.client.HTable'
import 'org.apache.hadoop.hbase.client.Put'
import 'javax.xml.stream.XMLStreamConstants'

def jbytes( *args )
  args.map { |arg| arg.to_s.to_java_bytes }
end

factory = javax.xml.stream.XMLInputFactory.newInstance
reader = factory.createXMLStreamReader(java.lang.System.in)

❶ document = nil
buffer = nil
count = 0

table = HTable.new( @hbase.configuration, 'wiki' )
❷ table.setAutoFlush( false )

while reader.has_next
  type = reader.next

  ❸ if type == XMLStreamConstants::START_ELEMENT

    case reader.local_name
    when 'page' then document = {}
    when /title|timestamp|username|comment|text/ then buffer = []
    end

  ❹ elsif type == XMLStreamConstants::CHARACTERS

    buffer << reader.text unless buffer.nil?

  ❺ elsif type == XMLStreamConstants::END_ELEMENT
```

```

case reader.local_name
when /title|timestamp|username|comment|text/
  document[reader.local_name] = buffer.join
when 'revision'
  key = document['title'].to_java_bytes
  ts = ( Time.parse document['timestamp'] ).to_i

  p = Put.new( key, ts )
  p.add( *jbytes( "text", "", document['text'] ) )
  p.add( *jbytes( "revision", "author", document['username'] ) )
  p.add( *jbytes( "revision", "comment", document['comment'] ) )
  table.put( p )
  count += 1
  table.flushCommits() if count % 10 == 0
  if count % 500 == 0
    puts "#{count} records inserted (#{document['title']})"
  end
end
end
end

table.flushCommits()
exit

```

- ❶ Первое заметное отличие – появление нескольких переменных:
 - `document`: текущая статья и данные о ее редакциях;
 - `buffer`: символьные данные из текущего поля документа (текст, название, автор и т. д.);
 - `count`: счетчик уже импортированных статей.
- ❷ Обратите особое внимание на вызов `table.setAutoFlush(false)`. HBase периодически *автоматически сбрасывает* данные на диск. Для большинства приложений этот режим очень удобен. Если мы отменим автосброс, то все выполняемые операции `put` накапливаются в буфере, пока явно не будет вызван метод `table.flushCommits()`. Это позволяет собирать операции записи в пакет и выполнять их, когда нам удобно.
- ❸ Далее посмотрим, что происходит на этапе разбора. Если открывающий тег – `<page>`, то мы инициализируем документ, присваивая в качестве значения пустой хеш. Если же это какой-то другой из интересующих нас тегов, то мы очищаем буфер для хранения текста.
- ❹ Обработка символьных данных сводится к добавлению в конец буфера.
- ❺ Для большинства закрывающих тегов нужно просто поместить содержимое буфера в `document`. Но для тега `</revision>` мы создаем новый экземпляр `Put`, заполняем его содержимым полей документа и сохраняем в таблице. После этого мы на каждой десятой итерации вызываем метод `flushCommits()`, а на каждой пятистотой печатаем на стандартный вывод сообщение о ходе обработки.

Сжатие и фильтры Блума

Мы уже почти готовы запустить скрипт, нужно только сделать еще одно дело. Семейство столбцов `text` будет содержать большие блоки текста, и хорошо бы их сжимать. Поэтому включим режим сжатия и быстрого поиска:

```
hbase> alter 'wiki', {NAME=>'text', COMPRESSION=>'GZ', BLOOMFILTER=>'ROW'}  
0 row(s) in 0.0510 seconds
```

HBase поддерживает два алгоритма сжатия: Gzip (GZ) и Лемпеля-Зива-Оберхумера (LZO). Сообщество HBase настоятельно рекомендует предпочесть алгоритм LZO, а не Gzip, но здесь мы все же используем GZ.

С алгоритмом LZO связана проблема лицензирования. Хотя его исходный код открыт, но предоставляется по лицензии, не совместимой с принципами Apache, поэтому LZO нельзя включать в дистрибутив HBase. На сайте приведены подробные инструкции о том, как установить и настроить LZO. Если вам нужно высокопроизводительное сжатие, установите LZO.

Фильтр Блума – очень любопытная структура данных, которая по существу позволяет ответить на вопрос: «Встречался ли этот объект раньше?». Изначально Бэртон Ховард Блум (Burton Howard Bloom) разработал его в 1970 году для проверки орфографии, но с тех пор фильтры Блума стали применяться в приложениях для хранения данных, чтобы быстро определить, существует ли некоторый ключ. Краткое введение в эту тему приведено на врезке «Как работают фильтры Блума».

В HBase фильтры Блума используются для того, чтобы узнать, существует ли в строке с указанным ключом некоторый столбец (`BLOOMFILTER=>'ROWCOL'`), а также для того, чтобы выяснить, существует ли вообще строка с данным ключом (`BLOOMFILTER=>'ROW'`). Количество столбцов в семействе столбцов и количество строк практически ничем не ограничены. Фильтр Блума позволяет быстро определить, существуют ли данные, еще до выполнения дорогостоящей операции чтения с диска.

Контакт? Есть контакт!

Вот теперь всё готово к запуску скрипта. Напомним, что размер файлов огромен, поэтому о том, чтобы скачать и распаковать их и речи быть не может. А как же тогда быть?

К счастью, с помощью волшебных конвейеров, имеющихся в любой *nix-системе, мы можем в одной команде скачать XML-файл, распаковать его и передать на вход нашему скрипту:

```
curl <dump_url> | bzipcat | \  
${HBASE_HOME}/bin/hbase shell import_from_wikipedia.rb
```

Вместо <dump_url> следует подставить URL того или иного выгруженного набора данных WikiMedia Foundation³. Возьмите файл [project]-latest-pages-articles.xml.bz2, содержащий либо англоязычную википедию (~6 ГБ)⁴, либо англоязычный вики-словарь (~185 МБ)⁵. Эти файлы содержат последние редакции страниц в пространстве имен Main, то есть опущены страницы пользователей, страницы обсуждения и т. д.

Подставляйте URL-адрес – и вперед! По экрану будут бежать строчки такого вида:

```
500 records inserted (Ashmore and Cartier Islands)  
1000 records inserted (Annealing)  
1500 records inserted (Ajanta Caves)
```

Как работают фильтры Блума

Если не вдаваться в детали реализации, то фильтр Блума представляет собой статический битовый массив, в котором все элементы первоначально равны 0. Всякий раз, как фильтру подается новый блок данных, некоторые биты переключаются в 1. Какие именно, зависит от сгенерированного хеша данных, который затем преобразуется в позиции битов.

Если впоследствии мы захотим узнать, предьявлялся ли фильтру конкретный блок данных, то фильтр вычислит, какие биты должны быть для него подняты, и проверит, действительно ли они равны 1. Если хотя бы один из этих битов равен 0, то фильтр уверенно отвечает «нет». Если же все равны 1, то фильтр отвечает «да, есть шанс, что этот блок ранее предьявлялся». Однако чем больше блоков было введено, тем выше вероятность ложноположительного ответа.

В этом и заключается отличие фильтра Блума от простого хеша. Хеш никогда не дает ложноположительный ответ, зато и объем памяти для его хранения ничем не ограничен. Размер фильтра Блума фиксирован, но иногда он дает ложноположительные ответы, причем вероятность зависит от уровня насыщения и поддается точной оценке.

Скрипт будет радостно работать, пока не встретит ошибку, но, скорее всего, вы захотите прервать его раньше. Для этого нажмите CTRL+C. Но пока не торопитесь, дайте скрипту поработать, чтобы мы

³ <http://dumps.wikimedia.org>

⁴ <http://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2>

⁵ <http://dumps.wikimedia.org/enwiktionary/latest/enwiktionary-latest-pages-articles.xml.bz2>

могли заглянуть под капот и разобраться в механизме горизонтальной масштабируемости HBase.

Знакомство с регионами и мониторингом места на диске

В HBase строки хранятся отсортированными по ключу. Регионом называется множество строк, определяемое начальным ключом (включительно) и конечным (исключительно). Регионы не перекрываются, и каждому из них назначается региональный сервер в кластере. В нашей упрощенной конфигурации, где имеется единственный автономный сервер, существует всего один сервер регионов, который отвечает за все регионы. Но в полностью распределенном кластере региональных серверов будет много.

Давайте посмотрим, как сервер HBase использует место на диске, это поможет понять, как размещаются данные. Для этого перейдите в каталог, указанный в параметре `hbase.rootdir`, и выполните команду `du`. Эта стандартная команда `*nix` говорит, сколько места занимает каталог и – рекурсивно – все его подкаталоги. Флаг `-h` означает, что `du` должна выводить числа в удобном для человека виде.

Вот что было показано на нашей машине, когда было вставлено примерно 12 000 страниц и импорт все еще продолжался:

```
$ du -h
231M  ./logs/localhost.localdomain,38556,1300092965081
231M  ./logs
4.0K  ./META./1028785192/info
12K   ./META./1028785192/.oldlogs
28K   ./META./1028785192
32K   ./META.
12K   ./-ROOT-/70236052/info
12K   ./-ROOT-/70236052/.oldlogs
36K   ./-ROOT-/70236052
40K   ./-ROOT-
72M   ./wiki/517496fecabb7d16af7573fc37257905/text
1.7M  ./wiki/517496fecabb7d16af7573fc37257905/revision
61M   ./wiki/517496fecabb7d16af7573fc37257905/.tmp
12K   ./wiki/517496fecabb7d16af7573fc37257905/.oldlogs
134M  ./wiki/517496fecabb7d16af7573fc37257905
134M  ./wiki
4.0K  ./oldlogs
365M  .
```

Отсюда можно сделать полезные выводы о том, как HBase использует место на диске. Строки, начинающиеся с `/wiki`, относятся к

таблице wiki. Подкаталог с длинным именем 517496fecabb7d16af7573fc37257905 представляет один регион (пока единственный). Его подкаталоги /text и /revision соответствуют семействам столбцов text и revision. Наконец, в последней строке приведен суммарный итог – всего HBase заняла на диске 365 МБ.

И еще одно. Верхние две строчки, начинающиеся с / .logs, показывают, сколько места занимают журналы упреждающей записи (WAL). В HBase упреждающая запись применяется для защиты от отказов узлов. Это довольно типичная техника аварийного восстановления. Так, в файловых системах упреждающая запись в журнал называется *журналированием*. В HBase информация в WAL заносится до фиксации результатов операций редактирования (put и increment) на диске.

Из соображений производительности результаты редактирования не пишутся на диск немедленно. Система работает гораздо быстрее, когда ввод/вывод буферизуется и запись на диск производится блоками. Если в течение этого времени региональный сервер, отвечающий за соответствующий регион, выйдет из строя, то HBase сможет использовать WAL, чтобы понять, какие операции были успешно выполнены, и предпринять корректирующие действия.

Запись в WAL необязательна, но по умолчанию включена. В классах Put и Increment имеется метод установки setWriteToWAL(), позволяющий отменить запись конкретной операции в WAL. Вообще говоря, лучше использовать режим по умолчанию, но в некоторых случаях имеет смысл его изменить. Например, когда производится импорт, который в любой момент можно повторить (как в нашем скрипте импорта из википедии), отключение записи в WAL позволяет пожертвовать возможностью аварийного восстановления ради достижения более высокой производительности.

Опрос регионов

Если позволить скрипту работать достаточно долго, то можно увидеть, как HBase разбивает таблицу на несколько регионов. Вот как в нашем случае выглядела выдача du после добавления примерно 150 000 страниц:

```
$ du -h
40K      ../logs/localhost.localdomain,55922,1300094776865
44K      ../logs
24K      ../META./1028785192/info
4.0K     ../META./1028785192/recovered.edits
4.0K     ../META./1028785192/.tmp
```

```

12K      ./META./1028785192/.oldlogs
56K      ./META./1028785192
60K      ./META.
4.0K     ./corrupt
12K      ./-ROOT-/70236052/info
4.0K     ./-ROOT-/70236052/recovered.edits
4.0K     ./-ROOT-/70236052/.tmp
12K      ./-ROOT-/70236052/.oldlogs
44K      ./-ROOT-/70236052
48K      ./-ROOT-
138M     ./wiki/0a25ac7e5d0be211b9e890e83e24e458/text
5.8M     ./wiki/0a25ac7e5d0be211b9e890e83e24e458/revision
4.0K     ./wiki/0a25ac7e5d0be211b9e890e83e24e458/.tmp
144M     ./wiki/0a25ac7e5d0be211b9e890e83e24e458
149M     ./wiki/15be59b7dfd6e71af9b828fed280ce8a/text
6.5M     ./wiki/15be59b7dfd6e71af9b828fed280ce8a/revision
4.0K     ./wiki/15be59b7dfd6e71af9b828fed280ce8a/.tmp
155M     ./wiki/15be59b7dfd6e71af9b828fed280ce8a
145M     ./wiki/0ef3903982fd9478e09d8f17b7a5f987/text
6.3M     ./wiki/0ef3903982fd9478e09d8f17b7a5f987/revision
4.0K     ./wiki/0ef3903982fd9478e09d8f17b7a5f987/.tmp
151M     ./wiki/0ef3903982fd9478e09d8f17b7a5f987
135M     ./wiki/a79c0f6896c005711cf6a4448775a33b/text
6.0M     ./wiki/a79c0f6896c005711cf6a4448775a33b/revision
4.0K     ./wiki/a79c0f6896c005711cf6a4448775a33b/.tmp
141M     ./wiki/a79c0f6896c005711cf6a4448775a33b
591M     ./wiki
4.0K     ./oldlogs
591M     .

```

Главное отличие состоит в том, что старый регион (517496fe-cabb7d16af7573fc37257905) пропал, а вместо него появились четыре новых. В автономном режиме все эти регионы обслуживаются одним и тем же сервером, но в распределенной среде за них отвечали бы разные региональные серверы.

В этой связи возникает ряд вопросов, например: «Откуда региональные серверы знают, за какие регионы несут ответственность?» и «Как узнать, какой регион (и, следовательно, региональный сервер) обслуживает данную строку?».

В оболочке HBase мы можем опросить таблицу `.META.`, чтобы получить дополнительные сведения об имеющихся регионах. `.META.` — это специальная таблица, единственное назначение которой — отслеживать все пользовательские таблицы и региональные серверы, отвечающие за обслуживание регионов каждой таблицы.

```
hbase> scan '.META.', { COLUMNS => [ 'info:server', 'info:regioninfo' ] }
```

Даже при небольшом количестве регионов мы получаем массу информации. Вот фрагмент того, что было получено на нашем компьютере, – после форматирования и убирания лишнего.

ROW

```
wiki,,1300099733696.a79c0f6896c005711cf6a4448775a33b.
```

COLUMN+CELL

```
column=info:server, timestamp=1300333136393, value=localhost.localdomain:3555
column=info:regioninfo, timestamp=1300099734090, value=REGION => {
  NAME => 'wiki,,1300099733696.a79c0f6896c005711cf6a4448775a33b.',
  STARTKEY => '',
  ENDKEY => 'Demographics of Macedonia',
  ENCODED => a79c0f6896c005711cf6a4448775a33b,
  TABLE => {{...}}
```

ROW

```
wiki,Demographics of Macedonia,1300099733696.0a25ac7e5d0be211b9e890e83e24e458.
```

COLUMN+CELL

```
column=info:server, timestamp=1300333136402, value=localhost.localdomain:35552
column=info:regioninfo, timestamp=1300099734011, value=REGION => {
  NAME => 'wiki,Demographics of Macedonia,1300099733696.0a25...e458.',
  STARTKEY => 'Demographics of Macedonia',
  ENDKEY => 'June 30',
  ENCODED => 0a25ac7e5d0be211b9e890e83e24e458,
  TABLE => {{...}}
```

Как видим, оба региона обслуживаются одним и тем же сервером localhost.localdomain:35552. Первый регион начинается с пустой строки (') и заканчивается строкой 'Demographics of Macedonia'. Второй регион начинается со строки 'Demographics of Macedonia' и заканчивается строкой 'June 30'.

Ключ STARTKEY включается в регион, а ключ ENDKEY исключается. Если бы мы искали строку 'Demographics of Macedonia', то нашли бы ее во втором регионе.

Поскольку строки таблицы хранятся в отсортированном виде, то информацию из таблицы .META. можно использовать для того, чтобы найти в каком регионе (и на каком сервере) находится данная строка. Но где хранится сама таблица .META.?

Оказывается, что таблица .META. разбита на регионы и обслуживается региональными серверами, как любая другая таблица. Чтобы найти, какие серверы отвечают за различные части таблицы .META., необходимо просканировать таблицу -ROOT-.

```
hbase> scan '-ROOT-', { COLUMNS => [ 'info:server', 'info:regioninfo' ] }
```

```
ROW
.META.,,1
COLUMN+CELL
column=info:server, timestamp=1300333135782, value=localhost.localdomain:35552
column=info:regioninfo, timestamp=1300092965825, value=REGION => {
  NAME => '.META.,,1',
  STARTKEY => '',
  ENDKEY => '',
  ENCODED => 1028785192,
  TABLE => {...}}
```

Распределением регионов, в том числе принадлежащих таблице .META., по региональным серверам отвечает *главный* узел, который часто называют HBaseMaster. Главный сервер может одновременно выполнять функции регионального сервера.

Если региональный сервер выходит из строя, то главный сервер вмешивается и передает кому-то ответственность за обслуживание регионов, назначенных отказавшему узлу. Новый смотритель заглянет в WAL и посмотрит, нужны ли какие-нибудь восстановительные действия. Если же выходит из строя главный сервер, то его роль принимает на себя один из региональных серверов.

Сканирование одной таблицы для построения другой

Прервав скрипт импорта, мы можем перейти к следующей задаче: извлечению информации из импортированного содержимого вики. Страницы вики-сайта изобилуют ссылками – какие-то ведут на другие статьи, а какие-то – на внешние ресурсы. Такая перекрестная структура таит в себе богатейшие залежи данных о топологии. Давайте их исследуем!

Наша цель – представить связи между статьями как направленные ссылки с одной статьи на другую. Ссылка на внутреннюю статью в вики-тексте имеет вид `[[<target name>|<alt text>]]`, где `<target name>` – статья, на которую указывает ссылка, а `<alt text>` – альтернативный отображаемый текст (необязательный).

Например, если текст статьи о фильме *Star Wars* (Звездные войны) содержит строку `[[Yoda|jedi master]]` (Йода|мастер-джедаей), то мы должны сохранить связь дважды: один раз как исходящую ссылку из *Star Wars*, а другой – как входящую из Yoda. В таком случае мы сможем быстро найти как все ссылки, исходящие из страницы, так и все ссылки, ведущие на страницу.

Куда делась схема моей таблицы?

Схема `TABLE` не показана в примере сканирования информации о регионах. Это сделано для того, чтобы уменьшить объем текста, а о настройке производительности мы будем говорить ниже. Если вам не терпится увидеть определение схемы таблицы, воспользуйтесь командой `describe`, например:

```
hbase> describe 'wiki'
hbase> describe '.META.'
hbase> describe '-ROOT-'
```

Для хранения дополнительных данных о ссылках мы создадим новую таблицу. Зайдите в оболочку и введите такую команду:

```
hbase> create 'links', {
  NAME => 'to', VERSIONS => 1, BLOOMFILTER => 'ROWCOL'
},{
  NAME => 'from', VERSIONS => 1, BLOOMFILTER => 'ROWCOL'
}
```

В принципе, можно было бы не заводить новую таблицу, а включить данные о ссылках в одно из уже существующих семейств столбцов или добавить в таблицу `wiki` одно или несколько семейств. Но у создания новой таблицы есть то преимущество, что с ней связаны отдельные регионы. Следовательно, кластер сможет при необходимости более эффективно разбивать таблицы на регионы.

Вобщемслучае сообщество HBase рекомендует сводить количество семейств столбцов в одной таблице к минимуму. Сделать это можно двумя способами: включить в одно семейство больше столбцов или выносить семейства в отдельные таблицы. Что предпочесть, зависит от того, как часто клиентам нужна вся строка данных (а не несколько столбцов из нее).

В нашем случае семейства столбцов `text` и `revision` должны находиться в одной таблице, чтобы при добавлении новой редакции у метаданных и текста были одинаковые временные метки. Напротив, у содержимого таблицы `links` временные метки не имеют ничего общего со статьями, откуда извлечены ссылки. Кроме того, большинство клиентов заинтересованы либо в тексте статьи, либо в информации о ссылках, но не в том и другом одновременно. Поэтому вынесение семейств столбцов `to` и `from` в отдельную таблицу оправдано.

Построение сканера

Создав таблицу `links`, мы можем перейти к написанию скрипта, который будет сканировать все строки таблицы `wiki`. Из каждой строки он извлечет викитекст и выделит в нем ссылки. И для каждой

найденной ссылки создаст входящую и исходящую запись в таблице links. Большая часть скрипта вам уже знакома. Многие фрагменты использованы повторно, а о новых мы расскажем ниже.

hbase/generate_wiki_links.rb

```
import 'org.apache.hadoop.hbase.client.HTable'
import 'org.apache.hadoop.hbase.client.Put'
import 'org.apache.hadoop.hbase.client.Scan'
import 'org.apache.hadoop.hbase.util.Bytes'

def jbytes( *args )
  return args.map { |arg| arg.to_s.to_java_bytes }
end

wiki_table = HTable.new( @hbase.configuration, 'wiki' )
links_table = HTable.new( @hbase.configuration, 'links' )
links_table.setAutoFlush( false )

❶ scanner = wiki_table.getScanner( Scan.new )

linkpattern = /\[([^\[\]\|\\:\#\][^\[\]\|:]*)(?:\|([^\[\]\|]+))?\]\]/
count = 0

while (result = scanner.next())
  ❷ title = Bytes.toString( result.getRow() )
  text = Bytes.toString( result.getValue( *jbytes( 'text', '' ) ) )
  if text
    put_to = nil
  ❸ text.scan(linkpattern) do |target, label|
    unless put_to
      put_to = Put.new( *jbytes( title ) )
      put_to.setWriteToWAL( false )
      end

    target.strip!
    target.capitalize!

    label = '' unless label
    label.strip!

    put_to.add( *jbytes( "to", target, label ) )
    put_from = Put.new( *jbytes( target ) )
    put_from.add( *jbytes( "from", title, label ) )
    put_from.setWriteToWAL( false )
  ❹ links_table.put( put_from )
  end
  ❺ links_table.put( put_to ) if put_to
  links_table.flushCommits()
end
```



```

count += 1
puts "#{count} pages processed (#{title})" if count % 500 == 0

end

links_table.flushCommits()
exit

```

- ❶ Сначала мы создаем объект `Scan`, с помощью которого будем сканировать таблицу `wiki`.
- ❷ Для извлечения данных из строк и столбцов необходимо манипулировать байтами, но ничего страшного здесь нет.
- ❸ Всякий раз, как в тексте страницы обнаруживается образец `linkpattern`, мы извлекаем конечную статью и текст ссылки, после чего передаем их объектам `Put`.
- ❹–❺ Напоследок мы просим таблицу выполнить сразу все накопившиеся операции `Put`. Возможно (хотя и маловероятно), что статья вообще не содержит ссылок, отсюда и условие `if put_to`.

Вызов метод `setWriteToWAL(false)` — чисто субъективное решение. С одной стороны, это всего лишь учебное упражнение, а, с другой, мы всегда можем перезапустить скрипт в случае ошибки, поэтому мы решили выбрать скорость и смириться с судьбой, если узел выйдет из строя.

Запуск скрипта

Если вы готовы, отбросив сомнения, идти ва-банк, то запускайте скрипт.

```
{HBASE_HOME}/bin/hbase shell generate_wiki_links.rb
```

Скрипт должен печатать примерно следующее:

```

500 pages processed (10 petametres)
1000 pages processed (1259)
1500 pages processed (1471 BC)
2000 pages processed (1683)
...

```

Как и раньше, можете дать скрипту доработать до конца или в любой момент прервать его, нажав `CTRL+C`.

За использованием места на диске можно следить с помощью команды `du` — мы уже это делали. Вы увидите новые строчки, относящиеся к только что созданной таблице `links`, причем указанный в них размер по мере работы скрипта будет монотонно возрастать.

А нельзя ли сделать то же самое с помощью Mapreduce?

Во введении мы говорили, что примеры будут написаны на языках (J)Ruby и JavaScript. JRuby плохо «дружит» с Hadoop, но если вы готовы программировать на Java, то могли бы написать код сканера в виде задачи mapreduce и передать его Hadoop.

Вообще говоря, подобные задачи идеально приспособлены для реализации с помощью mapreduce. Имеются объемные входные данные, которые может обработать распределитель (сканирование таблицы HBase) и масса операций по формированию выходных данных (запись строки в другую таблицу HBase), которые может пакетно выполнять редуктор.

Hadoop ожидает, что задачи (экземпляры класса `Job`) написаны на Java и инкапсулированы (вместе со всеми зависимостями) в jar-файл, который можно разослать по всем узлам кластера. Современные версии JRuby поддерживают расширение Java-классов, но та, что поставляется в комплекте с HBase, этого не умеет.

Существует несколько проектов с открытым исходным кодом, наводящих мосты между JRuby и Hadoop, но ни один из них не работает с HBase. Ходят слухи что в будущем инфраструктура HBase будет включать абстракции, которые позволят писать задачи mapreduce на JRuby. Что ж, будем надеяться.

Исследование результатов

Мы только что написали программный сканер для решения сложной задачи. Теперь с помощью команды оболочки `scan` просто выведем часть содержимого таблицы на консоль. Для каждой ссылки, которую скрипт находит в столбце `text:`, он создает записи `to` и `from` в таблице `links`. Чтобы увидеть, какие ссылки созданы, зайдите в оболочку и просканируйте таблицу.

```
hbase> scan 'links', STARTROW => "Admiral Ackbar", ENDROW => "It's a Trap!"
```

Будет выведено очень много информации. Но, разумеется, никто не мешает воспользоваться командой `get`, чтобы вывести ссылки для одной статьи:

```
hbase> get 'links', 'Star Wars'
```

COLUMN CELL

...	
links:from:Admiral Ackbar	timestamp=1300415922636, value=
links:from:Adventure	timestamp=1300415927098, value=
links:from:Alamogordo, New Mexico	timestamp=1300415953549, value=
links:to:"weird al" yankovic	timestamp=1300419602350, value=
links:to:20th century fox	timestamp=1300419602350, value=
links:to:3-d film	timestamp=1300419602350, value=
links:to:Aayla segura	timestamp=1300419602350, value=
...	

В таблице `wiki` структура строк очень регулярна. Как вы помните, в любой строке имеются столбцы `text:`, `revision:author` и `revision:comment`. В таблице `links` такой регулярности уже нет. В одной строке может быть как один, так и сотни столбцов. А названия столбцов столь же разнообразны, сколь сами ключи строки (названия статей википедии). И ничего страшного! HBase потому и называется хранилищем разреженных данных.

Чтобы узнать, сколько строк сейчас имеется в таблице, можно воспользоваться командой `count`.

```
hbase> count 'wiki', INTERVAL => 100000, CACHE => 10000
Current count: 100000, row: Alexander wilson (vauxhall)
Current count: 200000, row: Bachelor of liberal studies
Current count: 300000, row: Brian donlevy
...
Current count: 2000000, row: Thomas Hobbes
Current count: 2100000, row: Vardousia
Current count: 2200000, row: Würrstadt (verbandsgemeinde)
2256081 row(s) in 173.8120 seconds
```

Из-за распределенной архитектуры HBase не хранит информацию о количестве строк в каждой таблице. Чтобы получить эту величину, строки необходимо сосчитать (выполнив полное сканирование таблицы). К счастью, наличие регионов позволяет сканировать таблицу распределенно. Так что даже в случае, когда для решения задачи нужно просканировать всю таблицу, это не так страшно, как в других базах данных.

День 2: итоги

Уф, тот еще выдался денёк! Мы научились писать скрипт импорта, который загружает данные в таблицу HBase, разбирая поток XML-данных. Затем мы применили эту технику для закидывания содержимого википедии в нашу таблицу `wiki`. Мы узнали много нового о HBase API, в частности о некоторых доступных клиенту рычагах управления производительностью: `setAutoFlush()`, `flushCommits()`, `setWriteToWAL()`. Попутно мы обсудили ряд архитектурных особенностей HBase, в том числе аварийное восстановление за счет упреждающей записи в журнал.

И, раз уж речь зашла об архитектуре, то мы рассмотрели регионы таблиц и распределение ответственности за их обслуживание между региональными серверами. Просканировав таблицы `.META.` и `-ROOT-`, мы кое-что узнали о внутреннем устройстве HBase.

И наконец, мы обсудили, как разреженное хранение, заложенное в проект HBase, влияет на производительность. И при этом затронули некоторые рекомендации сообщества по поводу использования столбцов, семейств столбцов и таблиц.

День 2: домашнее задание

Информационный поиск

1. Найдите статью или тему в форуме, где обсуждаются плюсы и минусы сжатия в HBase.
2. Найдите статью, в которой объясняются общие принципы работы фильтра Блума и достоинства их применения в HBase.
3. Если не считать алгоритма, то какие еще параметры семейства столбцов относятся к сжатию?
4. Как тип данных и ожидаемые способы использования влияют на параметры сжатия семейства столбцов?

Задачи

Развивая идею импорта данных, давайте создадим базу данных, содержащую сведения о продуктах питания.

Скачайте набор данных MyPyramid Raw Food Data с сайта Data.gov⁶. Распакуйте архив и найдите в нем файл `Food_Display_Table.xml`.

Этот набор состоит из тегов `<Food_Display_Row>`. Внутри каждого такого тега имеются теги `<Food_Code>` (целое число), `<Display_Name>` (строка) и другие сведения о продукте питания в тегах с соответствующими именами.

1. Создайте новую таблицу `foods` с одним семейством столбцов для хранения сведений о продуктах питания. Как следует выбрать ключ строки? Какие параметры имеет смысл задать для этого семейства столбцов?
2. Напишите на JRuby скрипт для импорта данных о продуктах питания. Примените такой же способ потокового разбора с помощью SAX-анализатора, как при импорте данных из википедии, адаптировав его к другим данным.
3. С помощью конвейера подайте данные о продуктах питания на вход скрипта, чтобы заполнить таблицу.
4. С помощью оболочки HBase запросите из таблицы `foods` информацию о своих любимых продуктах.

⁶ <http://explore.data.gov/Health-and-Nutrition/MyPyramid-Food-Raw-Data/b978-7txq>

4.4. День 3: переходим в облако

В первый и во второй день мы много работали с HBase в автономном режиме. Но до сих пор наш опыт ограничивался только одним локальным сервером. На практике же, выбирая HBase, вы, наверное, захотите организовать кластер приличного размера, чтобы в полной мере задействовать все преимущества распределенной архитектуры.

Сегодня в фокусе нашего внимания будет эксплуатация удаленного кластера HBase. Сначала мы разработаем клиентское приложение на Ruby и подключимся к локальному серверу по двоичному протоколу Thrift. Затем мы поднимем кластер с несколькими узлами на платформе известного поставщика облачных служб – Amazon EC2 – применив технологию управления кластером Apache Whirr.

Разработка «бережливого» приложения для HBase⁷

До сих пор мы работали с оболочкой HBase, но HBase поддерживает и ряд других протоколов подключения к серверу. Ниже приведен полный перечень.

Название	Способ подключения	Готовность к промышленной эксплуатации
Оболочка	Прямой	Да
Java API	Прямой	Да
Thrift	Двоичный протокол	Да
REST	HTTP	Да
Avro	Двоичный протокол	Нет (в стадии эксперимента)

В таблице выше указано, как производится обращение к написанным на Java методам API: напрямую, поверх протокола HTTP или по компактному двоичному протоколу. Все способы подключения, кроме Avro, прошли тестирование и готовы к промышленной эксплуатации; Avro – сравнительно новый протокол, который пока считается экспериментальным.

Из всех вариантов Thrift, пожалуй, наиболее популярен для разработки клиентских приложений. Это зрелый двоичный протокол с минимальными накладными расходами. Первоначально он был разработан Facebook и распространялся в исходных кодах, позже полу-

⁷ Thrift по-английски означает «бережливость». *Прим. перев.*

чил статус инкубационного проекта Apache. Итак, подготовьте свою машину к подключению по протоколу Thrift.

Установка Thrift

Как часто бывает с базами данных, для работы с протоколом Thrift необходима предварительная настройка. Чтобы подключиться к серверу HBase по протоколу Thrift, необходимо:

1. Настроить HBase так, чтобы запускалась служба Thrift.
2. Установить командную утилиту Thrift.
3. Установить библиотеки для выбранного языка программирования клиентов.
4. Сгенерировать файлы модели HBase для выбранного языка.
5. Написать и запустить клиентское приложение.

Начнем с запуска службы Thrift, так как это самое простое. Из командной строки демон запускается следующим образом:

```
{HBASE_HOME}/bin/hbase-daemon.sh start thrift -b 127.0.0.1
```

Далее необходимо установить командную утилиту `thrift`. Конкретные шаги зависят от среды и, вообще говоря, включают компиляцию исходного кода. Чтобы проверить правильность установки, выполните команду с флагом `-version`. Должна быть напечатана примерно такая строка:

```
$ thrift -version
Thrift version 0.6.0
```

В качестве клиентского языка мы выберем Ruby, но для других языков действия аналогичны. Установите `gem`-пакет Thrift, выполнив следующую команду:

```
$ gem install thrift
```

Чтобы проверить правильность установки `gem`-пакета, выполните однострочный скрипт:

```
$ ruby -e "require 'thrift'"
```

Если на консоли ничего не напечатано, значит, всё прекрасно! Увидев сообщение об ошибке типа `no such file to load`, остановитесь и разберитесь в причинах.

Генерация моделей

Следующий шаг – генерация зависящих от языка моделей для HBase. Модельные файлы служат прослойкой между установленны-

ми версиями HBase и Thrift, поэтому они генерируются, а не поставляются в готовом виде.

Для начала найдите файл `hbase.thrift` в каталоге `${HBASE_HOME}/src`. Путь к нему выглядит примерно так:

```
${HBASE_HOME}/src/main/resources/org/apache/hadoop/hbase/thrift/Hbase.thrift
```

Теперь сгенерируйте модельные файлы, подставив в следующую команду определенный выше путь:

```
$ thrift --gen rb <path_to_Hbase.thrift>
```

В результате будет создан новый каталог `gen-rb`, содержащий следующие модельные файлы:

- `hbase_constants.rb`;
- `hbase.rb`;
- `hbase_types.rb`.

Далее мы будем использовать эти файлы в клиентском приложении.

Разработка клиентского приложения

Наша программа будет подключаться к HBase по протоколу Thrift и выводить список найденных таблиц вместе с семействами столбцов. Это можно рассматривать как первый шаг на пути создания административного интерфейса HBase. В отличие от предыдущих примеров этот скрипт написан на чистом Ruby, а не на JRuby. Его вполне можно включить в веб-приложение, написанное на Ruby.

Введите следующий текстовый файл (мы назвали его `thrift_example.rb`):

hbase/thrift_example.rb

```
$.push('./gen-rb')
require 'thrift'
require 'hbase'

socket = Thrift::Socket.new( 'localhost', 9090 )
transport = Thrift::BufferedTransport.new( socket )
protocol = Thrift::BinaryProtocol.new( transport )
client = Apache::Hadoop::Hbase::Thrift::Hbase::Client.new( protocol )

transport.open()

client.getTableNames().sort.each do |table|
  puts "#{table}"
```

```

client.getColumnDescriptors( table ).each do |col, desc|
  puts " #{desc.name}"
  puts " maxVersions: #{desc.maxVersions}"
  puts " compression: #{desc.compression}"
  puts " bloomFilterType: #{desc.bloomFilterType}"
end
end

transport.close()

```

Здесь мы сначала настраиваем среду так, чтобы Ruby мог найти модельные файлы, для чего добавляем в путь каталог `gen-rb`, а затем включаем файлы `thrift` и `hbase`. Далее мы создаем соединение с сервером Thrift и связываем его с экземпляром клиента HBase. Объект `client` понадобится для коммуникации с HBase.

Открыв объект `transport`, мы перебираем все таблицы, возвращенные методом `getTableNames()`. Для каждой таблицы мы обходим список семейств столбцов, возвращенный методом `getColumnDescriptors()`, и распечатываем некоторые свойства на стандартный вывод.

Теперь выполним эту программу из командной строки. На вашей машине должен получиться похожий результат, поскольку мы подключаемся к ранее запущенному локальному серверу HBase.

```

$> ruby thrift_example.rb
links
  from:
    maxVersions: 1
    compression: NONE
    bloomFilterType: ROWCOL
  to:
    maxVersions: 1
    compression: NONE
    bloomFilterType: ROWCOL
wiki
  revision:
    maxVersions: 2147483647
    compression: NONE
    bloomFilterType: NONE
  text:
    maxVersions: 2147483647
    compression: GZ
    bloomFilterType: ROW

```

Легко видеть, что Thrift API для HBase обладает в основном той же функциональностью, что и рассмотренный выше Java API, но многие вещи выражаются по-другому. Например, в Thrift вместо объекта

Put создается объект *Mutation* для обновления одного столбца или объект *BatchMutation*, когда в одной транзакции требуется обновить несколько столбцов.

Файл *Hbase.thrift*, который мы использовали для генерации модельных файлов, хорошо документирован в комментариях, где описаны все доступные структуры и методы. Убедитесь сами!

Введение в Whirr

Раньше настройка работающего облачного кластера требовала *значительных* усилий. К счастью, с появлением Whirr все изменилось. Whirr, который в настоящее время имеет статус инкубационного проекта Apache, предоставляет средства для запуска кластера виртуальных машин, подключения к нему и последующего уничтожения. Он поддерживает такие популярные службы, как Elastic Compute Cloud (EC2) компании Amazon и Cloud Servers компании RackSpace. Whirr может использоваться для настройки кластеров Hadoop, HBase, Cassandra, Voldemort и ZooKeeper, ведутся также разработки для поддержки других технологий, например MongoDB и ElasticSearch.

Хотя поставщики облачных служб, в частности Amazon, нередко предлагают средства для сохранения данных после уничтожения виртуальных машин, мы ими пользоваться не будем. Для наших целей вполне достаточно временного кластера, данные которого полностью теряются после завершения работы с ним. Но если впоследствии вы решите использовать HBase в промышленном режиме, то, конечно, надо будет позаботиться о постоянном хранилище. И тогда имеет смысл подумать – быть может, для вас больше подойдет выделенное оборудование. Динамические службы типа EC2 – отличное решение, когда нужно оперативно получить вычислительные мощности, но, вообще говоря, от кластера выделенных физических или виртуальных машин можно получить большую отдачу.

Подготовка к работе с EC2

Прежде чем приступить к настройке кластера с помощью Whirr, необходимо завести учетную запись у одного из поддерживаемых поставщиков облачных служб. В этой главе мы расскажем об Amazon EC2, но никто не мешает вам выбрать другого поставщика.

Если у вас еще нет учетной записи в Amazon, создайте ее на портале Amazon Web Services (AWS)⁸. Зайдите в свою учетную запись и

8 <http://aws.amazon.com/>

активируйте службу EC2, если она еще не активна⁹. Затем откройте страницу консоли EC2 AWS¹⁰ (см. рис. 17).

Для запуска узлов EC2 вам понадобятся учетные данные AWS. Перейдите на главную страницу AWS и выберите пункт Account→Security Credentials (Учетная запись→Учетные данные). Найдите раздел Access Credentials (Учетные данные для доступа) и запишите куда-нибудь свой идентификатор ключа доступа (Access Key ID). Нажмите кнопку Show (Показать) под надписью Secret Access Key (Секретный ключ доступа) и запишите выведенное значение. Далее во время настройки Whirr мы будем называть эти параметры AWS_ACCESS_KEY_ID и AWS_SECRET_ACCESS_KEY соответственно.

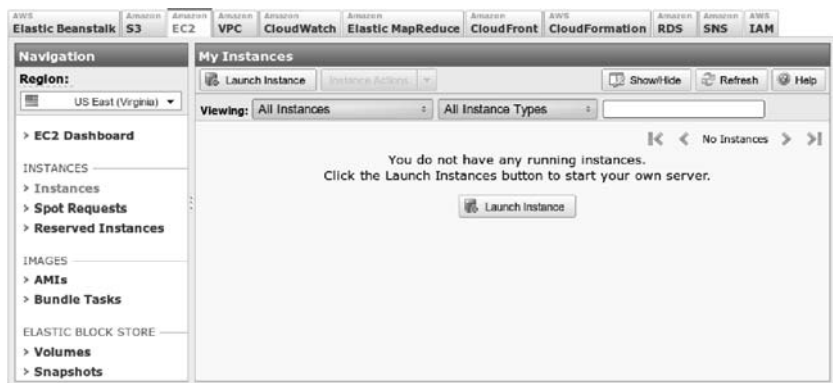


Рис. 17. Консоль Amazon EC2 – экземпляров пока нет

Подготовка Whirr

Получив учетные данные EC2, займемся Whirr. Перейдите на сайт Apache Whirr¹¹ и скачайте последнюю версию. Распакуйте скачанный архив и в том же каталоге откройте окно команд. Для проверки работоспособности Whirr выполните команду `version`.

```
$ bin/whirr version
Apache Whirr 0.6.0-incubating
```

Далее мы создадим не защищенные парольной фразой ключи SSH, которые Whirr будет использовать при запуске экземпляров (виртуальных машин).

⁹ <http://aws.amazon.com/ec2/>

¹⁰ <https://console.aws.amazon.com/ec2/#s=Instances>

¹¹ <http://incubator.apache.org/whirr/>

```
$ mkdir keys
$ ssh-keygen -t rsa -P '' -f keys/id_rsa
```

В результате будет создан каталог `keys`, а в нем – файлы `id_rsa` и `id_rsa.pub`. Покончив с этим, можно приступить к настройке кластера.

Настройка кластера

Для задания параметров кластера Whirr необходим файл с расширением `.properties`. Создайте в каталоге, где находится Whirr, файл `hbase.properties` с таким содержимым (вместо `AWS_ACCESS_KEY_ID` и `AWS_SECRET_ACCESS_KEY` подставьте полученные от Amazon значения):

hbase/hbase.properties

```
# поставщик служб
whirr.provider=aws-ec2
whirr.identity=your AWS_ACCESS_KEY_ID here
whirr.credential=your AWS_SECRET_ACCESS_KEY here

# учетные данные для ssh
whirr.private-key-file=keys/id_rsa
whirr.public-key-file=keys/id_rsa.pub

# конфигурация кластера
whirr.cluster-name=myhbasecluster
whirr.instance-templates=\
  1 zookeeper+hadoop-namenode+hadoop-jobtracker+hbase-master,\
  5 hadoop-datanode+hadoop-tasktracker+hbase-regionserver

# Задание версий HBase и Hadoop
whirr.hbase.tarball.url=\
  http://apache.cu.be/hbase/hbase-0.90.3/hbase-0.90.3.tar.gz
whirr.hadoop.tarball.url=\
  http://archive.cloudera.com/cdh/3/hadoop-0.20.2-cdh3u1.tar.gz
```

В двух первых разделах определяется поставщик служб и все необходимые учетные данные (эта часть общая для любых кластеров), а в двух последних – параметры создаваемого кластера HBase. Свойство `whirr.cluster-name` не имеет значения, если вы не собираетесь запускать одновременно несколько кластеров, а в таком случае они должны иметь разные имена. Свойство `whirr.instance-templates` содержит перечисленные через запятую роли узлов с указанием количества узлов в каждой роли. В данном случае мы хотим иметь один главный и пять региональных серверов. Наконец, свойство `whirr.`

`hbase.tarball.url` инструктирует Whirr, что следует использовать ту версию HBase, с которой мы работаем в этой книге.

Запуск кластера

Сохранив конфигурационные параметры в файле `hbase.properties`, мы можем запустить кластер. Находясь в каталоге Whirr, выполните команду `launchcluster`, указав имя созданного выше файла свойств.

```
$ bin/whirr launch-cluster --config hbase.properties
```

Команда работает сравнительно долго и выводит на экран много информации. Следить за ходом запуска кластера можно на консоли AWS EC2, ее вид показан на рис. 18.

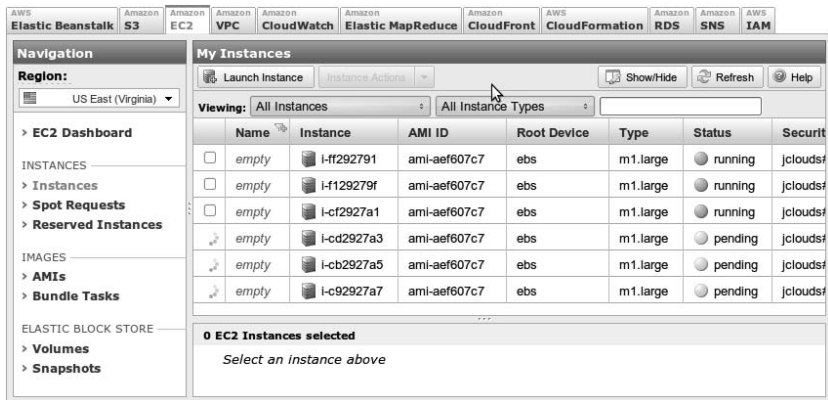


Рис. 18. Консоль Amazon EC2 – показан ход запуска экземпляров HBase

Дополнительные сведения о состоянии запуска можно найти в файле `whirr.log` в каталоге Whirr.

Подключение к кластеру

По умолчанию любой трафик с кластером должен быть защищен, поэтому для подключения к HBase необходимо открыть SSH-сеанс. Прежде всего, нам необходимо знать имя того сервера в кластере, к которому мы подключаемся. В нашем домашнем каталоге Whirr создал каталог `.whirr/myhbasecluster`. В нем вы найдете файл `instances`, в котором перечислены работающие в кластере экземпляры Amazon. В третьем столбце находятся доступные извне доменные имена сер-

веров. Подставьте в показанную ниже командную строку имя первого же сервера.

```
$ ssh -i keys/id_rsa ${USER}@<SERVER_NAME>
```

После успешного подключения запустите оболочку HBase:

```
$ /usr/local/hbase-0.90.3/bin/hbase shell
```

В оболочке можно опросить состояние кластера с помощью команды `status`.

```
hbase> status
6 servers, 0 dead, 2.0000 average load
```

Начиная с этого момента, можно выполнять все операции, которые мы рассматривали в предыдущие дни, в частности создание таблиц и вставку данных. Подключение клиентского приложения на базе протокола Thrift мы оставим в качестве упражнения читателю. Но перед тем как закончить, мы должны рассмотреть еще одно действие – уничтожение кластера.

Уничтожение кластера

Закончив работу с удаленным кластером HBase в облаке EC2, остановите его с помощью команды `Whirr destroycluster`. Отметим, что при этом будут потеряны все введенные в кластер данные, так как мы не указали, что сконфигурированные экземпляры должны использовать постоянное хранилище.

Находясь в каталоге Whirr, выполните следующую команду:

```
$ bin/whirr destroy-cluster --config hbase.properties
Destroying myhbasecluster cluster
Cluster myhbasecluster destroyed
```

Останов производится довольно быстро. Зайдя на консоль AWS (рис. 19), убедитесь, что экземпляры действительно останавливаются.

Если во время выполнения операции что-то пойдет не так, вы всегда сможете остановить кластер непосредственно на консоли AWS.

День 3: итоги

Сегодня мы изучили отличные от оболочки HBase способы подключения, в том числе двоичный протокол Thrift. Мы разработали клиентское приложение на базе Thrift, а затем создали и настроили удаленный кластер в облаке Amazon EC2 с помощью Apache Whirr.

AWS Elastic Beanstalk | Amazon S3 | Amazon EC2 | Amazon VPC | Amazon CloudWatch | Amazon Elastic MapReduce | Amazon CloudFront | AWS CloudFormation | Amazon RDS | Amazon SNS | AWS IAM

Navigation
Region: US East (Virginia) ▾

[EC2 Dashboard](#)

INSTANCES
[Instances](#)
[Spot Requests](#)
[Reserved Instances](#)

IMAGES
[AMIs](#)
[Bundle Tasks](#)

My Instances
 Launch Instance Instance Actions Show/Hide Refresh Help
Viewing: All Instances All Instance Types

	Name ↗	Instance	AMI ID	Root Device	Type	Status	Security Groups
🔍	empty	i-hf292791	ami-aef607c7	ebs	m1.large	shutting-down	jclouds#mytha
<input type="checkbox"/>	empty	i-lf29279f	ami-aef607c7	ebs	m1.large	terminated	jclouds#mytha
<input type="checkbox"/>	empty	i-cl2927a1	ami-aef607c7	ebs	m1.large	terminated	jclouds#mytha
🔍	empty	i-cd2927a3	ami-aef607c7	ebs	m1.large	shutting-down	jclouds#mytha
🔍	empty	i-cb2927a5	ami-aef607c7	ebs	m1.large	shutting-down	jclouds#mytha
🔍	empty	i-c92927a7	ami-aef607c7	ebs	m1.large	shutting-down	jclouds#mytha

0 EC2 Instances selected

Select an instance above

День 3: домашнее задание

Задачи

3. На веб-консоли Amazon EC2 откройте порт TCP 9090 в группе безопасности для своего кластера (Network & Security > Security Groups > Inbound > Create a new rule).

4. Измените разработанное нами клиентское приложение на базе протокола Thrift, так чтобы оно обращалось к узлу EC2, а не к localhost. Запустите программу и убедитесь, что она отображает правильную информацию о вновь созданной таблице.

4.5. Резюме

HBase – это сочетание простоты и сложности. Сама модель хранения данных относительно проста, в схему встраиваются лишь немногие ограничения. Однако при изучении системы очень мешает, что терминология заимствована из мира реляционных баз данных (например, это относится к словам *таблица* и *столбец*). При проектировании схемы HBase во главу угла ставится производительность таблиц и столбцов.

Сильные стороны HBase

Среди особенностей HBase особого внимания заслуживает архитектура горизонтальной масштабируемости и встроенные средства версионирования и сжатия. Для некоторых задач встроенное версионирование – чрезвычайно полезная функция. Например, хранение истории версий страниц вики-сайта необходимо для выработки политик и обслуживания. HBase позволяет не предпринимать никаких специальных мер для реализации истории страниц – мы получаем это бесплатно.

Если говорить о производительности, то HBase изначально задумана для масштабирования по горизонтали. Если объем ваших данных измеряется гигабайтами или терабайтами, то HBase – как раз то, что надо. В HBase имеются механизмы учета конкретных серверных стоек; репликация данных между узлами, установленными в одной или в разных стойках ЦОД, обеспечивает быструю отработку отказов без заметного снижения функциональности.

Сообщество HBase впечатляет. В IRC-канале¹² или в списках рассылки¹³ почти всегда есть кто-то, готовый ответить на вопрос и указать нужное направление. HBase используют многие известные компании, но организации, которая поддерживала бы HBase на коммерческой основе, нет. Иными словами, люди из сообщества HBase делают это исключительно из любви к проекту и общему благу.

¹² [irc://irc.freenode.net/#hbase](http://irc.freenode.net/#hbase)

¹³ <http://hbase.apache.org/mail-lists.html>

Слабые стороны HBase

Хотя HBase проектировалась с учетом горизонтальной масштабируемости, существует нижний порог числа узлов. Сообщество HBase согласно в том, что для надежной работы необходимо не менее пяти узлов. Поскольку система предназначена для обработки больших массивов данных, администрировать ее сравнительно трудно. Для решения небольших задач HBase вряд ли годится, а документации для неспециалистов практически не существует, что усложняет изучение.

Кроме того, HBase почти никогда не разворачивается в изоляции. Она часть экосистемы масштабируемых компонентов, в число которых входит Hadoop (независимая реализация каркаса MapReduce, впервые предложенного Google), распределенная файловая система Hadoop (HDFS) и Zookeeper (служба без главного сервера, координирующая работу узлов). У этой экосистемы имеются как плюсы, так и минусы; она обеспечивает высокую стабильность, но возлагает на администратора груз ответственности за обслуживание.

Следует отметить, что HBase не предоставляет никаких средств сортировки или индексирования, помимо ключей строк. Строки хранятся отсортированными по ключам, но ни по какому другому полю, скажем по имени и значению столбца, сортировка не производится. Поэтому для поиска строки по любому критерию, кроме ее ключа, придется либо сканировать всю таблицу, либо строить и сопровождать собственный индекс.

Отсутствует также понятие типа данных. Значения всех полей в HBase трактуются как неинтерпретируемые массивы байтов. Нет никакого различия между целым числом, строкой и датой. Для HBase всё это просто байты, а их интерпретация возлагается на приложение.

HBase и теорема CAP

В терминологии CAP HBase, безусловно, относится к классу CP (согласованная и устойчивая к потере связности). HBase дает строгие гарантии согласованности. Если один клиент успешно записал какое-то значение, то все остальные клиенты увидят его при следующем запросе. Некоторые базы данных, например Riak, позволяют изменять параметры CAP на уровне отдельной операции. Но только не HBase. При не слишком серьезной потере связности, например при выходе из строя одного узла, HBase останется доступной – ответственность

за обслуживание запросов будет передана другим узлам. Но в патологической ситуации, например, когда работоспособность сохранил всего один узел, у HBase не будет другого выбора, как отказать в обслуживании запроса.

Обсуждение свойств CAP несколько усложняется, если принять во внимание межкластерную репликацию, которую мы в этой главе не рассматривали. В типичной многокластерной конфигурации кластеры могут быть территориально разнесены. В таком случае для любого заданного семейства столбцов запись производится только на один кластер, а остальные просто предоставляют доступ к реплицированным данным. Такая система является *согласованной в конечном счете*, поскольку реплицированные кластеры возвращают последнее значение, о котором знают, а оно может и не совпадать с последним значением, записанным на главном кластере.

Перед расставанием

HBase – одна из первых рассмотренных нами нереляционных баз данных, и, конечно, не обошлось без сложностей. Терминология может показаться обманчиво знакомой, а установка и настройка – занятие не для слабых духом. С другой стороны, некоторые возможности HBase, в частности версионирование и сжатие, поистине уникальны и могут сделать HBase весьма привлекательным выбором для решения некоторых задач. Кроме того, нельзя не отметить возможность горизонтального масштабирования на узлы со стандартным оборудованием. В общем и целом, HBase – как и гвоздепистолет – серьезный инструмент, так что следите, чтобы не поранить пальцы.



ГЛАВА 5.

MongoDB

MongoDB – во многих отношениях можно уподобить профессиональной электродрели. Пригодность для той или иной работы зависит в основном от выбранных компонентов (от размера сверла до шлифовальной насадки). То же можно сказать и о достоинствах MongoDB – гибкость, мощь, простота использования и применимость для самых разных задач, больших и малых. Хотя изобретена она куда позже молотка, но постепенно становится инструментом, к которому рука рабочего тянется всё чаще.

Первая публичная версия MongoDB была выпущена в 2009 году, а теперь это восходящая звезда в мире NoSQL. Система задумывалась как масштабируемая база данных – название Mongo происходит от слова «humongous», получившегося объединением «huge» (гигантский) и «monstrous» (чудовищный), а в качестве основных проектных целей были поставлены высокая производительность и простота доступа к данным. Это документная база данных, которая позволяет не только хранить, но и опрашивать вложенные данные, предъявляя произвольные запросы. Схема базы данных не навязывается (в этом MongoDB похожа на Riak, но отличается от Postgres), поэтому один документ может содержать поля или типы, отсутствующие во всех остальных документах коллекции. Но не думайте, что гибкость MongoDB превращает ее в игрушку. Эту базу данных используют такие гигантские сайты, как Foursquare и bit.ly, а в Европейском центре ядерных исследований (ЦЕРН) она применяется для хранения данных, поступающих с большого адронного коллайдера.

5.1. Монстр

Mongo счастливо сочетает в себе мощные средства запросов, характерные для реляционных баз данных, и распределенную архитектуру, свойственную таким хранилищам, как Riak или HBase.

Основатель проекта Дуайт Мерримэн (Dwight Merriman) говорит, что MongoDB – это та база данных, с которой он хотел бы работать в компании DoubleClick, где занимал должность технического директора и отвечал за построение крупномасштабного хранилища данных, которое тем не менее могло бы отвечать на произвольные запросы.

Mongo – хранилище JSON-документов (хотя, строго говоря, данные хранятся в двоичном варианте JSON, который называется BSON). Документ Mongo можно уподобить строке реляционной таблицы без схемы, в которой допускается произвольная глубина вложенности значений. Рис. 20 поможет понять, как выглядит JSON-документ.

```
> printjson( db.towns.findOne({"_id" : ObjectId("4d0b6da3bb30773266f39fea")}))
{
  "_id" : ObjectId("4d0b6da3bb30773266f39fea"),
  "country" : {
    "$ref" : "countries",
    "$id" : ObjectId("4d0e6074deb8995216a8309e")
  },
  "famous_for" : [
    "beer",
    "food"
  ],
  "last_census" : "Thu Sep 20 2007 00:00:00 GMT-0700 (PDT)",
  "mayor" : {
    "name" : "Sam Adams",
    "party" : "D"
  },
  "name" : "Portland",
  "population" : 582000,
  "state" : "OR"
}
```

Коллекция

База данных

Идентификатор

Документ

Рис. 20. Документ Mongo, представленный в формате JSON

Mongo – отличный выбор для растущего класса веб-проектов, в которых необходимо работать с большими массивами данных, но бюджет слишком мал для приобретения дорогостоящего оборудования. Благодаря отсутствию структурированной схемы, Mongo может расти и изменяться вместе с моделью данных. Если вы работаете в недавно образованной компании, которая лелеет грандиозные планы или уже накопила столько данных, что возникла потребность в горизонтальном масштабировании, то присмотритесь к MongoDB.

5.2. День 1: операции CRUD и вложенность

Сегодняшний день мы посвятим некоторым операциям CRUD и завершим выполнением запросов к вложенным данным. Как обычно, шаги установки мы опустим, но с сайта Mongo¹ вы можете скачать готовый дистрибутив для своей ОС и найти инструкции по сборке из исходного кода. Если вы работаете с OS X, рекомендуем производить установку с помощью программы Homebrew (`brew install mongod`). А пользователи Debian/Ubuntu Linux могут скачать пакет для `apt-get`, собранный самой компанией MongoDB.org.

Говорит Эрик: оглядываясь по сторонам

Я посматривал в разные стороны в поисках документного хранилища данных, на которое мог бы переключиться в своем коде. Имея опыт работы с реляционными СУБД, я считал, что перейти на Mongo с ее поддержкой произвольных запросов будет проще всего. А возможность горизонтального масштабирования отвечала моим мечтам о проекте масштаба веб. Но даже если отвлечься от технических деталей, я чувствовал доверие к команде разработчиков. Они с готовностью признавали, что Mongo не идеальна, но их планы (которых они, в общем-то, придерживались) основывались на реальных примерах инфраструктуры в веб, а не на идилических рассуждениях о масштабируемости и репликации. Такая прагматическая сосредоточенность на удобстве работы с системой становится очевидной сразу, как только начинаешь использовать MongoDB. В силу эволюционного характера развития в Mongo есть несколько способов выполнить любую поставленную задачу.

Во избежание опечаток Mongo требует, чтобы вы заранее создали каталог, где сервер `mongod` будет хранить данные. Обычно выбирают каталог `/data/db`. Убедитесь, что пользователь, от имени которого работает сервер, имеет право чтения и записи в этот каталог. Если служба Mongo еще не запущена, это можно сделать, выполнив команду `mongod`.

Поработаем с командной строкой

Чтобы создать новую базу данных `book`, выполните в окне терминала показанную ниже команду. Она запускает интерактивный интерфейс, устроенный по образцу MySQL.

```
$ mongo book
```

¹ <http://www.mongodb.org/downloads>

Для начала наберите слово `help`. Сейчас текущей является база данных `book`, но команда `show dbs` покажет другие базы, а команда `use` позволит сменить текущую базу.

Для создания коллекции (аналог *сегмента* в терминологии Riak) в Mongo достаточно просто добавить в нее первую запись. Поскольку в Mongo нет схем, то заранее ничего определять не надо. Более того, даже сама база данных `book` физически не существует, пока мы не добавим в нее первый документ. Следующий код создает коллекцию `towns` и вставляет в нее данные:

```
> db.towns.insert({
  name: "New York",
  population: 22200000,
  last_census: ISODate("2009-07-31"),
  famous_for: [ "statue of liberty", "food" ],
  mayor : {
    name : "Michael Bloomberg",
    party : "I"
  }
})
```

Выше мы упоминали, что документы хранятся в формате JSON (точнее, BSON), поэтому в таком формате мы их и добавляем. Фигурные скобки `{...}` обозначают объект (аналог ассоциативного массива или хеш-таблицы), содержащий ключи и значения, а квадратные скобки `[...]` – линейный массив. Значения могут быть вложенными, причем глубина вложенности не ограничена. Команда `show collections` позволит убедиться, что коллекция действительно существует.

```
> show collections
system.indexes
towns
```

Коллекция `towns` создана нами, а коллекция `system.indexes` существует всегда. Просмотреть содержимое коллекции позволяет команда `find()`. Для удобства чтения мы отформатировали вывод, поскольку обычно выводится сплошная строка.

```
> db.towns.find()
{
  "_id" : ObjectId("4d0ad975bb30773266f39fe3"),
  "name" : "New York",
  "population": 22200000,
  "last_census": "Fri Jul 31 2009 00:00:00 GMT-0700 (PDT)",
  "famous_for" : [ "statue of liberty", "food" ],
  "mayor" : { "name" : "Michael Bloomberg", "party" : "I" }
}
```

В отличие от реляционных СУБД, Mongo не поддерживает соединения на стороне сервера. Один вызов JavaScript-функции извлекает документ *и* все вложенные в него данные.

Возможно, вы обратили внимание на добавленное системой поле `_id` типа `ObjectId`. Это близкий аналог ключевого слова `SERIAL`, которое в PostgreSQL служит для автоматического инкремента числового первичного ключа. Объект `ObjectId` всегда занимает 12 байтов и состоит из временной метки, идентификатора клиентской машины, идентификатора клиентского процесса и 3-байтового инкрементируемого счетчика. Структура этого объекта показана на рис. 21.

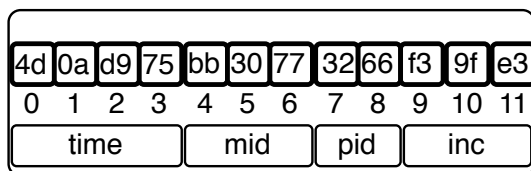


Рис. 21. Структура `ObjectId`

У этой схемы автоматической нумерации есть одно несомненное достоинство: любой процесс на любой машине сам отвечает за генерацию идентификаторов, не вступая в конфликт с другими экземплярами `mongod`. Это проектное решение – прямой намек на распределенную природу.

JavaScript

Родным языком Mongo является JavaScript, который применяется и в сложных MapReduce-запросах, и для простейшего получения справки:

```
> db.help()
> db.towns.help()
```

Эти команды выводят список доступных для данного объекта функций. `db` представляет собой JavaScript-объект, который содержит информацию о текущей базе данных. `db.x` – JavaScript-объект, представляющий коллекцию с именем `x`. Сами команды – обычные JavaScript-функции.

```
> typeof db
object
> typeof db.towns
object
```

```
> typeof db.towns.insert
function
```

Чтобы ознакомиться с исходным кодом функции, вызовите ее без параметров и без скобок (это больше походит на Python, чем на Ruby).

```
db.towns.insert
function (obj, _allow_dot) {
  if (!obj) {
    throw "no object passed to insert!";
  }
  if (!_allow_dot) {
    this._validateForStorage(obj);
  }
  if (typeof obj._id == "undefined") {
    var tmp = obj;
    obj = {_id:new ObjectId};
    for (var key in tmp) {
      obj[key] = tmp[key];
    }
  }
  this._mongo.insert(this._fullName, obj);
  this._lastID = obj._id;
}
```

Давайте добавим еще несколько документов в коллекцию towns, для чего напомним собственную JavaScript-функцию.

mongo/insert_city.js

```
function insertCity(
  name, population, last_census,
  famous_for, mayor_info
) {
  db.towns.insert({
    name:name,
    population:population,
    last_census: ISODate(last_census),
    famous_for:famous_for,
    mayor : mayor_info
  });
}
```

Можете просто скопировать этот код в оболочку, а затем вызвать его.

```
insertCity("Punxsutawney", 6200, '2008-31-01',
  ["phil the groundhog"], { name : "Jim Wehrle" }
)
```

```
insertCity("Portland", 582000, '2007-20-09',
  ["beer", "food"], { name : "Sam Adams", party : "D" }
)
```

Теперь в коллекции `towns` должно быть три города, в чем можно легко убедиться, вызвав функцию `db.towns.find()`, как и раньше.

Чтение: продолжаем изучать Mongo

Ранее мы вызывали функцию `find()` без параметров, чтобы получить все документы. Для получения конкретного документа достаточно задать свойство `_id`. Поскольку `_id` имеет тип `ObjectId`, то для формулирования запроса необходимо преобразовать строку к этому типу, обернув ее функцией `ObjectId(str)`.

```
db.towns.find({ "_id" : ObjectId("4d0ada1fbb30773266f39fe4") })
{
  "_id" : ObjectId("4d0ada1fbb30773266f39fe4"),
  "name" : "Punxsutawney",
  "population" : 6200,
  "last_census" : "Thu Jan 31 2008 00:00:00 GMT-0800 (PST)",
  "famous_for" : [ "phil the groundhog" ],
  "mayor" : { "name" : "Jim Wehrle" }
}
```

Функция `find()` принимает и еще один параметр: объект, позволяющий указать, какие поля извлекать из базы. Если вас интересует только название города (и его `_id`), передайте объект, в котором свойство `name` принимает значение `1` (или `true`).

```
db.towns.find({ _id : ObjectId("4d0ada1fbb30773266f39fe4") }, { name : 1 })
{
  "_id" : ObjectId("4d0ada1fbb30773266f39fe4"),
  "name" : "Punxsutawney"
}
```

Чтобы извлечь все поля, *кроме* `name`, задайте свойство `name` равным `0` (или `false`, или `null`).

```
db.towns.find({ _id : ObjectId("4d0ada1fbb30773266f39fe4") }, { name : 0 })
{
  "_id" : ObjectId("4d0ada1fbb30773266f39fe4"),
  "population" : 6200,
  "last_census" : "Thu Jan 31 2008 00:00:00 GMT-0800 (PST)",
  "famous_for" : [ "phil the groundhog" ]
}
```

Как и PostgreSQL, Mongo позволяет предъявлять произвольные запросы по значениям полей, диапазонам или с несколькими крите-

риями. Чтобы найти все города, названия которых начинаются с буквы *P*, а численность населения меньше 10 000, можно воспользоваться совместимым с Perl синтаксисом регулярных выражений (PCRE)² и оператором диапазона.

```
db.towns.find(
  { name : /^P/, population : { $lt : 10000 } },
  { name : 1, population : 1 }
){
  "name" : "Punxsutawney", "population" : 6200 }
```

Условные операторы в Mongo записываются в виде `field : { $op : value }`, где `$op` – операция, например `$ne` (не равно). Хотелось бы, чтобы синтаксис был покороче, скажем `field < value`. Но это код на JavaScript, а не предметно-ориентированный язык запросов, поэтому запросы должны подчиняться синтаксическим правилам JavaScript (ниже мы увидим, что в некоторых случаях краткий синтаксис все же допустим, но пока не будем об этом).

Но есть и хорошие новости: поскольку язык запросов – это JavaScript, то операции можно конструировать, как объекты. Ниже конструируется условие на численность населения: от 10 000 до миллиона человек.

```
var population_range = {}
population_range['$lt'] = 1000000
population_range['$gt'] = 10000
db.towns.find(
  { name : /^P/, population : population_range },
  { name: 1 }
)

{ "_id" : ObjectId("4d0ada87bb30773266f39fe5"), "name" : "Portland" }
```

Возможны не только числовые диапазоны, но и диапазоны дат. Найдем все города, в которых дата последнего проведения переписи (*last_census*) меньше или равна 31 января 2008:

```
db.towns.find(
  { last_census : { $lte : ISODate('2008-31-01') } },
  { _id : 0, name: 1 }
)

{ "name" : "Punxsutawney" }
{ "name" : "Portland" }
```

Обратите внимание, что мы подавили включение поля `_id` в результат, явно присвоив соответствующему свойству значение 0.

² <http://www.pcre.org/>

Копнем глубже

Mongo обожает вложенные массивы. В запросе можно задавать сравнение с конкретным значением...

```
db.towns.find(
  { famous_for : 'food' },
  { _id : 0, name : 1, famous_for : 1 }
)

{ "name" : "New York", "famous_for" : [ "statue of liberty", "food" ] }
{ "name" : "Portland", "famous_for" : [ "beer", "food" ] }
```

... или сравнение с подстрокой...

```
db.towns.find(
  { famous_for : /statue/ },
  { _id : 0, name : 1, famous_for : 1 }
)

{ "name" : "New York", "famous_for" : [ "statue of liberty", "food" ] }
```

...или совпадение каждого из нескольких значений...

```
db.towns.find(
  { famous_for : { $all : ['food', 'beer'] } },
  { _id : 0, name:1, famous_for:1 }
)

{ "name" : "Portland", "famous_for" : [ "beer", "food" ] }
```

...или несовпадение ни с одним из указанных значений:

```
db.towns.find(
  { famous_for : { $nin : ['food', 'beer'] } },
  { _id : 0, name : 1, famous_for : 1 }
)

{ "name" : "Punxsutawney", "famous_for" : [ "phil the groundhog" ] }
```

Но истинная мощь Mongo заключается в возможность заглянуть внутрь документа и вернуть результаты поиска глубоко вложенных поддокументов. Чтобы опросить поддокумент, имя поля должно быть строкой, в которой уровни вложенности разделены точками. Например, вот как можно найти города, во главе которых стоят независимые мэры...

```
db.towns.find(
  { 'mayor.party' : 'I' },
  { _id : 0, name : 1, mayor : 1 }
)

{
```

```

    "name" : "New York",
    "mayor" : {
      "name" : "Michael Bloomberg",
      "party" : "I"
    }
  }
}

```

... или беспартийные мэры:

```

db.towns.find(
  { 'mayor.party' : { $exists : false } },
  { _id : 0, name : 1, mayor : 1 }
)

{ "name" : "Punxsutawney", "mayor" : { "name" : "Jim Wehrle" } }

```

Показанные выше запросы годятся, когда нужно найти документы с одним удовлетворяющим критерию полем. Но что если требуется сравнивать несколько полей поддокумента?

Оператор `elemMatch`

Завершив наш экскурс знакомством с оператором `$elemMatch`. Создадим еще одну коллекцию, в которой будем хранить страны. На этот раз переопределим поле `_id`, задав строку по своему выбору:

```

db.countries.insert({
  _id : "us",
  name : "United States",
  exports : {
    foods : [
      { name : "bacon", tasty : true },
      { name : "burgers" }
    ]
  }
})
db.countries.insert({
  _id : "ca",
  name : "Canada",
  exports : {
    foods : [
      { name : "bacon", tasty : false },
      { name : "syrup", tasty : true }
    ]
  }
})
db.countries.insert({
  _id : "mx",
  name : "Mexico",
  exports : {
    foods : [{
      name : "salsa",

```

```
tasty : true,
condiment : true
  ]]
}
})
```

Чтобы проверить, сколько стран было добавлено, мы можем воспользоваться функцией `count`, которая должна вернуть 3.

```
> print( db.countries.count() )
3
```

Давайте поищем страну, которая экспортирует не просто бекон, а *вкусный* (tasty) бекон.

```
db.countries.find(
  { 'exports.foods.name' : 'bacon', 'exports.foods.tasty' : true },
  { _id : 0, name : 1 }
)

{ "name" : "United States" }
{ "name" : "Canada" }
```

Но это не то, что мы хотели. Mongo вернула Канаду (Canada), потому что оттуда экспортируется бекон и вкусный сироп (syrup). На выручку приходит оператор `$elemMatch`. Он означает, что документ (или вложенный поддокумент) считается подходящим, если удовлетворяет *всем* критериям.

```
db.countries.find(
  {
    'exports.foods' : {
      $elemMatch : {
        name : 'bacon',
        tasty : true
      }
    },
    { _id : 0, name : 1 }
  }
)

{ "name" : "United States" }
```

В операторе `$elemMatch` допускаются и более сложные условия. Так, можно найти все страны, которые экспортируют вкусную еду, приправленную к тому же специями (condiment):

```
db.countries.find(
  {
    'exports.foods' : {
      $elemMatch : {
```

```
        tasty : true,
        condiment : { $exists : true }
    }
},
{ _id : 0, name : 1 }
)
{ "name" : "Mexico" }
```

Мексика – как раз то, что мы хотели.

Булевские операции

До сих пор во всех критериях неявно подразумевалась связка *и*. Если запросить страну с названием *United States* и идентификатором *_id*, равным *mx*, то Mongo ничего не найдет.

```
db.countries.find(
  { _id : "mx", name : "United States" },
  { _id : 1 }
)
```

Однако поиск того *или* другого – с помощью оператора *\$or* – вернет два результата. Этот оператор записывается в *префиксной нотации*: *OR A B*.

```
db.countries.find(
  {
    $or : [
      { _id : "mx" },
      { name : "United States" }
    ]
  },
  { _id:1 }
)

{ "_id" : "us" }
{ "_id" : "mx" }
```

Существует еще много других операторов, рассмотреть здесь их все мы не сможем, но надеемся, что вы почувствовали выразительную мощь механизма запросов в MongoDB. Ниже приводится неполный, но достаточно представительный перечень операторов.

Команда	Описание
\$regex	Соответствие строки регулярному выражению, совместимому с синтаксисом PCRE (можно также использовать ограничители <i>//</i> , как было показано выше)

Команда	Описание
\$ne	Не равно
\$lt	Меньше
\$lte	Меньше или равно
\$gt	Больше
\$gte	Больше или равно
\$exists	Проверяет существование поля
\$all	Соответствие всем элементам массива
\$in	Соответствие хотя бы одному элементу массива
\$nin	Несоответствие ни одному элементу массива
\$elemMatch	Соответствие всех полей вложенного документа
\$or	Или
\$nor	Не или
\$size	Соответствие размеру массива
\$mod	Деление по модулю
\$type	Соответствие, если поле имеет указанный тип
\$not	Отрицание

Все операторы описаны в онлайн-документации по MongoDB, а также в шпаргалке, опубликованной на сайте Mongo. Мы еще вернемся к запросам завтра и послезавтра.

Обновление

У нас проблема. Названия Нью-Йорк и Панкстони достаточно уникальны, но как быть с Портлендом? Какой имеется в виду – в штате Орегон или в штате Мэн (или в Техасе, или еще где-нибудь)? Изменим коллекцию towns, добавив штаты.

Функция `update(criteria, operation)` принимает два обязательных параметра. Первый – критерий отбора – такой же, как для функции `find()`. Второй – либо объект, поля которого заменяют поля отобранных документов, либо модификатор. В данном случае модификатор `$set` записывает в поле `state` строку *OR*.

```
db.towns.update(
  { _id : ObjectId("4d0ada87bb30773266f39fe5") },
  { $set : { "state" : "OR" } }
);
```

Может возникнуть вопрос, зачем вообще нужна операция `$set`. Mongo не работает в терминах атрибутов; на внутреннем уровне имеет лишь неявное представление об атрибутах для целей оптимизации. Однако в интерфейсе Mongo никакие *атрибуты* не упоминаются, есть только *документы*. Вряд ли вы когда-нибудь захотите выполнить нечто в таком роде (обратите внимание на отсутствие оператора `$set`):

```
db.towns.update(
{ _id : ObjectId("4d0ada87bb30773266f39fe5") },
{ state : "OR" }
);
```

В этом случае *весь* подходящий документ был бы заменен переданным вами документом (`{ state : "OR" }`). Раз вы не указали команду, например `$set`, Mongo считает, что вы просто хотите целиком заменить документ. Будьте осторожны.

Чтобы проверить, было ли обновление успешным, мы можем поискать документ (обратите внимание на использование функции `findOne()` для поиска только одного подходящего документа).

```
db.towns.findOne({ _id : ObjectId("4d0ada87bb30773266f39fe5") })
{
  "_id" : ObjectId("4d0ada87bb30773266f39fe5"),
  "famous_for" : [
    "beer",
    "food"
  ],
  "last_census" : "Thu Sep 20 2007 00:00:00 GMT-0700 (PDT)",
  "mayor" : {
    "name" : "Sam Adams",
    "party" : "D"
  },
  "name" : "Portland",
  "population" : 582000,
  "state" : "OR"
}
```

К значению можно применять не только оператор `$set`. Часто бывает полезен также оператор `$inc` (увеличить число). Давайте увеличим численность населения Портленда на 1000.

```
db.towns.update(
{ _id : ObjectId("4d0ada87bb30773266f39fe5") },
{ $inc : { population : 1000} }
)
```

Существуют и другие операторы, например, позиционный оператор `$` для массивов. Новые операторы добавляются довольно часто

и описываются в онлайн-о документации. Ниже приведен список наиболее важных операторов.

Команда	Описание
\$set	Записывает указанное значение в указанное поле
\$unset	Удаляет поле
\$inc	Прибавляет указанное число к указанному полю
\$pop	Удаляет последний (или первый) элемент из массива
\$push	Помещает новый элемент в массив
\$pushAll	Помещает все указанные элементы в массив
\$addToSet	Аналогичен push, но дубликаты не добавляются
\$pull	Удаляет из массива подходящее значение, если оно в нем есть
\$pullAll	Удаляет из массива все подходящие значения

Ссылки

Выше мы уже отмечали, что Mongo не предназначена для выполнения соединений. Из-за распределенной природы системы соединение оказалось бы крайне неэффективной операцией. Тем не менее, иногда полезно, чтобы документы могли ссылаться друг на друга. Для таких случаев в Mongo имеется конструкция вида { \$ref : "collection_name", \$id : "reference_id" }. Например, можно добавить в коллекцию towns ссылку на документ из коллекции countries.

```
db.towns.update(
  { _id : ObjectId("4d0ada87bb30773266f39fe5") },
  { $set : { country: { $ref: "countries", $id: "us" } } }
)
```

Теперь можно извлечь из коллекции towns данные о Портленде:

```
var portland = db.towns.findOne(
  { _id : ObjectId("4d0ada87bb30773266f39fe5") })
```

А чтобы получить сведения о стране, в которой находится город, мы можем опросить коллекцию countries, передав сохраненный ранее идентификатор \$id.

```
db.countries.findOne({ _id: portland.country.$id })
```

Можно даже поступить еще лучше: запросить у документа о городе имя коллекции, хранящейся в поле ссылки:


```
db[ portland.country.$ref ].findOne( { _id: portland.country.$id } )
```

Последние два запроса эквивалентны, но второй в большей степени управляется данными.

Удаление

Удалить документы из коллекции несложно. Достаточно вместо функции `find()` воспользоваться функцией `remove()` – и все документы, удовлетворяющие критерию, будут удалены. Важно отметить, что удаляется документ целиком, а не только совпавший элемент или поддокумент.

Об опечатках

Mongo не склонна прощать опечатки. Если вы еще не сталкивались с этой проблемой, то рано или поздно обязательно столкнетесь – будьте внимательны. Тут можно провести параллель со статическими и динамическими языками. В статическом языке типы известны заранее, тогда как динамический готов принять даже значения, которые вы вовсе не имели в виду, например `person_name = 5`.

У документов нет схемы, поэтому Mongo не может знать, действительно ли вы хотели вставить в описание города поле `pipulation` или предъявить запрос по полю `lust_census`; она радостно вставит указанное поле и соответственно не вернет ни одного подходящего документа.

У гибкости есть цена. *Caveat emptor*³.

Мы рекомендуем сначала выполнять `find()`, чтобы убедиться в правильности критерия, а только потом `remove()`. Mongo не станет семь раз отмерять, перед тем как выполнить операцию. Давайте удалим все страны, которые экспортируют невкусный бекон.

```
var bad_bacon = {
  'exports.foods' : {
    $elemMatch : {
      name : 'bacon',
      tasty : false
    }
  }
}
db.countries.find( bad_bacon )

{
  "_id" : ObjectId("4d0b7b84bb30773266f39fef"),
  "name" : "Canada",
  "exports" : {
```

3 Да будет осмотрителен покупатель (лат.) Прим. перев.

```

    "foods" : [
      {
        "name" : "bacon",
        "tasty" : false
      },
      {
        "name" : "syrup",
        "tasty" : true
      }
    ]
  }
}

```

Вроде бы все хорошо. Удаляем.

```

db.countries.remove( bad_bacon )
db.countries.count()
2

```

Теперь выполните команду `count()` и проверьте, что осталось всего две страны. Если так, то удаление прошло успешно!

Функциональные критерии

Этот день мы завершим рассмотрением более интересной возможности предъявления запросов — путем выполнения кода. Можно потребовать, чтобы MongoDB применяла к документам функцию, принимающую решение. Мы оставили этот раздел напоследок, потому что пользоваться описанной здесь возможностью следует только в самом крайнем случае. Такие запросы исполняются очень медленно, они не учитывают имеющихся индексов, и Mongo не в состоянии их оптимизировать. Но иногда ничто не может сравниться с выразительной мощностью специализированного кода.

Предположим, что требуется найти города, в которых проживает от 6000 до 600 000 человек.

```

db.towns.find( function() {
  return this.population > 6000 && this.population < 600000;
} )

```

В Mongo даже имеется сокращенная нотация для простых функций принятия решения:

```

db.towns.find("this.population > 6000 && this.population < 600000")

```

Эту функцию можно применять в сочетании с другими критериями, воспользовавшись фразой `$where`. В следующем примере отбираются также города, славные своими сурками (groundhog):

```
db.towns.find( {  
  $where : "this.population > 6000 && this.population < 600000",  
  famous_for : /groundhog/  
} )
```

Предупреждение: Mongo будет тупо вычислять эту функцию для каждого документа, хотя нет никакой гарантии, что упоминаемое в ней поле вообще существует. Например, если вы предполагаете, что поле *population* существует, а хотя бы в одном документе оно отсутствует, то весь запрос завершится ошибкой, потому что написанный на JavaScript код невозможно будет правильно выполнить. Будьте внимательны при написании собственных функций принятия решений и постарайтесь обойтись стандартными средствами.

День 1: итоги

Сегодня мы познакомились с первой в этой книге документной базой данных, MongoDB. Мы видели, что она позволяет хранить вложенные документы в виде JSON-объектов и опрашивать их по полям, расположенным на любом уровне вложенности. Вы узнали, что *документ* можно рассматривать как бессхемную строку в реляционной модели со сгенерированным ключом *_id*. Набор документов, который в Mongo называется *коллекцией*, – аналог *таблицы* в PostgreSQL.

В отличие от ранее встречавшихся нам баз данных, в Mongo хранятся не наборы данных простых типов, а сложные денормализованные документы, представленные в виде коллекций произвольных JSON-объектов. При этом Mongo дополняет гибкую стратегию хранения мощным механизмом запросов, не ограниченным предопределенной схемой.

Благодаря денормализации документное хранилище является отличным выбором для хранения данных с заранее неизвестными свойствами, тогда как в других СУБД (реляционных или столбцовых) типы данных нужно знать заранее, а для добавления или изменения полей необходима миграция схемы.

День 1: домашнее задание

Информационный поиск

1. Поставьте закладку на онлайн-документацию по MongoDB.
2. Изучите, как в Mongo конструируются регулярные выражения.



3. Разберитесь, что выдают команды `db.help()` и `db.collections.help()`.
4. Найдите драйвер своего любимого языка программирования для Mongo (Ruby, Java, PHP и т. д.).

Задачи

1. Напечатайте JSON-документ, содержащий `{ "hello" : "world" }`.
2. Найдите города, названия которых содержат слово *new*, применив регулярное выражение без учета регистра.
3. Найдите все города, названия которых содержат букву *e* и славны (*famous_for*) своей едой (*food*) или пивом (*beer*).
4. Создайте новую базу данных *blogger* и в ней коллекцию *articles*. Вставьте новую статью, содержащую имя и электронный почтовый адрес автора, дату создания и текст.
5. Измените статью, добавив массив комментариев; каждый комментарий должен содержать имя автора и текст.
6. Выполните запрос, хранящийся во внешнем JavaScript-файле.

5.3. День 2: индексирование, группировка, mapreduce

Сегодня первым пунктом в нашей повестке дня будет повышение производительности запросов в MongoDB. За ним последуют более сложные запросы с группировкой. И завершим мы день анализом данных с помощью технологии mapreduce по аналогии с тем, что делали при изучении Riak.

Индексирование: когда быстродействия не хватает

Одной из полезных функций Mongo является возможность индексирования для повышения скорости выполнения запросов – вещь, которая, как мы видели, присутствует не во всех NoSQL-базах. MongoDB поддерживает несколько проверенных временем структур индексов, в том числе классическое B-дерево, а также двумерные и сферические пространственные индексы.

Для начала проведем небольшой эксперимент, чтобы оценить эффективность B-деревьев в MongoDB: введем последовательность

телефонных номеров с произвольным префиксом страны (можете подставить префикс своей страны, если хотите). Введите с помощью консоли следующий код, который сгенерирует 100 000 номеров от 1-800-555-0000 до 1-800-565-9999 (это может занять некоторое время).

mongo/populate_phones.js

```
populatePhones = function(area,start,stop) {
  for(var i=start; i < stop; i++) {
    var country = 1 + ((Math.random() * 8) << 0);
    var num = (country * 1e10) + (area * 1e7) + i;
    db.phones.insert({
      _id: num,
      components: {
        country: country,
        area: area,
        prefix: (i * 1e-4) << 0,
        number: i
      },
      display: "+" + country + " " + area + "-" + i
    });
  }
}
```

Выполните эту функцию, задав трехзначный код зоны (например, 800) и диапазон семизначных номеров (от 5550000 до 5650000, при вводе обращайте внимание на количество нулей).

```
populatePhones( 800, 5550000, 5650000 )
db.phones.find().limit(2)

{ "_id" : 18005550000, "components" : { "country" : 1, "area" : 800,
  "prefix" : 555, "number" : 5550000 }, "display" : "+1 800-5550000" }
{ "_id" : 88005550001, "components" : { "country" : 8, "area" : 800,
  "prefix" : 555, "number" : 5550001 }, "display" : "+8 800-5550001" }
```

При создании любой коллекции Mongo автоматически строит индекс по полю `_id`. Эти индексы присутствуют в коллекции `system.indexes`. Следующий запрос покажет все имеющиеся в базе данных индексы:

```
db.system.indexes.find()

{ "name" : "_id_", "ns" : "book.phones", "key" : { "_id" : 1 } }
```

Как правило, в запросах указывают и другие поля, помимо `_id`, поэтому по ним тоже следует построить индексы.

Мы построим индекс со структурой В-дерева по полю `display`. Но поскольку наша цель – убедиться, что индекс действительно ус-

коряет работу, мы сначала выполним запрос без индекса. Чтобы получить сведения о том, как выполняется операция, мы воспользуемся методом `explain()`.

```
db.phones.find({display: "+1 800-5650001"}).explain()
{
  "cursor" : "BasicCursor",
  "nscanned" : 109999,
  "nscannedObjects" : 109999,
  "n" : 1,
  "millis" : 52,
  "indexBounds" : {
  }
}
```

На вашем компьютере результат может отличаться, но величина `millis` – время выполнения запроса в миллисекундах – скорее всего, будет двузначной.

Для построения индекса вызывается метод коллекции `ensureIndex(fields, options)`. Параметр `fields` – это объект, содержащий поля, по которым строится индекс, а параметр `options` описывает тип индекса. В данном случае нам нужен уникальный индекс по полю `display`, причем обнаруженные при построении дубликаты должны просто удаляться.

```
db.phones.ensureIndex(
  { display : 1 },
  { unique : true, dropDups : true }
)
```

Теперь снова выполним `find()` и посмотрим, что скажет `explain()`.

```
db.phones.find({ display: "+1 800-5650001" }).explain()
{
  "cursor" : "BtreeCursor display_1",
  "nscanned" : 1,
  "nscannedObjects" : 1,
  "n" : 1,
  "millis" : 0,
  "indexBounds" : {
    "display" : [
      [
        "+1 800-5650001",
        "+1 800-5650001"
      ]
    ]
  }
}
```

Величина `millis` упала от 52 до 0 – ускорение в бесконечное число раз (52 / 0)! Шучу, конечно, но ускорение на несколько порядков налицо. Обратите также внимание, что поле `cursor` теперь равно не Basic, а `BtreeCursor` (курсор указывает на место хранения данных, но сам никаких данных не содержит). MongoDB больше не выполняет полное сканирование коллекции, а обходит дерево для поиска требуемого значения. Важно и то, что количество просмотренных объектов снизилось с 109999 до 1 – поскольку индекс уникален.

`explain()` – полезная функция, но использовать ее следует только при проверке конкретных запросов. Для изучения производительности системы в тестовом или промышленном режиме понадобится *системный профилировщик*.

Зададим уровень профилирования 2 (при уровне 2 сохраняется информация обо всех запросах, при уровне 1 – только о «медленных» запросах, исполнявшихся более 100 миллисекунд) и запустим `find()`, как обычно.

```
db.setProfilingLevel(2)
db.phones.find({ display : "+1 800-5650001" })
```

В результате создается новый объект в коллекции `system.profile`, которую можно читать, как любую другую. В поле `ts` хранится время начала запроса, в поле `info` – строка с описанием операции, а в поле `millis` – затраченное время.

```
db.system.profile.find()
{
  "ts" : ISODate("2011-12-05T19:26:40.310Z"),
  "op" : "query",
  "ns" : "book.phones",
  "query" : { "display" : "+1 800-5650001" },
  "responseLength" : 146,
  "millis" : 0,
  "client" : "127.0.0.1",
  "user" : ""
}
```

Mongo умеет строить индексы и по значениям на более глубоких уровнях вложенности. Чтобы построить индекс по кодам зон, нужно воспользоваться нотацией именования полей с точками: `components.area`. В промышленной системе индекс всегда следует строить в фоновом режиме, задавая параметр `{ background : 1 }`:

```
db.phones.ensureIndex({ "components.area": 1 }, { background : 1 })
```

Если теперь с помощью `find()` поискать индексы для коллекции `phones`, то новый индекс окажется в списке последним. Первым идет

индекс по полю `_id`, который всегда создается автоматически, а вторым – построенный нами уникальный индекс.

```
db.system.indexes.find({ "ns" : "book.phones" })

{
  "name" : "_id_",
  "ns" : "book.phones",
  "key" : { "_id" : 1 }
}
{
  "_id" : ObjectId("4d2c96d1df18c2494fa3061c"),
  "ns" : "book.phones",
  "key" : { "display" : 1 },
  "name" : "display_1",
  "unique" : true,
  "dropDups" : true
}
{
  "_id" : ObjectId("4d2c982bdf18c2494fa3061d"),
  "ns" : "book.phones",
  "key" : { "components.area" : 1 },
  "name" : "components.area_1"
}
```

В заключение отметим, что для создания индекса над большой коллекцией может потребоваться много времени и системных ресурсов. Это следует учитывать и стараться строить индексы в периоды наименьшей нагрузки, в фоновом режиме и вручную, не применяя автоматизированные процедуры. В Сети есть и дополнительные советы по индексированию, но это основы, о которых неплохо бы помнить.

Агрегированные запросы

Рассмотренные вчера запросы полезны для того, чтобы извлечь данные, но их последующая обработка возлагалась на клиента. Пусть, например, требуется подсчитать количество телефонов, больших 559-9999; хотелось бы, конечно, чтобы такие операции умела выполнять сама СУБД. Как и в PostgreSQL, функция `count()` является самым простым агрегатором. Она принимает запрос и возвращает число (количество удовлетворяющих критерию документов).

```
db.phones.count({'components.number': { $gt : 5599999 } })
50000
```

Чтобы в полной мере осознать мощь других агрегированных запросов, добавим в коллекцию `phones` еще 100 000 номеров с другим кодом зоны.

```
populatePhones( 855, 5550000, 5650000 )
```


О пользе видоизменения

Агрегированные запросы возвращают структуру, отличную от отдельных документов, к которым мы привыкли. Функция `count()` агрегирует результаты, возвращая количество документов, `distinct()` возвращает массив результатов, а `group()` – документы, которые сама конструирует. Даже `mapreduce` в общем случае старается возвращать объекты, напоминающие документы, которые вы сохранили.

Команда `distinct()` возвращает удовлетворяющие критерию значения (не полные документы), если хотя бы одно существует. Вот как можно получить различные номера (без учета зоны), меньшие 5 550 005:

```
db.phones.distinct('components.number',
  {'components.number': { $lt : 5550005 } })

[ 5550000, 5550001, 5550002, 5550003, 5550004 ]
```

Хотя номер 5 550 000 встречается дважды (в зоне 800 и в зоне 855), в список он вошел только один раз.

Функция `group()` агрегирует результаты примерно так же, как запросы с фразой `GROUP BY` в `SQL`. Это самый сложный из агрегированных запросов в `Mongo`. Мы можем подсчитать все номера, большие 5 599 999, и поместить их в отдельные группы по коду зоны. Функции передаются три параметра: `key` – поле, по которому производится группировка; `cond` – условие, определяющее, какие значения нас интересуют; `reduce` – функция, которая решает, как выводить результаты.

Помните о каркасе `mapreduce`, который обсуждался в главе о `Riak`? Наши данные уже *отображены* на существующую коллекцию документов. Больше никакое распределение не требуется, достаточно просто редуцировать документы.

```
db.phones.group({
  initial: { count:0 },
  reduce: function(phone, output) { output.count++; },
  cond: { 'components.number': { $gt : 5599999 } },
  key: { 'components.area' : true }
})
```

```
[ { "800" : 50000, "855" : 50000 } ]
```

Следующие два примера несколько искусственны и предназначены лишь для демонстрации гибкости `group()`.

Функцию `count()` нетрудно реализовать самостоятельно с помощью следующего вызова `group()`. Ключ агрегирования в результат не включается.



```
db.phones.group({
  initial: { count:0 },
  reduce: function(phone, output) { output.count++; },
  cond: { 'components.number': { $gt : 5599999 } }
})
```

```
[ { "count" : 100000 } ]
```

Здесь мы сначала настраиваем начальный объект, записав в поле `count` значение 0, – заданные в этом объекте поля будут присутствовать в результате. Затем мы описываем, что делать с этим полем, – объявляем функцию `reduce`, которая прибавляет к `count` единицу для каждого встретившегося документа. И наконец, мы задаем для группы условие, ограничивающее множество редуцируемых документов. Результат будет такой же, как при вызове функции `count()`, потому что задано такое же условие. Ключ мы опустили, потому что хотим, чтобы в список помещался каждый встретившийся документ.

Можно реализовать и функцию `distinct()`. Для повышения производительности мы начнем с создания объекта, в котором номера хранятся в виде полей (по существу, мы создаем собственный вариант структуры данных *множество*). В функции `reduce` (которая вызывается для каждого найденного документа) мы просто сопоставляем элементу множества значение 1 (говоря, что этот номер нам нужен).

Технически больше ничего не надо. Однако, если мы хотим полностью повторить поведение функции `distinct()`, то необходимо вернуть массив целых чисел. Поэтому мы добавим метод `finalize(out)`, который вызывается один раз перед возвратом значения, чтобы преобразовать объект в массив значений полей. Затем эта функция преобразует строковые телефонные номера в целые числа (если вы хотите наблюдать, как жарится омлет, выполните показанную ниже функцию, на задавая функцию `finalize`).

```
db.phones.group({
  initial: { prefixes : {} },
  reduce: function(phone, output) {
    output.prefixes[phone.components.prefix] = 1;
  },
  finalize: function(out) {
    var ary = [];
    for(var p in out.prefixes) { ary.push( parseInt( p ) ); }
    out.prefixes = ary;
  }
})[0].prefixes
```

```
[ 555, 556, 557, 558, 559, 560, 561, 562, 563, 564 ]
```

Функция `group()` не менее мощная, чем фраза `GROUP BY` в SQL, но у реализации ее в Mongo есть и недостатки. Во-первых, результат не может содержать более 10 000 документов. К тому же, если коллекция сегментирована (эту тему мы будем рассматривать завтра), то `group()` вообще не будет работать. Есть еще много способов гибко настроить запрос. По этой и ряду других причин мы посвятим еще немного времени вопросу о реализации `mapreduce` в Mongo. Но сначала очертим границу между командами на стороне сервера и клиента, поскольку различие между ними будет играть важную роль в ваших приложениях.

Команды на стороне сервера

Если вы запустите следующую функцию из командной оболочки (или из программы, написанной с помощью языкового драйвера), то клиент будет запрашивать каждый из 100 000 телефонов, модифицировать его и сохранять новый документ на сервере.

mongo/update_area.js

```
update_area = function() {
  db.phones.find().forEach(
    function(phone) {
      phone.components.area++;
      phone.display = "+" +
        phone.components.country + " " +
        phone.components.area + " - " +
        phone.components.number;
      db.phone.update({ _id : phone._id }, phone, false);
    }
  )
}
```

Однако объект `Mongo db` располагает командой `eval()`, которая передает указанную функцию серверу. Это кардинально уменьшает трафик между клиентом и сервером, потому что код выполняется удаленно.

```
> db.eval(update_area)
```

Помимо вычисления JavaScript-функций, в Mongo встроен еще ряд готовых команд, большая часть которых выполняется на сервере, хотя для некоторых необходимо, чтобы текущей была база данных `admin` (сделать ее текущей позволяет команда `use admin`).

```
> use admin
> db.runCommand("top")
```

Команда `top` выводит сведения обо всех коллекциях на сервере.

```
> use book
> db.listCommands()
```

Команда `listCommands()` выводит список команд, многие из которых мы уже использовали. На самом деле, многие типичные команды, например подсчет телефонных номеров, можно выполнить с помощью метода `runCommand()`. Однако результаты будут слегка различаться.

```
> db.runCommand({ "count" : "phones" })
{ "n" : 100000, "ok" : 1 }
```

Само число (*n*) вычислено правильно (100 000), но в ответе возвращается объект, в котором есть поле *ok*. Объясняется это тем, что `db.phones.count()` — функция-обертка, созданная для нашего удобства оболочкой, тогда как метод `runCommand()` выполняется на сервере. Напомним, что мы можем изучить исходный код любой функции, в частности `count()`, набрав ее имя без последующих скобок.

```
> db.phones.count
function (x) {
  return this.find(x).count();
}
```

Интересно! Метод `collection.count()` — простая обертка для вызова `count()` применительно к результатам функции `find()` (а та и сама является оберткой вокруг встроенного объекта запроса, который возвращает курсор, указывающий на результаты). Если же выполнить *такой* запрос...

```
> db.phones.find().count
```

то мы увидим куда более длинную функцию (слишком длинную, чтобы приводить здесь ее код). Но присмотритесь к коду — после разного рода инициализации вы обнаружите такие строки:

```
var res = this._db.runCommand(cmd);
if (res && res.n != null) {
  return res.n;
}
```

Вдвойне интересно! Функция `count()` выполняет метод `runCommand()` и возвращает значение поля `n`.

Функция runCommand

И раз уж мы завели разговор о том, как работают методы, рассмотрим еще и функцию `runCommand()`

```
> db.runCommand
function (obj) {
  if (typeof obj == "string") {
    var n = {};
    n[obj] = 1;
    obj = n;
  }
  return this.getCollection("$cmd").findOne(obj);
}
```

Оказывается, что `runCommand()` — тоже вспомогательная функция, обертывающая вызов метода коллекции `$cmd`. Любую команду можно выполнить, напрямую обратившись к этой коллекции.

```
> db.$cmd.findOne({'count' : 'phones'})
{ "n" : 100000, "ok" : 1 }
```

Но это уже уровень «железа» — так общаются с сервером Mongo языковые драйверы.

Отступление от темы

Мы включили этот небольшой раздел по двум причинам:

- чтобы донести до вас ту мысль, что большая часть работы, иницируемой с консоли `mongo`, выполняется на стороне сервера, а не клиента, который лишь предоставляет вспомогательные функции-обертки;
- чтобы показать, что идеей исполнения кода на стороне сервера можно воспользоваться и в своих целях, создав в MongoDB нечто похожее на *хранимые процедуры* в PostgreSQL.

Любую JavaScript-функцию можно сохранить в специальной коллекции `system.js`. Это обычная коллекция и, чтобы сохранить в ней функцию, нужно просто записать ее имя в поле `_id`, а объект-функцию — в поле `value`.

```
> db.system.js.save({
  _id: 'getLast',
  value: function(collection) {
    return collection.find({}).sort({'_id': 1}).limit(1)[0]
  }
})
```

А дальше можно выполнить эту функцию прямо на сервере, передав ее имя функции `eval()`. Сервер исполнит JavaScript-код и вернет результаты.

```
> db.eval('getLast(db.phones)')
```

Результаты должны быть точно такими же, как при локальном вызове `getLast(collection)`.

```
> db.system.js.findOne({'_id': 'getLast'}).value(db.phones)
```

Следует отметить, что на время выполнения `eval()` работа сервера `mongod` блокируется, поэтому такой подход применяется главным образом для запуска быстрых одноразовых задач или тестов, а не как общий механизм исполнения процедур в производственной системе. Сохраненную функцию можно вызывать также из оператора `$where` и из функций `mapreduce`. Теперь наш арсенал пополнился последним инструментом, необходимым, чтобы приступить к исследованию каркаса `mapreduce` в `MongoDB`.

Mapreduce (и Finalize)

Принцип работы `mapreduce` в `Mongo` аналогичен тому, что мы видели в `Riak`, но есть и небольшие отличия. Функция `map()` не возвращает преобразованное значение; вместо этого `Mongo` требует, чтобы распределитель вызывал функцию `emit()`, передавая ей ключ. Идея в том, что функцию `emit()` можно вызывать несколько раз при обработке одного документа. Функция `reduce()` принимает один ключ и список значений, эмитированных для данного ключа. Наконец, в `Mongo` имеется необязательный третий шаг, `finalize()`, который выполняется ровно один раз на каждое распределенное значение – после завершения работы редукторов. Это позволяет произвести завершающие вычисления или очистку.

Поскольку мы уже знакомы с основами `mapreduce`, то пройдем мимо бассейна для малышей сразу к десятиметровой вышке для прыжков в воду. Сгенерируем отчет, в котором для каждой страны подсчитывается количество телефонов, содержащих одинаковый набор цифр. Для начала сохраним вспомогательную функцию, которая возвращает массив различных цифр в номере телефона (для понимания принципа работы `mapreduce` устройство этой функции не столь важно).

mongo/distinct_digits.js

```
distinctDigits = function(phone){
  var
    number = phone.components.number + '',
    seen = [],
    result = [],
```

```

    i = number.length;
    while(i--) {
        seen[+number[i]] = 1;
    }
    for (i=0; i<10; i++) {
        if (seen[i]) {
            result[result.length] = i;
        }
    }
    return result;
}
db.system.js.save({_id: 'distinctDigits', value: distinctDigits})

```

Загрузите этот файл в оболочку `mongo`. Если файл находится в том же каталоге, из которого запускается программа `mongo`, то достаточно указать только имя файла, иначе – полный путь к нему.

```
> load('distinct_digits.js')
```

Затем прогоним простенький тест (если вы не вполне понимаете, что делает функция, не робейте и понаставляйте вызовы `print()`).

```

db.eval("distinctDigits(db.phones.findOne({ 'components.number' : 5551213 }))")
[ 1, 2, 3, 5 ]

```

Теперь у нас есть, чем занять распределитель. Как всегда в `map-reduce`, решение о том, какие поля должен формировать распределитель, критично, так как от этого зависит, какие будут возвращены агрегированные значения. Поскольку наш отчет показывает телефоны с разными цифрами, то одним полем будет массив различающихся цифр. А поскольку мы хотим запрашивать данные по стране, то вторым полем будет страна. Скомпонуем из обоих полей составной ключ: `{digits : X, country : Y}`.

Наша задача – просто подсчитать количество значений, поэтому будем эмитировать значение 1 (каждый документ добавляет единицу к итогу). Задача редуктора – просуммировать эти единицы.

```

mongo/map_1.js
map = function() {
    var digits = distinctDigits(this);
    emit({digits : digits, country : this.components.country}, {count : 1});
}

mongo/reduce_1.js
reduce = function(key, values) {
    var total = 0;
    for(var i=0; i<values.length; i++) {
        total += values[i].count;
    }
}

```



```

    return { count : total };
}

results = db.runCommand({
  mapReduce: 'phones',
  map: map,
  reduce: reduce,
  out: 'phones.report'
})

```

В параметре `out` мы задали имя коллекции (`out : 'phones.report'`), и ее можно опросить, как любую другую. Это материализованное представление, которое показывает команда `show tables`.

```

> db.phones.report.find({'_id.country' : 8})
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 4, 5, 6 ], "country" : 8 },
  "value" : { "count" : 19 }
}
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 5 ], "country" : 8 },
  "value" : { "count" : 3 }
}
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 5, 6 ], "country" : 8 },
  "value" : { "count" : 48 }
}
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 5, 6, 7 ], "country" : 8 },
  "value" : { "count" : 12 }
}
has more

```

Введите команду `it`, чтобы продолжить обход результатов. Обратите внимание, что уникальные эмитированные ключи находятся в поле `_id`, а данные, возвращенные редукторами, — в поле `value`.

Если вы предпочитаете, чтобы редуктор просто выводил результаты, а не записывал их в коллекцию, то можете присвоить `out` значение `{ inline : 1 }`, но помните об ограничении на размер результата. В версии Mongo 2.0 он составляет 16 МБ.

Напомним, что в главе, посвященной Riak, отмечалось, что на вход редуктора может подаваться как выход распределителя, так и выход других редукторов. Зачем подавать выход одного редуктора на вход другого, если ключи результатов одинаковы? Подумайте, как это могло бы выглядеть при запуске на разных серверах (рис. 22).

Каждый сервер должен исполнять собственные экземпляры функций `map()` и `reduce()`, а затем передавать результаты для объеди-

нения тому серверу, который инициировал вызов. Это классический алгоритм «разделяй и властвуй». Если бы мы назвали выход редуктора `total`, а не `count`, то пришлось бы обрабатывать оба случая, как показано ниже.

`mongo/reduce_2.js`

```
reduce = function(key, values) {  
  var total = 0;  
  for(var i=0; i<values.length; i++) {  
    var data = values[i];  
    if('total' in data) {  
      total += data.total;  
    } else {  
      total += data.count;  
    }  
  }  
  return { total : total };  
}
```

Однако разработчики Mongo предвидели, что могут понадобиться какие-то изменения на завершающем этапе, например переименование поля или дополнительные вычисления. Если нам действительно так важно, чтобы поле результата называлось `total`, то можно написать функцию `finalize()`, которая работает так же, как в случае `group()`.

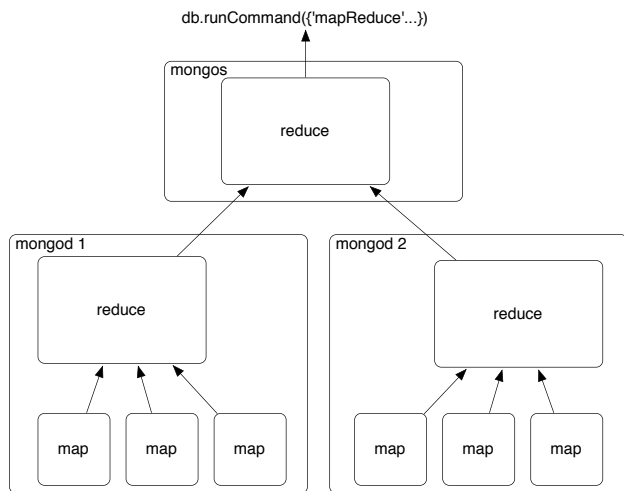


Рис. 22. Распределители и редукторы работают на двух серверах Mongo



День 2: итоги

Сегодня мы рассмотрели более сложные запросы, включающие агрегатные функции: `count()`, `distinct()` и, наконец, `group()`. Для ускорения их выполнения мы пользовались встроенными в MongoDB средствами индексирования. Если всего этого недостаточно, то к нашим услугам каркас `mapreduce`.

День 2: домашнее задание

Информационный поиск

1. Найдите сокращенные эквиваленты административных команд.
2. Найдите онлайнную документацию по запросам и курсорам.
3. Найдите на сайте MongoDB документацию по каркасу `MapReduce`.
4. Пользуясь JavaScript-интерфейсом, исследуйте код трех методов коллекций: `help()`, `findOne()` и `stats()`.

Задачи

1. Реализуйте метод `finalize` для вывода суммарного счетчика в виде поля `total`.
2. Установите драйвер своего любимого языка для Mongo и подключитесь к базе данных. Заполните какую-нибудь коллекцию и постройте над ней индекс по одному из полей.

5.4. День 3: наборы реплик, сегментирование, пространственные данные и GridFS

Mongo обладает развитыми средствами для хранения и опроса данных. Но они есть и в других СУБД. Уникальная особенность документных баз данных заключается в способности эффективно обрабатывать бессхемные документы с произвольным уровнем вложенности. А конкретно Mongo отличается умением масштабироваться на несколько серверов путем репликации (копирования данных на другие серверы) или сегментирования коллекций (разбиения коллекции на части), а также умением параллельно выполнять запросы. То и другое повышает доступность.

Наборы реплик

Mongo разрабатывалась не как монолитный сервер, а с прицелом на горизонтальное масштабирование. В проект заложена согласованность данных и устойчивость к частичной потере связности сети. Однако у сегментирования есть свои минусы: если какая-то часть коллекции потеряна, то и вся коллекция теряет ценность. Какой смысл опрашивать коллекцию стран, в которой представлены только страны западного полушария? Mongo решает эту внутренне присущую сегментированию проблему очень просто: за счет дублирования. Редко встретишь производственную систему, состоящую всего из одного экземпляра Mongo; чаще данные реплицируются на несколько серверов.

Сегодня мы не будем и дальше мучить уже существующую базу данных, а начнем с чистого листа и запустим несколько новых серверов. По умолчанию Mongo прослушивает порт 27017, поэтому дополнительным серверам мы назначим другие порты. Напомним, что сначала нужно создать каталоги данных:

```
$ mkdir ./mongo1 ./mongo2 ./mongo3
```

Далее запустим серверы Mongo. На этом раз зададим флаг `replSet` со значением `book` и укажем номера портов. Заодно зададим флаг `rest`, чтобы можно было обращаться к веб-интерфейсу.

```
$ mongod --replSet book --dbpath ./mongo1 --port 27011 --rest
```

Откройте еще одно окно терминала и запустите аналогичную команду для запуска второго сервера, указав другой каталог и другой порт. Точно так же запустите третий сервер в третьем окне терминала.

```
$ mongod --replSet book --dbpath ./mongo2 --port 27012 --rest
$ mongod --replSet book --dbpath ./mongo3 --port 27013 --rest
```

Обратите внимание, что будет выводиться следующее сообщение:

```
[startReplSets] replSet can't get local.system.replset config from self \
or any seed (EMPTYCONFIG)
```

Это нормально – нам еще только предстоит инициализировать набор реплик, и Mongo напоминает об этом. Откройте оболочку mongo, подключитесь к любому серверу и выполните функцию `rs.initiate()`.

```
$ mongo localhost:27011
> rs.initiate({
```



```
_id: 'book',
members: [
  {_id: 1, host: 'localhost:27011'},
  {_id: 2, host: 'localhost:27012'},
  {_id: 3, host: 'localhost:27013'}
]
})
> rs.status()
```

Здесь мы воспользовались новым объектом `rs` (replica set – набор реплик). У него, как и любого другого объекта, имеется метод `help()`. Команда `status()` позволяет узнать, работает ли данный набор реплик, продолжайте с ее помощью проверять состояние, пока инициализация не завершится, только потом можно будет продолжать. Наблюдая за тем, что выводят все три сервера, вы обнаружите, что один напечатает строку

```
[rs Manager] replSet PRIMARY
```

а два других – строку

```
[rs_sync] replSet SECONDARY
```

Тот сервер, что напечатал `PRIMARY`, является главным. Скорее всего, это будет сервер, прослушивающий порт `27011` (так как он был запущен первым); если это не так, подключите консоль к главному серверу. Попробуйте вставить какую-нибудь чепуху из командного интерфейса и посмотрите, что получится.

```
> db.echo.insert({ say : 'HELLO!' })
```

После вставки выйдите из консоли и проверьте, что изменение реплицировано. Для этого остановите главный узел, достаточно просто нажать `CTRL+C`. Заглянув в журналы двух оставшихся серверов, вы обнаружите, что один из них повышен в ранге до главного (он выведет строку `replSet PRIMARY`). Откройте консоль и подключитесь к этой машине (у нас это был сервер `localhost:27012`), после чего выполните команду `db.echo.find()` – она должна вернуть введенное ранее значение.

Сыграем в перетасовывание консолей еще разок. Подключите консоль к оставшемуся подчиненному (`SECONDARY`) серверу. На всякий случай выполните функцию `isMaster()`. В нашем случае ее результат выглядел так:

```
$ mongo localhost:27013
MongoDB shell version: 1.6.2
connecting to: localhost:27013/test
```

```
> db.isMaster()
{
  "setName" : "book",
  "ismaster" : false,
  "secondary" : true,
  "hosts" : [
    "localhost:27013",
    "localhost:27012",
    "localhost:27011"
  ],
  "primary" : "localhost:27012",
  "ok" : 1
}
```

В этом экземпляре оболочки попробуем вставить еще одно значение.

```
> db.echo.insert({ say : 'is this thing on?' })
not master
```

Сообщение *not master* означает, что мы не можем писать на подчиненный сервер. И читать непосредственно с него тоже не получится. В каждом наборе реплик существует только один главный узел, и взаимодействовать нужно именно с ним. Это шлюз ко всему набору.

С репликацией данных сопряжены проблемы, отсутствующие в базах данных с единственным источником. В случае Mongo одна из проблем состоит в том, чтобы решить, какой узел повышать в ранге, когда главный сервер выходит из строя. Mongo решает ее, проводя выборы процесса `mongod`. Побеждает тот, у которого самые свежие данные. В настоящий момент у вас должно работать две службы `mongod`. Попробуйте остановить текущий главный узел. Когда мы проделали это, имея три узла, один из оставшихся стал новым главным. Но теперь всё будет по-другому. В журнале последнего оставшегося сервера появится запись такого вида:

```
[ReplSetHealthPollTask] replSet info localhost:27012 is now down (или...
[rs Manager] replSet can't see a majority, will not try to elect self

([rs Manager] replSet не видит большинства, не пытаюсь выбрать самого себя)
```

Отсюда становится понятна философия, стоящая за конфигурированием серверов в Mongo, и причина, по которой число серверов всегда должно быть нечетно (три, пять и т. д.).

А теперь снова запустите другие серверы и загляните в журналы. Когда узел поднимается, он переходит в состояние восстановления и пытается ресинхронизировать свои данные с новым главным узлом. «Минуточку! – восклицаете вы. – А что если у исходного главного



узла были данные, которые еще не успели распространиться?» Такие операции просто отбрасываются. Операция записи в наборе реплик не считается успешной, пока на большинстве узлов не будет сохранена копия данных.

Проблема четного числа узлов

Идея репликации довольно проста: вы пишете на один сервер MongoDB, а затем данные копируются на все остальные узлы набора реплик. Если главный сервер недоступен, то на его роль выбирается один из оставшихся, и он продолжает обслуживать запросы. Однако сервер может оказаться недоступен не только в результате выхода из строя. Иногда пропадает сетевое соединение между узлами. В таком случае Mongo заявляет, что *сеть состоит из большинства узлов, все еще способных видеть друг друга*.

MongoDB ожидает, что в наборе реплик нечетное число узлов. Рассмотрим, к примеру, сеть из пяти узлов. Если в результате потери связности сеть распалась на два фрагмента – с тремя и двумя узлами, – то больший фрагмент составляет явное большинство, поэтому может выбрать главный узел и продолжать обслуживание запросов. В отсутствие явного большинства собрать кворум невозможно.

Чтобы понять, почему необходимо нечетное число узлов, рассмотрим, что может произойти, когда в наборе реплик четыре узла. Предположим, что в результате потери связности образовалось два сегмента из двух узлов. В одном из них по-прежнему находится исходный главный узел, но поскольку он не видит *явного большинства* сети, то снимает с себя обязанности главного сервера. Но и в другом фрагменте выбрать главный узел невозможно, потому что нет связи с явным большинством узлов. Ни тот, ни другой набор не могут обслуживать запросы, и вся система останавливается. При наличии нечетного числа узлов этот сценарий – фрагментированная сеть, ни в одном фрагменте которой нет явного большинства, – менее вероятен.

Некоторые СУБД (например, CouchDB) допускают наличие нескольких главных узлов, но Mongo к их числу не относится и не умеет разрешать конфликты, возникающие при обновлении данных. MongoDB подходит к конфликтам между несколькими главными узлами просто – не допускает их вовсе.

В отличие от, скажем, Riak, Mongo всегда знает последнее записанное значение; клиенту ничего решать не надо. Цель Mongo – обеспечить строгую согласованность при записи, и запрет нескольких главных узлов – не самый плохой метод ее достижения.

Голосование и арбитры

Не всегда возможно обеспечить наличие нечетного числа серверов для репликации данных. В таком случае можно либо назначить специального арбитра (рекомендуется), либо дать больше голосов некоторым серверам (не рекомендуется). В Mongo арбитром называется член набора реплик, который принимает участие в голосовании, но не занимается репликацией. Запускается он как обычный сервер, но в конфигурационном файле задается специальный флаг, например: `{_id: 3, host: 'localhost:27013', arbiterOnly : true}`. Арбитры полезны для выхода из тупика и этим напоминают роль вице-президента США в Сенате. По умолчанию у каждого экземпляра `mongod` имеется ровно один голос.

Сегментирование

Одна из основных причин существования Mongo – безопасная и быстрая обработка очень больших наборов данных. Очевидный способ достижения этой цели – горизонтальное сегментирование по диапазонам значений – или для краткости просто *сегментирование* (sharding). Вместо того чтобы хранить всю коллекцию на одном сервере, она разбивается на части (иными словами, сегментируется), которые размещаются на нескольких серверах. Например, мы могли бы поместить телефонные номера, меньшие 1-500-000-0000, на сервер А, а большие или равные 1-500-000-0001 – на сервер В. Mongo упрощает решение этой задачи с помощью механизма автосегментирования, который самостоятельно производит разбиение.

Давайте запустим два (нереплицирующих) сервера `mongod`. Как и в случае набора реплик, существует специальный параметр, означающий, что данный сервер может участвовать в сегментировании.

```
$ mkdir ./mongo4 ./mongo5
$ mongod --shardsvr --dbpath ./mongo4 --port 27014
$ mongod --shardsvr --dbpath ./mongo5 --port 27015
```

Теперь нужно сделать так, чтобы сервер действительно вел учет ключей. Предположим, что мы создали таблицу для хранения названий городов в алфавитном порядке. Необходимо как-то сообщить, что названия, начинающиеся с букв от А до N (к примеру) должны храниться на сервере `mongo4`, а начинающиеся с букв от О до Z – на сервере `mongo5`. В Mongo для этой цели создается *конфигурационный сервер* (обычный экземпляр `mongod`, запущенный со специальным флагом), который учитывает, какой сервер (`mongo4` или `mongo5`) какие значения хранит.

```
$ mkdir ./mongoconfig
$ mongod --configsvr --dbpath ./mongoconfig --port 27016
```

Наконец, нам необходим еще четвертый сервер, `mongos`, являющийся единственной точкой входа для клиентов. Сервер `mongos` подключается к конфигурационному серверу (назовем его `mongoconfig`) для получения информации о сегментировании. Запустим его на порту 27020, задав параметр `chunkSize` (размер порции) равным 1. (Порция размером 1 МБ является минимально допустимой. Такое значение позволит наблюдать за тем, как происходит сегментирование нашего небольшого набора данных. В производственной системе следовало бы оставить значение по умолчанию или выбрать гораздо больший размер.) О местонахождении конфигурационного сервера (доменное имя:номер порта) `mongos` узнает благодаря флагу `--configdb`.

```
$ mongos --configdb localhost:27016 --chunkSize 1 --port 27020
```

mongos и mongoconfig

Может возникнуть вопрос, зачем Mongo разделяет роли *конфигурация и точка входа* (`mongos`). Связано это с тем, что в производственных системах эти процессы обычно запускаются на физически различных серверах. Конфигурационный сервер (который сам реплицируется) управляет информацией о сегментировании в интересах других сегментированных серверов, тогда как `mongos` обычно запускается на той же машине, где работает сервер приложений, чтобы клиенты легко могли к нему подключаться (не думая о том, где находятся конкретные сегменты).

Интересно, что `mongos` является облегченной версией полнофункционального сервера `mongod`. Почти все команды, распознаваемые `mongod`, распознаются и `mongos`, что делает его идеальным посредником для клиентов, которым необходимо подключаться к нескольким сегментированным серверам. На рис. 23 изображена конфигурация наших серверов.

Теперь подключите консоль к базе данных `admin` на сервере `mongos`. Сейчас мы займемся конфигурированием сегментов.

```
$ mongo localhost:27020/admin
> db.runCommand( { addshard : "localhost:27014" } )
{ "shardAdded" : "shard0000", "ok" : 1 }
> db.runCommand( { addshard : "localhost:27015" } )
{ "shardAdded" : "shard0001", "ok" : 1 }
```

Далее необходимо указать базу данных, подлежащую сегментированию коллекцию и поле, по которому сегментировать (в нашем случае название города).


```
> db.runCommand( { enablesharding : "test" } )  
{ "ok" : 1 }  
> db.runCommand( { shardcollection : "test.cities", key : {name : 1} } )  
{ "collectionsharded" : "test.cities", "ok" : 1 }
```

Итак, настройка завершена и можно загружать данные. В исходном коде, прилагаемом к книге, имеется файл `mongo_cities1000.json` размером 12 МБ, в котором содержатся сведения обо всех городах мира с населением больше 1000 человек. Скачайте этот файл и запустите скрипт, который импортирует данные на наш сервер `mongos`:

```
$ mongoimport -h localhost:27020 -db test --collection cities \  
--type json mongo_cities1000.json
```

На подключенной к `mongos` консоли введите команду `use test`, чтобы вернуться в базу данных `test`.

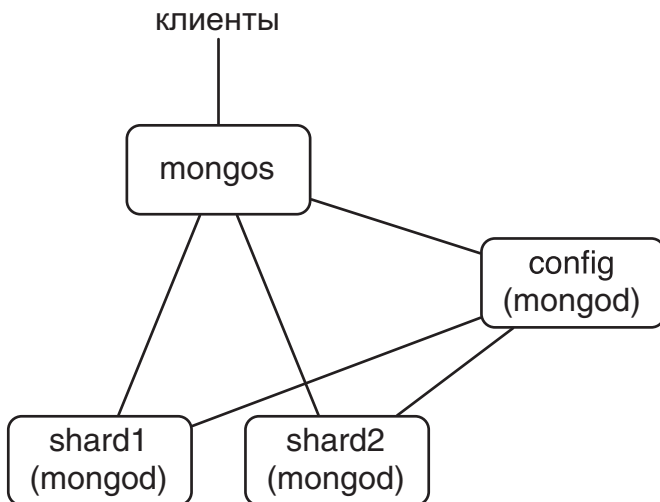


Рис. 23. Наш крохотный сегментированный кластер

Пространственные запросы

В Mongo имеется одна замечательная штука. Хотя основной темой на сегодня является настройка серверов, без небольшого аттракциона день был бы пропащим. И в качестве такого развлечения мы представим пространственные запросы в Mongo. Для начала подключитесь к серверу `mongos`.

```
$ mongo localhost:27020
```



Секрет пространственных запросов заключается в специальном способе индексирования географических данных, который называется *геохешированием*. Такой индекс позволяет искать не только по значению или диапазону значений координат, но и быстро находить близлежащие точки. Кстати говоря, в предыдущем разделе мы как раз загрузили большой массив географических данных. Чтобы обращаться к ним с запросами, мы должны первым делом проиндексировать данные по полю *location*. Для построения двумерного индекса (типа *2d*) необходимо указать любые два поля. В нашем случае они представлены в виде хеша (например, { longitude:1.48453, latitude:42.57205 }), но с тем же успехом могли бы быть элементами массива (например, [1.48453, 42.57205]).

```
> db.cities.ensureIndex({ location : "2d" })
```

Если бы коллекция не была сегментирована, то можно было бы запрашивать, какой город (или города) находится в данной точке или рядом с ней. Но в текущей версии Mongo показанный ниже запрос работает только для несегментированных коллекций.

```
> db.cities.find({ location : { $near : [45.52, -122.67] } }).limit(5)
```

В будущих версиях планируется распространить эту возможность и на сегментированные коллекции. А пока для поиска близлежащих городов в сегментированной коллекции *cities* пользуйтесь командой *geoNear()*. Вот как выглядит возвращаемый ей результат:

```
> db.runCommand({geoNear : 'cities', near : [45.52, -122.67],
  num : 5, maxDistance : 1})
{
  "ns" : "test.cities",
  "near" : "1000110001000000011100101011100011001001110001111110",
  "results" : [
    {
      "dis" : 0.007105400003747849,
      "obj" : {
        "_id" : ObjectId("4d81c216a5d037634ca98df6"),
        "name" : "Portland",
        ...
      }
    },
    ...
  ],
  "stats" : {
    "time" : 0,
    "btreelocs" : 53,
    "nscanned" : 49,
```

```

    "objectsLoaded" : 6,
    "avgDistance" : 0.02166813996454613,
    "maxDistance" : 0.07991909980773926
  },
  "ok" : 1
}

```

Команда `geoNear()` заодно помогает отлаживать пространственные команды. Она возвращает кладезь полезной информации, например: расстояние до указанной в запросе точки, среднее и максимальное расстояние для возвращенного набора, а также сведения о самом индексе.

GridFS

Один из недостатков распределенной системы – отсутствие единой файловой системы. Допустим, вы работаете с сайтом, на который пользователи могут загружать фотографии. Если сайт обслуживается несколькими веб-серверами, размещенными в разных узлах, то необходимо либо вручную копировать загруженные изображения на каждый сервер, либо организовывать какую-то централизованную файловую систему. В Mongo для такого случая имеется собственная распределенная файловая система GridFS.

В дистрибутив Mongo включена командная утилита для взаимодействия с GridFS. Что замечательно, для работы с ней ничего не нужно настраивать. Если сейчас с помощью команды `mongofiles` запросить файлы на управляемых `mongos` сегментах, то мы получим пустой список.

```

$ mongofiles -h localhost:27020 list
connected to: localhost:27020

```

Но попробуйте загрузить какой-нибудь файл.

```

$ mongofiles -h localhost:27020 put my_file.txt
connected to: localhost:27020
added file: { _id: ObjectId('4d81cc96939936015f974859'), \
  filename: "my_file.txt", chunkSize: 262144, \
  uploadDate: new Date(1300352150507), \
  md5: "844ab0d45e3bded0d48c2e77ed4f3b0e", length: 3067 }
done!

```

И, о чудо, если теперь снова выполнить `mongofiles`, то мы обнаружим загруженный файл.

```

$ mongofiles -h localhost:27020 list
connected to: localhost:27020
my_file.txt 3067

```

Вернувшись в консоль `mongo`, можно посмотреть, в каких коллекциях Mongo хранит данные.

```
> show collections
cities
fs.chunks
fs.files
system.indexes
```

Так как это самые обычные коллекции, то их можно реплицировать и опрашивать привычными способами.

День 3: итоги

На этом завершается наше исследование MongoDB. Сегодня мы познакомились с тем, как Mongo обеспечивает долговечность данных с помощью наборов реплик, и с поддержкой горизонтальной масштабируемости за счет сегментирования. Мы показали, как следует конфигурировать кластер серверов, и прояснили роль сервера `mongos` в качестве посредника, управляющего автосегментированием между несколькими узлами. Наконец, мы поэкспериментировали с дополнительными встроенными в Mongo средствами, а именно с пространственными запросами и распределенной файловой системой GridFS.

День 3: домашнее задание

Информационный поиск

1. Прочитайте в онлайн-официальной документации обо всех конфигурационных параметрах набора реплик.
2. Выясните, как создается трехмерный пространственный индекс (по сферическим координатам).

Задачи

1. В Mongo имеется поддержка ограничивающих фигур (точнее, квадратов и кругов). Найдите все города, расположенные в пределах 50 миль от центра Лондона⁴.
2. Запустите шесть серверов: по три сервера в двух наборах реплик, каждый из которых является сегментом. Запустите конфигурационный сервер и `mongos`. Настройте в такой конфигурации GridFS (это решающий экзамен).

⁴ <http://www.mongodb.org/display/DOCS/Geospatial+Indexing>

5.5. Резюме

Надеемся, что это краткое знакомство с MongoDB пробудило в вас интерес и доказало, что не зря она заслужила название «монструозной» базы данных. Это была довольно насыщенная глава, но, как обычно, мы лишь пробежались по самым верхам.

Сильные стороны Mongo

Главное достоинство Mongo – способность обрабатывать гигантские массивы данных (и огромный поток запросов) за счет репликации и горизонтального масштабирования. Но она также предлагает весьма гибкую модель данных, так как не нужно заранее определять схему, а данные могут быть вложенными (для достижения аналогичного эффекта в РСУБД пришлось бы использовать соединения).

Наконец, при проектировании MongoDB закладывалась простота использования. Возможно, вы обратили внимание на сходство между командами Mongo и некоторыми концепциями баз данных на основе SQL (за вычетом соединений на стороне сервера). Это не случайно, и именно по этой причине Mongo привлекает так много сторонников из числа бывших пользователей объектно-реляционных моделей (ORM). Эта система имеет достаточно отличий, чтобы раззадорить многих разработчиков, но не настолько много, чтобы отпугнуть своей чуждостью.

Слабые стороны Mongo

То, что Mongo поощряет денормализацию схемы (хотя сами схемы и отсутствуют), некоторым может показаться неприемлемым. Есть разработчики, в которые жесткие ограничения, налагаемые реляционной СУБД с ее холодной логикой, вселяют уверенность. Возможность вставки в любую коллекцию значения произвольного типа вызывает опасения. Единственная опечатка может стать причиной многочасовых мучений, если вы не догадаетесь искать причину в именах полей и коллекций. Гибкость Mongo – не такое уж большое преимущество, если модель данных уже достаточно зрелая и давно зафиксирована.

Поскольку Mongo ориентирована на большие наборы данных, то использовать ее имеет смысл в крупных кластерах, для проектирования и обслуживания которых требуются некоторые усилия. В отличие от Riak, где добавление новых узлов производится прозрачно и почти незаметно в процессе эксплуатации, настройка кластера Mongo подразумевает предварительное планирование.

Перед расставанием

Mongo – отличный выбор для тех, кто уже привычно использует для хранения данных реляционную базу данных и обращается к ней с помощью какой-нибудь системы объектно-реляционного отображения. Мы часто рекомендуем ее работающим на платформах Rails, Django и вообще всем, кто пользуется паттерном модель-представление-контроллер (MVC), так как они все равно реализуют проверку данных и управление полями с помощью моделей на уровне приложения и так как с миграцией схемы можно будет распрощаться (по большей части). Для добавления новых полей в документ достаточно просто добавить поле в модель данных, и Mongo спокойно воспримет новые понятия. Мы считаем, что по сравнению с реляционными базами данных Mongo дает гораздо более естественное решение многих типичных задач, в которых набор данных определяется приложением.



ГЛАВА 6.

CouchDB

Трещоточный гаечный ключ – легкий и удобный инструмент, пригодный как для мелких, так и для серьезных работ. Как и для электродрели, для него существуют различные насадки. Но в отличие от дрели, которой нужна сеть переменного тока, гаечный ключ спокойно помещается в кармане и работает на мускульной силе. Вот так и Apache CouchDB. Эта база данных может масштабироваться вверх и вниз, а потому легко адаптируется к задачам любого размера и сложности.

CouchDB – типичная документо-ориентированная база данных на основе JSON и REST. Уже первая версия, выпущенная в 2005 году, была спроектирована в расчете на сеть веб и все бесчисленные ошибки, отказы и «глюки», которые ей сопутствуют. Поэтому по стабильности с CouchDB не может сравниться почти ни одна из существующих баз данных. Если другие системы способны пережить случайные пропадания сети, то CouchDB прекрасно себя чувствует даже тогда, когда появление сети – редкое событие.

Как и MongoDB, CouchDB хранит *документы* – JSON-объекты, состоящие из пар ключ-значение, причем значениями могут быть данные разных типов, в том числе и другие объекты с неограниченной вложенностью. Однако произвольные запросы не поддерживаются; основной способ поиска документов – это индексированные представления, порождаемые инкрементной процедурой *mapreduce*.

6.1. Располагайтесь на кушетке¹

CouchDB побуждает расслабиться – как и следует из ее названия. Она рассчитана не только на гигантские кластеры из дорогих серверов, но и на другие сценарии развертывания: от центра обработки данных до смартфона. Запустить CouchDB можно на телефоне Android, на персональном MacBook'e или в ЦОДе. Будучи написана на Er-

¹ couch – кушетка. Прим. перев.

lang, CouchDB на удивление живуча: остановить ее можно только одним способом – убив процесс! А модель хранения, допускающая только дописывание, гарантирует, практическую неповреждаемость данных, а также простоту репликации, резервного копирования и восстановления.

CouchDB – документо-ориентированная база данных, в которой форматом хранения и взаимодействия с внешним миром является JSON. Как и в Riak, все обращения к CouchDB производятся с помощью REST-интерфейса. Репликация может быть как одно-, так и двусторонней, по запросу или непрерывной. CouchDB обеспечивает высокую гибкость при принятии решений о структуре, защите и распределении данных.

Сравнение CouchDB с MongoDB

Один из самых интересных вопросов в этой книге – в чем разница между CouchDB и MongoDB? На первый взгляд, CouchDB и MongoDB (рассматривалась в главе 5) выглядят очень похоже. Та и другая – документо-ориентированные хранилища данных, тесно связанные с языком JavaScript. В обоих форматом транспортировки данных является JSON. Тем не менее, существует много различий, начиная с идеологии проекта и кончая реализацией и характеристиками масштабируемости. Часть из них мы опишем, когда будем исследовать CouchDB во всей ее восхитительной простоте.

Во время нашего трехдневного путешествия мы рассмотрим многие притягательные особенности CouchDB и выбранные проектные решения. Как всегда, начнем с операций CRUD, а затем перейдем к индексированию с помощью mapreduce-представлений. Как и в других случаях, мы импортируем кое-какие структурированные данные и на их примере изучим ряд более сложных концепций. Наконец, мы разработаем несколько простеньких событийно-управляемых клиентских приложений с помощью Node.js и изучим, как принятая в CouchDB стратегия репликации главный-главный позволяет разрешать конфликты при обновлении. В путь!

6.2. День 1: операции CRUD, Futon и снова cURL

Сегодня мы отправимся в путешествие по CouchDB, и нашим спутником будет дружелюбный веб-интерфейс Futon, позволяющий

выполнять простые операции CRUD. Затем мы снова обратимся к программе сURL, которой пользовались для взаимодействия с Riak в главе 3, она поможет нам отправлять REST-запросы. Все библиотеки и драйверы для CouchDB под капотом посылают REST-запросы, поэтому неплохо бы понимать, как они устроены.

Знакомство с Futon

В состав CouchDB входит полезный веб-интерфейс, который называется Futon². Установив и запустив CouchDB, откройте браузер и перейдите по адресу http://localhost:5984/_utils/. Откроется страница Overview (Общие сведения), показанная на рис. 24.

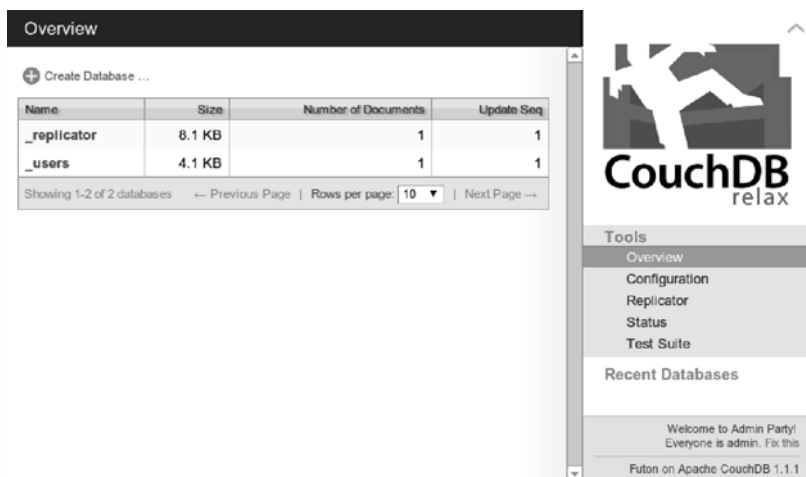


Рис. 24. CouchDB Futon: страница Overview

Прежде чем приступать к работе с документами, мы должны создать базу данных для их хранения. В нашей базе мы будем хранить сведения о музыкантах, альбомах и композициях. Щелкните по ссылке Create Database... (Создать базу данных). Во всплывающем окне введите имя базы *music* и нажмите Create. В результате вы будете перенаправлены на страницу базы данных. Здесь можно создавать новые документы и открывать существующие.

На странице базы данных *music* щелкните по ссылке New Document (Новый документ). Вы перейдете на страницу, показанную на рис. 25.

² *futon* – футон, традиционная японская постельная принадлежность в виде толстого хлопчатобумажного матраса. *Прим. перев.*

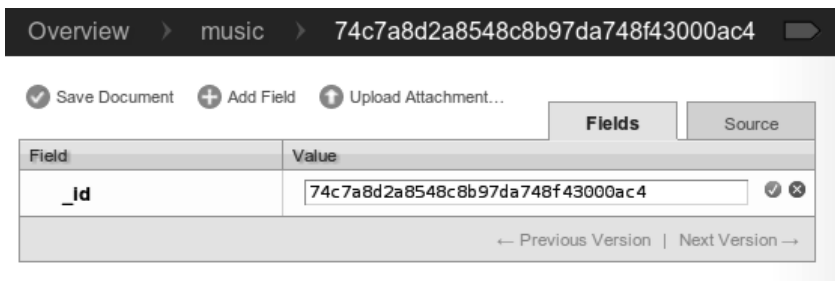


Рис. 25. CouchDB Futon: создание документа

Добро пожаловать в клуб администраторов!

Возможно, вы обратили внимание на предупреждение в правом нижнем углу интерфейса Futon – «Everyone is admin» (Любой пользователь – администратор). Будь это производственный сервер, следующим вашим шагом был бы щелчок по ссылке «Fix this» (Исправить это) и создание пользователя с правами администратора, который может определять, кому что разрешено. Ну а пока оставим всё как есть, поскольку так будет проще решать стоящие перед нами задачи.

Как и в MongoDB, документ представляет собой JSON-объект, состоящий из пар ключ-значение, которые называются *полями*. У любого документа в CouchDB имеется поле `_id`, которое должно быть уникально и никогда не изменяется. Можно задать значение `_id` явно, но если вы этого не сделаете, CouchDB сгенерирует его автоматически. Нас это устраивает, поэтому щелкните по ссылке **Save Document** (Сохранить документ).

Сразу после сохранения CouchDB запишет в документ дополнительное поле `_rev`. Ему присваивается новое значение всякий раз, как в документ вносятся изменения. Структурно это строка, в которой сначала идет целое число, затем дефис, а затем псевдослучайная уникальная строка. Начальное целое число – это номер редакции, в нашем случае 1.

Имена полей, начинающиеся со знака подчеркивания, являются в CouchDB специальными; из них `_id` и `_rev` особенно важны. Чтобы обновить или удалить существующий документ, необходимо указать *как `_id`, так и `_rev`*. Если значения не соответствуют друг другу, CouchDB откажется выполнять операцию. Именно так предотвращаются конфликты – гарантируется, что любая модификация применяется к самой последней версии данных.

В CouchDB нет ни транзакций, ни блокировок. Чтобы модифицировать существующую запись, вы сначала читаете ее и запоминаете значения `_id` и `_rev`. Затем запрашиваете обновление, предоставляя документ целиком, включая поля `_id` и `_rev`. Все операции выполняются строго по порядку – первым пришел, первым обслужен. Требуя согласованности `_rev`, CouchDB гарантирует, что документ, который вы модифицируете, не был за вашей спиной изменен кем-то еще.

Все на той же странице документа щелкните по ссылке Add Field (Добавить поле). В столбце Field введите имя поля *name*, а в столбце Value – значение *The Beatles*. Щелкните по зеленой галочке слева, чтобы подтвердить свое намерение, а затем – по ссылке Save Document (Сохранить документ). Обратите внимание, что теперь поле `_rev` начинается с цифры 2.

CouchDB может хранить не только строки, но и произвольные JSON-структуры с неограниченной вложенностью. Снова щелкните по ссылке Add Field. На этот раз назовите поле *albums*, а в столбце Value введите следующие названия альбомов группы Битлз (список не полный):

```
[
  "Help!",
  "Sgt. Pepper's Lonely Hearts Club Band",
  "Abbey Road"
]
```

После щелчка по ссылке Save Document страница должна выглядеть, как показано на рис. 26.

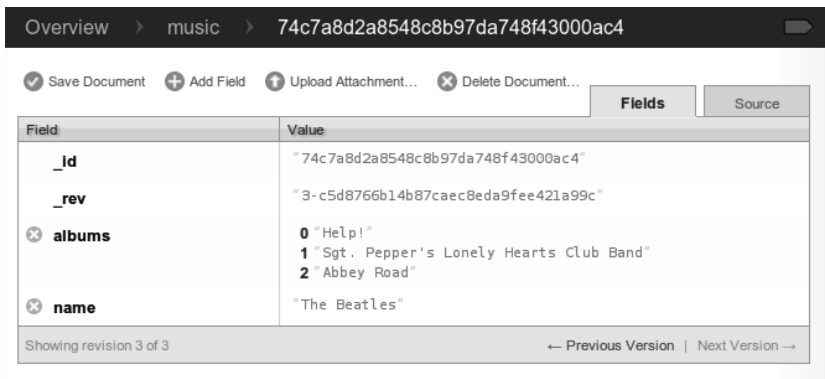


Рис. 26. CouchDB Futon: документ со значением-массивом

Альбом характеризуется не только названием, поэтому давайте введем и дополнительную информацию. Замените значение в поле `albums` таким:

```
[{
  "title": "Help!",
  "year": 1965
}, {
  "title": "Sgt. Pepper's Lonely Hearts Club Band",
  "year": 1967
}, {
  "title": "Abbey Road",
  "year": 1969
}]
```

Теперь после сохранения документа вы сможете раскрыть значение `albums` и показать вложенные документы (рис. 27).



Рис. 27. CouchDB Futon: документ с глубоким уровнем вложенности

Щелчок по ссылке `Delete Document` (Удалить документ) делает то, что и ожидается, — удаляет документ из базы данных `music`. Но пока повремените. Вместо этого давайте перейдем к командной строке и посмотрим, как взаимодействовать с CouchDB через REST-интерфейс.

Выполнение операций CRUD с помощью REST-интерфейса и cURL

Любое взаимодействие с CouchDB основано на REST, и это означает, что команды посылаются только по протоколу HTTP. CouchDB — не

первая из обсуждавшихся нами баз данных, которая обладает этим качеством. Riak (см. главу 3) также использует для общения с клиентами REST-интерфейс. И, как и в случае с Riak, для отправки команд CouchDB можно применить утилиту сURL.

Прежде чем переходить к представлениям, попробуем несколько простых операций CRUD. Для начала введите в окне терминала такую команду:

```
$ curl http://localhost:5984/  
{ "couchdb": "Welcome", "version": "1.1.1" }
```

В ответ на GET-запросы (сURL отправляет запросы таким методом, если явно не указано противное) CouchDB извлекает информацию об объекте, указанном в URL. При обращении к корню сайта, как в данном случае, CouchDB просто информирует о том, что работает, и сообщает номер установленной версии. Теперь получим сведения о созданной ранее базе данных `music` (для удобства чтения результат переформатирован):

```
$ curl http://localhost:5984/music/  
{  
  "db_name": "music",  
  "doc_count": 1,  
  "doc_del_count": 0,  
  "update_seq": 4,  
  "purge_seq": 0,  
  "compact_running": false,  
  "disk_size": 16473,  
  "instance_start_time": "1326845777510067",  
  "disk_format_version": 5,  
  "committed_update_seq": 4  
}
```

В ответ возвращается информация о том, сколько в базе данных документов, как долго работает сервер и сколько операций он уже выполнил.

Чтение документа с помощью GET

Для поиска конкретного документа укажите в URL его `_id` после имени базы данных:

```
$ curl http://localhost:5984/music/  
74c7a8d2a8548c8b97da748f43000ac4  
{  
  "_id": "74c7a8d2a8548c8b97da748f43000ac4",  
  "_rev": "4-93a101178ba65f61ed39e60d70c9fd97",
```

```

"name": "The Beatles",
"albums": [
  {
    "title": "Help!",
    "year": 1965
  }, {
    "title": "Sgt. Pepper's Lonely Hearts Club Band",
    "year": 1967
  }, {
    "title": "Abbey Road",
    "year": 1969
  }
]
}

```

В CouchDB выполнение GET-запросов всегда безопасно, так как никаких изменений в документ не вносится. Если требуется что-то изменить, необходимо использовать другие глаголы HTTP, а именно: PUT, POST и DELETE.

Создание документа с помощью POST

Для создания нового документа используется глагол POST. Не забудьте включить заголовок Content-Type, указав в нем значение *application/json*, иначе CouchDB откажется выполнять запрос.

```

$ curl -i -X POST "http://localhost:5984/music/" \
  -H "Content-Type: application/json" \
  -d '{ "name": "Wings" }'
HTTP/1.1 201 Created
Server: CouchDB/1.1.1 (Erlang OTP/R14B03)
Location: http://localhost:5984/music/
74c7a8d2a8548c8b97da748f43000f1b
Date: Wed, 18 Jan 2012 00:37:51 GMT
Content-Type: text/plain; charset=utf-8
Content-Length: 95
Cache-Control: must-revalidate

{
  "ok": true,
  "id": "74c7a8d2a8548c8b97da748f43000f1b",
  "rev": "1-2fe1dd1911153eb9df8460747dfe75a0"
}

```

Код HTTP-ответа 201 Created означает, что создание завершилось успешно. В теле ответа находится JSON объект с полезной информацией, в частности, значения `_id` и `_rev`.

Обновление документа с помощью PUT

Глагол PUT применяется для обновления существующего документа или создания нового с явно заданным значением `_id`. Как и в случае GET, URL-адрес для PUT должен содержать имя базы данных, за которым следует `_id` документа.

```
$ curl -i -X PUT \
  "http://localhost:5984/music/74c7a8d2a8548c8b97da748f43000f1b" \
  -H "Content-Type: application/json" \
  -d '{
    "_id": "74c7a8d2a8548c8b97da748f43000f1b",
    "_rev": "1-2feldd1911153eb9df8460747dfe75a0",
    "name": "Wings",
    "albums": ["Wild Life", "Band on the Run", "London Town"]
  }'
HTTP/1.1 201 Created
Server: CouchDB/1.1.1 (Erlang OTP/R14B03)
Location: http://localhost:5984/music/
74c7a8d2a8548c8b97da748f43000f1b
Etag: "2-17e4ce41cd33d6a38f04a8452d5a860b"
Date: Wed, 18 Jan 2012 00:43:39 GMT
Content-Type: text/plain; charset=utf-8
Content-Length: 95
Cache-Control: must-revalidate

{
  "ok": true,
  "id": "74c7a8d2a8548c8b97da748f43000f1b",
  "rev": "2-17e4ce41cd33d6a38f04a8452d5a860b"
}
```

В отличие от MongoDB, где модификация документа производится *на месте*, CouchDB при любом изменении перезаписывает документ целиком. Веб-интерфейс Futon создает иллюзию, будто можно изменить единственное поле, но за кулисами при нажатии Save все равно переписывается весь документ.

Как мы уже отмечали, при обновлении документа переданные поля `_id` и `_rev` должны точно соответствовать хранимым, иначе операция завершится ошибкой. Чтобы убедиться в этом, попробуйте выполнить ту же самую операцию PUT еще раз.

```
HTTP/1.1 409 Conflict
Server: CouchDB/1.1.1 (Erlang OTP/R14B03)
Date: Wed, 18 Jan 2012 00:44:12 GMT
Content-Type: text/plain; charset=utf-8
Content-Length: 58
Cache-Control: must-revalidate

{"error": "conflict", "reason": "Document update conflict."}
```

Получаем ответ с кодом 409 Conflict, в теле которого находится JSON-объект с описанием причины ошибки. Таким образом CouchDB обеспечивает согласованность данных.

Удаление документа с помощью DELETE

Наконец, для удаления документа из базы служит глагол DELETE.

```
$ curl -i -X DELETE \
  "http://localhost:5984/music/74c7a8d2a8548c8b97da748f43000f1b" \
  -H "If-Match: 2-17e4ce41cd33d6a38f04a8452d5a860b"
HTTP/1.1 200 OK
Server: CouchDB/1.1.1 (Erlang OTP/R14B03)
Etag: "3-42aafb7411c092614ce7c9f4ab79dc8b"
Date: Wed, 18 Jan 2012 00:45:36 GMT
Content-Type: text/plain;charset=utf-8
Content-Length: 95
Cache-Control: must-revalidate

{
  "ok":true,
  "id":"74c7a8d2a8548c8b97da748f43000f1b",
  "rev":"3-42aafb7411c092614ce7c9f4ab79dc8b"
}
```

Операция DELETE возвращает новый номер редакции, несмотря на то что документа уже нет. Стоит отметить, что на самом деле документ не удаляется с диска, а добавляется новый документ, в котором стоит пометка об удалении старого. Как и в случае обновления, CouchDB не модифицирует документы на месте. Но для всех практических целей документ можно считать удаленным.

День 1: итоги

Теперь, научившись выполнять простые операции CRUD в интерфейсе Futon и с помощью cURL, мы готовы перейти к более сложным темам. Завтра мы займемся созданием индексированных *представлений*, которые открывают другие пути получения документов — не только по значению `_id`.

День 1: домашнее задание

Информационный поиск

1. Найдите онлайнную документацию по HTTP Document API в CouchDB.
2. Мы рассмотрели использование глаголов GET, POST, PUT и DELETE. Какие еще глаголы определены в протоколе HTTP?

Задачи

1. С помощью сURL отправьте PUT-запрос, который поместит в базу данных music новый документ с конкретным `_id` по вашему выбору.
2. С помощью сURL создайте новую базу данных, а затем – также с помощью сURL – удалите ее.
3. Опять же с помощью сURL создайте новый документ, который будет содержать текстовый документ, заданный в виде вложения. И наконец, отправьте (конечно, с помощью сURL) запрос, который вернет этот документ-вложение.

6.3. День 2: создание и опрос представлений

В CouchDB *представление* можно рассматривать как окно в множество хранящихся в базе документов. Представления – это основной способ доступа к документам, если не считать тривиальных случаев, например, отдельных операций CRUD, которые мы рассматривали в первый день. Сегодня мы увидим, как пишутся функции, определяющие представление. Мы также научимся выполнять произвольные запросы к представлениям, применяя сURL. Наконец, мы импортируем данные о музыкантах, чтобы было с чем работать, и продемонстрируем использование популярной написанной на Ruby библиотеки couchrest для доступа к CouchDB.

Доступ к документам через представления

Представление состоит из функций распределения и редукции, которые используются для генерации упорядоченного списка пар ключ-значение. Как ключи, так и значения могут быть произвольными JSON-объектами. Простейшее представление называется `_all_docs`. Оно автоматически предоставляется для любой базы данных и содержит по одной записи для каждого хранящегося документа, причем ключом является строковый идентификатор документа `_id`.

Для получения всех документов из базы выполните GET-запрос к представлению `_all_docs`.

```
$ curl http://localhost:5984/music/_all_docs
{
```

```

"total_rows":1,
"offset":0,
"rows":[{"
  "id":"74c7a8d2a8548c8b97da748f43000ac4",
  "key":"74c7a8d2a8548c8b97da748f43000ac4",
  "value":{"
    "rev":"4-93a101178ba65f61ed39e60d70c9fd97"
  }}
]
}

```

Как видите, в ответ возвращен тот единственный документ, который мы пока создали. Результат представлен в виде JSON-объекта, содержащего массив строк `rows`. Каждая строка – это объект с тремя полями:

- `id` – идентификатор документа `_id`;
- `key` – ключ (JSON-объект), порожденный `mapreduce`-функциями;
- `value` – ассоциированное с ключом значение (JSON-объект), также порождаемое `mapreduce`-функциями.

В случае `_all_docs` поля `id` и `key` совпадают, но для пользовательских представлений так почти никогда не бывает.

По умолчанию представление не включает в возвращенное значение `value` всё содержимое документа. Чтобы выбрать все поля, добавьте в URL параметр `include_docs=true`.

```

$ curl http://localhost:5984/music/_all_docs?include_docs=true
{
  "total_rows":1,
  "offset":0,
  "rows":[{"
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"74c7a8d2a8548c8b97da748f43000ac4",
    "value":{"
      "rev":"4-93a101178ba65f61ed39e60d70c9fd97"
      "name":"The Beatles",
      "albums":[{"
        "title":"Help!",
        "year":1965
      }],{
        "title":"Sgt. Pepper's Lonely Hearts Club Band",
        "year":1967
      },{
        "title":"Abbey Road",
        "year":1969
      }}
    ]}
}

```

```
    }  
  }  
}
```

Как видите, теперь в объект `value` включены дополнительные свойства `name` и `albums`. Помня об этой базовой структуре, приступим к созданию собственных представлений.

Создание первого представления

Понимая в общих чертах, как работают представления, попробуем создать свое собственное. Для начала воспроизведем поведение представления `_all_docs`, а затем усложним задачу и будем извлекать из документов данные, расположенные на более глубоких уровнях, для индексирования.

Чтобы выполнить временное представление, откройте в браузере интерфейс Futon³, как мы делали в первый день. Затем откройте базу данных `music`, щелкнув по соответствующей ссылке. Из раскрывающегося списка View в правом верхнем углу страницы выберите пункт «Temporary view...» (рис. 28).

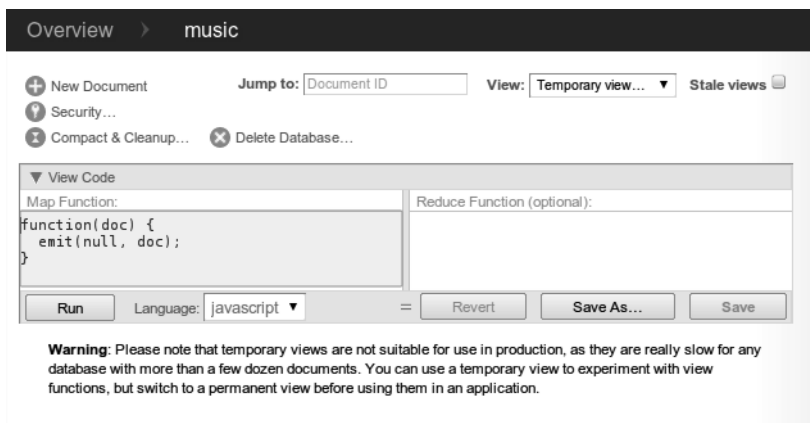


Рис. 28. CouchDB Futon: временное представление

В левой области Map Function будет присутствовать такой код:

```
function(doc) {  
  emit(null, doc);  
}
```

Если сейчас нажать расположенную под этой областью кнопку Run, то CouchDB выполнит эту функцию по одному разу для каждо-

³ http://localhost:5984/_utils/



го документа в базе, передавая текущий документ в параметре `doc`. В результате будет сгенерирована таблица, содержащая по одной строке для каждого документа:

Ключ	Значение
<code>null</code>	<code>{_id: "74c7a8d2a8548c8b97da748f43000ac4", _rev: "4-93a101178ba65f61ed39e60d70c9fd97", name: "The Beatles", albums: [{title: "Help!", year: 1965}, {title: "Sgt. Pepper's Lonely Hearts Club Band", year: 1967}, {title: "Abbey Road", year: 1969}]}</code>

Секрет заключается в функции `emit()` (которая работает аналогично одноименной функции в MongoDB). Эта функция принимает два аргумента: ключ и значение. Функция распределения может вызывать `emit` для одного документа нуль, один или много раз. В данном случае распределитель эмитирует пару `null/doc`. Как видно из таблицы, ключ действительно равен `null`, а значение – тот самый объект, который мы видели вчера, когда запрашивали документ напрямую с помощью `sURL`.

Но чтобы добиться того же результата, что `_all_docs`, распределитель должен эмитировать нечто иное. Напомним, что `_all_docs` эмитирует поле `_id` документа в качестве ключа и простой объект, содержащий только поле `_rev` – в качестве значения. Поэтому измените код в области Map Function, как показано ниже, и нажмите кнопку Run.

```
function(doc) {
  emit(doc._id, { rev: doc._rev });
}
```

Теперь будет возвращена таблица с той же парой ключ-значение, которую мы видели раньше, когда перебирали записи с помощью `_all_docs`:

Ключ	Значение
<code>"74c7a8d2a8548c8b97da748f43000ac4"</code>	<code>{rev: "4-93a101178ba65f61ed39e60d70c9fd97"}</code>

Отметим, что для выполнения временных представлений Futon не обязателен. Можно вместо этого отправить POST-запрос обработчику `_temp_view`, а в теле запроса передать функцию-распределитель в виде JSON-объекта.

```
$ curl -X POST \
  http://localhost:5984/music/_temp_view \
```

```
-H "Content-Type: application/json" \
-d '{"map": "function(doc) {emit(doc._id, {rev: doc._rev});}"}'
{
  "total_rows": 1,
  "offset": 0,
  "rows": [{
    "id": "74c7a8d2a8548c8b97da748f43000ac4",
    "key": "74c7a8d2a8548c8b97da748f43000ac4",
    "value": {
      "rev": "4-93a101178ba65f61ed39e60d70c9fd97"
    }
  }]
}
```

Ответ точно такой же, как при использовании `_all_docs`. А что будет, если добавить параметр `include_docs=true`? Попробуем!

```
$ curl -X POST \
http://localhost:5984/music/_temp_view?include_docs=true \
-H "Content-Type: application/json" \
-d '{"map": "function(doc) {emit(doc._id, {rev: doc._rev});}"}'
{
  "total_rows": 1,
  "offset": 0,
  "rows": [{
    "id": "74c7a8d2a8548c8b97da748f43000ac4",
    "key": "74c7a8d2a8548c8b97da748f43000ac4",
    "value": {
      "rev": "4-93a101178ba65f61ed39e60d70c9fd97"
    },
    "doc": {
      "_id": "74c7a8d2a8548c8b97da748f43000ac4",
      "_rev": "4-93a101178ba65f61ed39e60d70c9fd97",
      "name": "The Beatles",
      "albums": [...]
    }
  }]
}
```

На этот раз дополнительные поля не включаются в объект `value`, а в строку добавляется свойство `doc`, в котором содержится весь документ целиком.

Представление может эмитировать любое значение, в том числе `null`. Создание отдельного свойства `doc` предотвращает проблемы, которые могли бы возникнуть в результате объединения `value` с документом. Далее мы покажем, как сохранить представление, чтобы CouchDB могла проиндексировать результаты.

Сохранение представления в виде проектного документа

Исполняя временное представление, CouchDB должна применить переданный распределитель к *каждому* документу в базе данных. Эта процедура требует огромного количества ресурсов, потребляет много процессорного времени и вообще работает медленно. Временные представления лучше использовать только на этапе разработки. В производственной системе представления следует сохранять в виде проектных *документов* (*design documents*).

Проектный документ хранится в базе, как обычный документ – такой же, как созданный нами ранее документ о группе Битлз. А, стало быть, его можно показывать в представлениях и реплицировать на другие серверы CouchDB. Чтобы сохранить временное представление в виде проектного документа в Futon, нажмите кнопку Save As... (Сохранить как), а затем заполните поля Design Document и View Name (Имя представления).

Идентификаторы проектных документов всегда начинаются строкой `_design/`. Один проектный документ может содержать одно или несколько представлений. Представления, хранящиеся в одном проектном документе, различаются по именам. Какое представление в какой проектный документ помещать, зависит от приложения, и решение отдается на откуп разработчику. Вообще говоря, представления рекомендуются группировать в соответствии с тем, как они соотносятся с данными. Примеры будут приведены ниже, когда мы начнем рассматривать более интересные представления.

Поиск исполнителей по имени

Итак, с принципами создания представлений мы познакомились, теперь давайте разработаем представление для конкретного приложения. Напомним, что в базе данных `music` хранится информация о музыкантах, в том числе название группы в поле `name`. Обратившись к представлению `_all_docs` с обычным GET-запросом, мы сможем получить документ по значению `_id`, но нас больше интересует поиск групп по названию.

Иными словами, сейчас мы умеем находить документ с `_id`, равным `74c7a8d2a8548c8b97da748f43000ac4`, но как найти документ, у которого в поле `name` хранится строка *The Beatles*? Для этого нужно представление. В Futon вернитесь на страницу Temporary View, введите в область Map Function показанный ниже код и нажмите кнопку Run.

```
couchdb/artists_by_name_mapper.js
function(doc) {
  if ('name' in doc) {
    emit(doc.name, doc._id);
  }
}
```

Эта функция проверяет, имеется ли в текущем документе поле `name` и, если да, то эмитирует его значение и идентификатор `_id` в виде пары ключ-значение. В результате получится таблица вида:

Ключ	Значение
"The Beatles"	"74c7a8d2a8548c8b97da748f43000ac4"

Нажмите кнопку Save As... и введите в поле Design Document значение *artists*, а в поле View Name – значение *by_name*. Нажмите Save, чтобы сохранить изменения.

Поиск альбомов по названию

Умение искать исполнителей по названию – вещь полезная, но зачем на этом останавливаться? Давайте еще напишем представление для поиска альбомов. Это будет первый пример, когда функция-распределитель эмитирует несколько результатов для одного документа.

Снова перейдите на страницу Temporary View и введите такой код распределителя:

couchdb/albums_by_name_mapper.js

```
function(doc) {
  if ('name' in doc && 'albums' in doc) {
    doc.albums.forEach(function(album) {
      var
        key = album.title || album.name,
        value = { by: doc.name, album: album };
      emit(key, value);
    });
  }
}
```

Эта функция проверяет, есть ли в текущем документе поля `name` и `albums`. Если да, то она эмитирует пару ключ-значение для каждого альбома, причем ключом является название альбома, а значением – составной объект, содержащий имя исполнителя (или название группы) и объект, представляющий сам альбом. Получается такая таблица:



Ключ	Значение
"Abbey Road"	{by: "The Beatles", album: {title: "Abbey Road", year: 1969}}
"Help!"	{by: "The Beatles", album: {title: "Help!", year: 1965}}
"Sgt. Pepper's Lonely Hearts Club Band"	{by: "The Beatles", album: {title: "Sgt. Pepper's Lonely Hearts Club Band", year: 1967}}

Как и раньше, нажмите кнопку Save As..., только теперь в поле Design Document введите *albums*, а в поле View Name – *by_name*. Нажмите Save, чтобы сохранить изменения. Теперь посмотрим, как опрашивать такие документы.

Опрос представлений исполнителей и альбомов

Теперь, когда у нас есть два проектных документа, вернемся в окно терминала и попробуем запросить их с помощью curl. Начнем с представления исполнителей по именам. Введите такую команду:

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name
{
  "total_rows":1,
  "offset":0,
  "rows":[{"
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"The Beatles",
    "value":"74c7a8d2a8548c8b97da748f43000ac4"
  }]}
}
```

URL для опроса представления имеет вид `path /<database_name>/_design/<design_doc>/_view/<view_name>`, где вместо строк в угловых скобках подставляются имена проектного документа и представления. В данном случае мы опрашиваем представление *by_name* в проектном документе *artists*, который хранится в базе данных *music*. Неудивительно, что в результате возвращается единственный документ с ключом, содержащим название группы.

А теперь попробуем опросить представление альбомов по названию:

```
$ curl http://localhost:5984/music/_design/albums/_view/by_name
{
  "total_rows":3,
```



```

"offset":0,
"rows":[{
  "id":"74c7a8d2a8548c8b97da748f43000ac4",
  "key":"Abbey Road",
  "value":{
    "by":"The Beatles",
    "album":{
      "title":"Abbey Road",
      "year":1969
    }
  }
},{
  "id":"74c7a8d2a8548c8b97da748f43000ac4",
  "key":"Help!",
  "value":{
    "by":"The Beatles",
    "album":{
      "title":"Help!",
      "year":1965
    }
  }
},{
  "id":"74c7a8d2a8548c8b97da748f43000ac4",
  "key":"Sgt. Pepper's Lonely Hearts Club Band",
  "value":{
    "by":"The Beatles",
    "album":{
      "title":"Sgt. Pepper's Lonely Hearts Club Band",
      "year":1967
    }
  }
}]
}

```

CouchDB гарантирует, что записи отсортированы в алфавитно-цифровом порядке по эмитированному ключам. По существу, это результат индексирования, выполненного CouchDB. При проектировании представлений важно выбирать эмитируемые ключи, так чтобы порядок сортировки был осмысленным. Показанный выше запрос возвращает все содержимое представления, но что, если нам нужно только его подмножество? Добиться этого можно, например, добавив в URL параметр `key`. В этом случае возвращаются только строки с заданным ключом.

```

$ curl 'http://localhost:5984/music/_design/albums/_view/by_name? \
key="Help!'"
{
  "total_rows":3,
  "offset":1,

```

```
"rows": [{
  "id": "74c7a8d2a8548c8b97da748f43000ac4",
  "key": "Help!",
  "value": {
    "by": "The Beatles",
    "album": { "title": "Help!", "year": 1965 }
  }
}]
}
```

Обратите внимание на поля `total_rows` и `offset`. В поле `total_rows` находится общее количество записей в представлении (не только тех, что возвращены в ответ на запрос). Поле `offset` показывает смещение первой возвращенной записи от начала полного набора. Зная эти два числа и размер массива `rows`, мы можем вычислить, сколько записей осталось слева и справа от выбранной порции. Выборку из представлений можно осуществлять не только по параметру `key`, но для демонстрации нам понадобится больше данных.

Импорт данных в CouchDB с помощью программы на Ruby

Импорт данных – задача, которая неизменно встает вне зависимости от используемой СУБД. И CouchDB – не исключение. В этом разделе мы напишем на Ruby программу импорта структурированных данных в базу `music`. Это, во-первых, даст возможность познакомиться с массовым импортом в CouchDB, а, во-вторых, мы получим достаточно большой набор данных для изучения более сложных представлений.

Мы воспользуемся данными о музыкальных произведениях с сайта `Jamendo.com`⁴, на котором размещается музыка с бесплатной лицензией. Jamendo представляет данные об исполнителях, альбомах и композициях в формате XML, идеальном для импорта в документо-ориентированную базу типа CouchDB.

Перейдите на страницу `NewDatabaseDumps`⁵ сайта Jamendo и скачайте файл `dbdump_artistalbumtrack.xml.gz`⁶. Этот архив «весит» всего около 15 МБ. Для разбора XML-файла мы воспользуемся gem-пакетом `libxml-ruby`.

Вместо того чтобы писать собственный драйвер Ruby-CouchDB или напрямую формировать HTTP-запросы, мы прибегнем к

4 <http://www.jamendo.com/>

5 <http://developer.jamendo.com/en/wiki/NewDatabaseDumps>

6 http://img.jamendo.com/data/dbdump_artistalbumtrack.xml.gz

популярному gem-пакету `couchrest`, который обертывает вызовы удобным API на Ruby. Нам понадобится всего несколько методов из этого API, но к услугам желающих использовать драйвер в собственных проектах имеется отличная документация⁷.

Установите необходимые gem-пакеты, выполнив следующую команду:

```
$ gem install libxml-ruby couchrest
```

Как и при разборе данных из википедии в главе 4, мы будем использовать SAX-анализатор, который последовательно читает и вставляет документы в базу, как если бы они поступали из стандартного ввода. Код программы приведен ниже.

couchdb/import_from_jamendo.rb

```
❶ require 'rubygems'
   require 'libxml'
   require 'couchrest'

   include LibXML

❷ class JamendoCallbacks
   include XML::SaxParser::Callbacks
❸   def initialize()
     @db = CouchRest.database!("http://localhost:5984/music")
     @count = 0
     @max = 100 # максимальное количество вставляемых документов
     @stack = []
     @artist = nil
     @album = nil
     @track = nil
     @tag = nil
     @buffer = nil
   end

❹   def on_start_element(element, attributes)
     case element
     when 'artist'
       @artist = { :albums => [] }
       @stack.push @artist
     when 'album'
       @album = { :tracks => [] }
       @artist[:albums].push @album
       @stack.push @album
     when 'track'
       @track = { :tags => [] }
```

7 <http://rdoc.info/github/couchrest/couchrest/master/>

```

        @album[:tracks].push @track
        @stack.push @track
    when 'tag'
        @tag = {}
        @track[:tags].push @tag
        @stack.push @tag
    when 'Artists', 'Albums', 'Tracks', 'Tags'
        # игнорировать
    else
        @buffer = []
    end
end

5 def on_characters(chars)
    @buffer << chars unless @buffer.nil?
end

6 def on_end_element(element)
    case element
    when 'artist'
        @stack.pop
        @artist['_id'] = @artist['id'] # используем идентификатор,
                                     # присвоенный Jamendo
                                     # исполнителю, в качестве
                                     # _id документа

        @artist[:random] = rand
        @db.save_doc(@artist, false, true)
        @count += 1
        if !@max.nil? && @count >= @max
            on_end_document
        end
        if @count % 500 == 0
            puts " #{@count} records inserted"
        end
    when 'album', 'track', 'tag'
        top = @stack.pop
        top[:random] = rand
    when 'Artists', 'Albums', 'Tracks', 'Tags'
        # игнорировать
    else
        if @stack[-1] && @buffer
            @stack[-1][element] = @buffer.join.force_encoding('utf-8')
            @buffer = nil
        end
    end
end

def on_end_document()
    puts "TOTAL: #{@count} records inserted"
    exit(1)
end

```

```
end
end
```

```
7 parser = XML::SaxParser.io(ARGF)
  parser.callbacks = JamendoCallbacks.new
  parser.parse
```

- ❶ Первым делом мы импортируем модуль `rubygems` и необходимые нам `gem`-пакеты.
- ❷ Стандартный способ работы с пакетом `LibXML` подразумевает создание класса обратного вызова. Здесь мы определяем класс `JamendoCallbacks`, который инкапсулирует обработчики различных событий SAX.
- ❸ На этапе инициализации наш класс сначала подключается к локальному серверу CouchDB, пользуясь `CouchRest` API, а затем создает базу данных `music` (если ее еще не существует). Затем инициализируются различные переменные экземпляра для хранения информации о состоянии разбора. Отметим, что если присвоить параметру `@max` значение `nil`, то будут импортированы все документы, а не только первые 100.
- ❹ Во время разбора метод `on_start_element()` будет обрабатывать все открывающие теги. Нас интересуют только теги `<artist>`, `<album>`, `<track>` и `<tag>`. Некоторые контейнерные элементы – `<Artists>`, `<Albums>`, `<Tracks>` и `<Tags>` – мы явно пропускаем, а все остальные трактуем как свойства, устанавливаемые для ближайшего контейнера.
- ❺ Распознанные анализатором символьные данные мы буферизуем и впоследствии добавляем в качестве свойства текущего контейнерного элемента (находящегося в конце массива `@stack`).
- ❻ Самое интересное происходит в методе `on_end_element()`. Здесь мы закрываем текущий контейнерный элемент, выталкивая его из стека. Если обнаружен закрывающий тег для элемента `<artist>`, то мы сохраняем документ в CouchDB, вызывая метод `@db.save_doc()`. Кроме того, для любого контейнерного элемента добавляется свойство `random`, содержащее сгенерированное случайное число. Впоследствии мы воспользуемся им для выбора случайной композиции, альбома или исполнителя.
- ❼ В потоке ARGV Ruby объединяет стандартный ввод и все файлы, указанные в командной строке. Мы передаем этот поток пакету `LibXML` и указываем экземпляр класса `JamendoCallbacks`, который будет обрабатывать распознанные лексемы: открывающие и закрывающие теги, а также символьные данные.

При запуске скрипта импорта передаем ему по конвейеру результат распаковки архива с XML-файлом:

```
$ zcat dbdump_artistalbumtrack.xml.gz | ruby import_from_jamendo.rb
TOTAL: 100 records inserted
```

По завершении импорта посмотрим, как теперь выглядят наши представления. Сначала отберем несколько исполнителей. Параметр

limit в URL говорит, что мы хотим получить не более указанного в нем количества документов.

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name?limit=5
{"total_rows":100,"offset":0,"rows":[
{"id":"370255","key":"\\\"ATTIC\\\"","value":"370255"},
{"id":"353262","key":"10centSunday","value":"353262"},
{"id":"367150","key":"abdielyromero","value":"367150"},
{"id":"276","key":"AdHoc","value":"276"},
{"id":"364713","key":"Adversus","value":"364713"}
]}
```

Этот запрос начинает отбор с начала списка исполнителей. Чтобы начать с середины, нужно задать параметр startkey:

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name?
limit=5&startkey=%22C%22
{"total_rows":100,"offset":16,"rows":[
{"id":"340296","key":"CalexB","value":"340296"},
{"id":"353888","key":"carsten may","value":"353888"},
{"id":"272","key":"Chroma","value":"272"},
{"id":"351138","key":"Compartir D\u00f3na Gustet","value":"351138"},
{"id":"364714","key":"Daringer","value":"364714"}
]}
```

Выше мы начали с исполнителей, имена которых начинаются с буквы С. Параметр endkey дает еще один способ ограничить выборку. Ниже мы запрашиваем исполнителей с именами, начинающимися с букв С и D:

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name?
startkey=%22C%22&endkey=%22D%22
{"total_rows":100,"offset":16,"rows":[
{"id":"340296","key":"CalexB","value":"340296"},
{"id":"353888","key":"carsten may","value":"353888"},
{"id":"272","key":"Chroma","value":"272"},
{"id":"351138","key":"Compartir D\u00f3na Gustet","value":"351138"}
]}
```

Чтобы отсортировать строки в обратном порядке, воспользуемся параметром descending. Не забудьте только поменять местами startkey и endkey.

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name?
startkey=%22D%22&endkey=%22C%22&descending=true
{"total_rows":100,"offset":16,"rows":[
{"id":"351138","key":"Compartir D\u00f3na Gustet","value":"351138"},
{"id":"272","key":"Chroma","value":"272"},
{"id":"353888","key":"carsten may","value":"353888"},
{"id":"340296","key":"CalexB","value":"340296"}
]}
```

Есть еще ряд задаваемых в URL параметров, которые модифицируют запросы к представлениям, но описанные наиболее употребительны. Некоторые параметры имеют отношение к группировке, являющейся частью фазы редукции в CouchDB. Их мы рассмотрим завтра.

День 2: итоги

Сегодня мы рассмотрели обширный материал. Мы научились создавать в CouchDB простые представления и сохранять их в виде проектных документов. Мы познакомились с разными способами опроса представлений для получения выборки из индексированного содержимого. Воспользовавшись языком Ruby и популярным gem-пакетом `couchrest`, мы импортировали структурированные данные и на их основе построили представления. Завтра мы разовьем эти идеи и создадим более сложные представления, добавив редукторы. А затем перейдем к другим API, которые поддерживает CouchDB.

День 2: домашнее задание

Информационный поиск

1. Мы видели, что метод `emit()` может выводить строковые ключи. А какие еще типы он поддерживает? Что произойдет, если эмитировать массив в качестве ключа?
2. Найдите описание перечня поддерживаемых параметров в URL (типа `limit` и `startkey`) и выясните, для чего они предназначены.

Задачи

1. Скрипт `import_from_jamendo.rb` сопоставил каждому исполнителю случайное число, записав его в свойство `random`. Напишите функцию-распределитель, которая эмитирует пары ключ-значение, где ключом является это случайное число, а значением – название группы. Сохраните ее в новом проектном документе `_design/random` под именем `artist`.
2. Сформулируйте в cURL запрос, который будет возвращать случайного исполнителя.

Подсказка: нужно будет использовать параметр `startkey` и сгенерировать в командной строке случайное число с помощью команды ``ruby -e 'puts rand'``.

3. Скрипт импорта сформировал также свойство `random` для каждого альбома, композиции и тега. Создайте в проектном документе `_design/random` еще три представления с именами `album`, `track` и `tag` по образцу ранее созданного представления `artist`.

6.4. День 3: более сложные представления, Changes API и репликация данных

В предыдущие два дня мы научились выполнять простые операции CRUD и с помощью представлений искать данные. Сегодня мы продолжим эту тему и разберемся с фазой редукции в технологии `mapreduce`. Затем мы разработаем несколько приложений на JavaScript для сервера `Node.js`, чтобы изучить особенности API обновления, предоставляемого `CouchDB`. И напоследок обсудим репликацию и механизм разрешения конфликтов.

Создание более сложных представлений с помощью редукторов

Основанные на технологии `mapreduce` представления позволяют воспользоваться встроенными в `CouchDB` средствами индексирования и агрегирования. Все созданные вчера представления содержали только распределители. А сегодня мы подключим редукторы и применим новые знания к данным, импортированным с сайта `Jamendo`.

Данные с сайта `Jamendo` обладают замечательной особенностью – глубокой вложенностью. У исполнителей есть альбомы, у альбомов – композиции, а у композиций – атрибуты, в том числе теги. Сейчас мы займемся тегами и посмотрим, как написать представление для их выборки и подсчета.

Вернитесь на страницу `Temporary View` и введите следующую функцию-распределитель.

couchdb/tags_by_name_mapper.js

```
function(doc) {
  (doc.albums || []).forEach(function(album) {
    (album.tracks || []).forEach(function(track) {
      (track.tags || []).forEach(function(tag) {
        emit(tag.idstr, 1);
      });
    });
  });
}
```



```
    });  
  });  
});  
}
```

Эта функция заходит внутрь документа, описывающего исполнителя, затем внутрь каждого альбома, каждой композиции и, наконец, каждого тега. Для каждого тега она эмитирует пару ключ-значение, состоящую из свойства `idstr` тега (которое содержит его строковое представление, например `"rock"`) и числа 1.

После этого введите следующий код в область Reduce Function:

couchdb/simple_count_reducer.js

```
function(key, values, rereduce) {  
  return sum(values);  
}
```

Эта функция просто суммирует все числа в списке `values`, о котором мы поговорим чуть ниже – после запуска этого представления. Нажмите кнопку Run. В результате будет выведена такая таблица:

Ключ	Значение
"17sonsrecords"	1
"17sonsrecords"	1
"17sonsrecords"	1
"17sonsrecords"	1
"17sonsrecords"	1
"acid"	1
"acousticguitar"	1
"acousticguitar"	1
"action"	1
"action"	1

Ничего удивительного. Значение всюду равно 1, поскольку так его сформировал распределитель, а один и тот же ключ повторяется столько раз, сколько встречается в композициях. Однако обратите внимание на флажок Reduce в правом верхнем углу выведенной таблицы. Отметьте его и снова посмотрите на таблицу. Теперь она выглядит так:

Ключ	Значение
"17sonsrecords"	5
"acid"	1
"acousticguitar"	2
"action"	2
"adventure"	3
"aksband"	1
"alternativ"	1
"alternativ"	3
"ambient"	28
"autodidacta"	17

Что произошло? Если коротко, то редуктор *редуцировал* результат, объединив одинаковые строки так, как диктует функция редукции. Механизм `mapreduce` в CouchDB концептуально устроен так же, как в других рассмотренных нами базах данных (Riak и MongoDB). Точнее, CouchDB выполняет следующие шаги при построении представления.

1. Отправить документы функции-распределителю.
2. Собрать все эмитированные значения.
3. Отсортировать эмитированные строки по ключам.
4. Отправить группы строк с одинаковыми ключами функции-редуктору.
5. Если данных слишком много для редуцирования за один вызов, вызвать редуктор еще раз, но уже с ранее редуцированными значениями.
6. Повторять рекурсивные вызовы редуктора столько раз, сколько необходимо, до тех пор, пока не останется повторяющихся ключей.

Функции-редукторы в CouchDB принимают три аргумента: `key`, `values` и `rereduce`. В аргументе `key` передается массив кортежей; каждый кортеж – это массив из двух элементов: ключа, эмитированного распределителем, и поля `_id` того документа, по которому этот ключ порожден. В аргументе `values` передается массив значений, соответствующих ключам.

Третий аргумент, `rereduce` – булевское значение, равное `true`, если данный вызов является *повторной редукцией*, то есть ключи и значения поступили не от распределителя, а в результате предыду-

щих вызовов редукторов. В таком случае параметр `key` будет равен `null`.

По следам вызовов редукторов

Давайте внимательно изучим показанный выше пример вывода.

Рассмотрим документы (исполнителей), в которых есть композиции, помеченные тегом «ambient». Распределители обрабатывают документы и эмитируют пары ключ-значение вида «ambient»/1.

В какой-то момент эмитированных пар набирается достаточно для того, чтобы CouchDB вызвала редуктор. Этот вызов может выглядеть следующим образом:

```
reduce(  
  [{"ambient", id1}, {"ambient", id2}, ...], // ключи одинаковые  
  [1, 1, ...],                             // все значения равны 1  
  false                                     // rereduce равно false  
)
```

Напомним, что наш редуктор выполняет суммирование значений: вызывает функцию `sum()`, передавая в параметре `values`. Поскольку все значения равны 1, то их суммой будет просто количество композиций, имеющих тег «ambient». CouchDB сохраняет возвращенное значение для последующей обработки. Для определенности положим, что оно равно 10.

Позже, после нескольких таких вызовов, CouchDB решает объединить промежуточные результаты редуцирования, выполнив повторную редукцию:

```
reduce(  
  null, // массив key равен null  
  [10, 10, 8], // values - результаты прешествующих вызовов редуктора  
  true        // rereduce равно true  
)
```

Наш редуктор снова суммирует значения. На этот раз получается сумма 28. Повторные вызовы редукторов могут быть рекурсивными. Всё это продолжается, пока есть что редуцировать, и в итоге все промежуточные значения будут редуцированы в одно.

Большинство `mapreduce`-систем, в том числе входящие в рассмотренные выше базы данных Riak и MongoDB, отбрасывают результаты распределителей и редукторов по завершении работы. В таких системах `mapreduce` рассматривается как средство к достижению цели — нечто такое, что делается, когда возникает необходимость, начиная каждый раз заново. В CouchDB дело обстоит иначе.

После того как представление сохранено в проектном документе, CouchDB сохраняет промежуточные результаты распределителей и редукторов до тех пор, пока какое-то изменение документа не сделает их недействительными. В этот момент CouchDB инкрементно прогоняет распределители и редукторы, чтобы скорректировать данные. Но не вычисляет всё заново с самого начала. В этом и состоит суть представлений CouchDB – благодаря тому, что промежуточные данные не отбрасываются, технология mapreduce используется как механизм первичного индексирования.

Отслеживание изменений в CouchDB

Инкрементный подход CouchDB к mapreduce – безусловно, новаторская идея, одна из многих особенностей, выделяющих CouchDB среди прочих СУБД. Далее мы рассмотрим еще одну: Changes API – интерфейс, позволяющий отслеживать изменения в базе данных и мгновенно получать новые значения.

Changes API делает CouchDB идеальным кандидатом на роль системы, куда записываются оригинальные данные. Представьте себе систему из нескольких СУБД, в которую данные стекаются из разных источников, причем все базы должны поддерживаться в актуальном состоянии (нечто подобное мы построим в разделе 8.4 «Комбинирование с другими базами данных»). В качестве примера можно назвать поисковую систему на основе Lucene или ElasticSearch или слой кэширования, реализованный с помощью memcached или Redis. Кроме того, в ответ на изменения могут запускаться различные административные скрипты – например, для сжатия базы или удаленного резервного копирования. Короче говоря, этот простой API открывает массу возможностей. Сегодня мы научимся использовать его себе во благо.

Для демонстрации API мы разработаем несколько простых клиентских приложений, используя Node.js⁸ – платформу для исполнения JavaScript-кода на стороне сервера, построенную на основе движка JavaScript V8 – того самого, который встроен в браузер Google Chrome. Поскольку в Node.js применяется событийно-управляемая модель, а код пишется на JavaScript, то он вполне естественно интегрируется с CouchDB. Если на вашей машине Node.js еще не установлен, зайдите на сайт продукта и скачайте последнюю стабильную версию (мы работаем с версией 0.6).

Существует три разновидности Changes API: опрос, долгий опрос и непрерывное отслеживание. Рассмотрим их по очереди. Как обычно,

8 <http://nodejs.org/>

начнем с сURL, чтобы быть «ближе к железу», а затем поговорим о доступе из программы.

Отслеживание изменений с помощью сURL

Простейший способ обратиться к Changes API – воспользоваться интерфейсом опроса. Выполните следующую команду (мы сократили вывод, на вашей машине результаты могут быть другими).

```
$ curl http://localhost:5984/music/_changes
{
  "results": [{
    "seq": 1,
    "id": "370255",
    "changes": [{"rev": "1-a7b7cc38d4130f0a5f3eae5d2c963d85"}]
  }, {
    "seq": 2,
    "id": "370254",
    "changes": [{"rev": "1-2c7e0deec3ffca959ba0169b0e8bfcef"}]
  }, {
    ... еще 97 записей ...
  }, {
    "seq": 100,
    "id": "357995",
    "changes": [{"rev": "1-aa649aa53f2858cb609684320c235aee"}]
  }],
  "last_seq": 100
}
```

В ответ на GET-запрос к URL `_changes` без параметров CouchDB возвращает всё, что имеет. Но, как и при доступе к представлениям, можно задать параметр `limit`, чтобы получить подмножество данных, а параметр `include_docs=true` заставляет вернуть полные документы.

Как правило, нам ни к чему все изменения, имевшие место с начала времен. Интереснее те изменения, которые произошли с момента последней проверки. Для этого предназначен параметр `since`.

```
$ curl http://localhost:5984/music/_changes?since=99
{
  "results": [{
    "seq": 100,
    "id": "357995",
    "changes": [{"rev": "1-aa649aa53f2858cb609684320c235aee"}]
  }],
  "last_seq": 100
}
```

Если значение `since` больше последнего порядкового номера, то будет возвращен пустой ответ:

```
$ curl http://localhost:5984/music/_changes?since=9000
{
  "results": [
  ],
  "last_seq": 9000
}
```

Таким способом клиентское приложение может периодически запрашивать новые изменения и предпринимать соответствующие действия.

Опрос – вещь хорошая, если приложение может смириться с запаздывающей реакцией на обновления, например, в случае, когда обновления происходят сравнительно редко. Так, опрос новых записей в блоге раз в пять минут может оказаться вполне приемлемым решением.

Если же получать обновления требуется быстрее, не тратя время на повторное открытие соединения, то лучше подойдет долгий опрос (long polling). Если добавить в URL параметр `feed=longpoll`, то CouchDB оставит соединение открытым на некоторое время, ожидая, что могут произойти какие-то изменения, о которых следует сообщить в ответе. Попробуйте такой запрос:

```
$ curl 'http://localhost:5984/music/_changes?feed=longpoll&since=9000'
{"results": [
```

Как видите, сервер вернул начало JSON-ответа и больше ничего. Если оставить терминал открытым, то в конце концов CouchDB закроет соединение и закончит ответ:

```
],
  "last_seq": 9000}
```

С точки зрения разработки, написание драйвера, которые наблюдает за изменениями в базе данных CouchDB с помощью опроса, эквивалентно написанию драйвера, реализующего долгий опрос. Разница лишь в том, сколько времени CouchDB будет держать соединение открытым. А теперь перейдем к написанию приложения для Node.js, которое будет следить за «лентой» изменений.

Опрос изменений в приложении для Node.js

Node.js – система, управляемая событиями, поэтому и наблюдатель для CouchDB будет придерживаться того же принципа. Драйвер будет следить за изменениями и генерировать события, получив от CouchDB измененные документы. Для начала покажем общую структуру драйвера, обсудим ее основные компоненты, а затем восполним детали, относящиеся к конкретной ленте – потоку изменений.

Итак, вот как устроена наша программа-наблюдатель.

couchdb/watch_changes_skeleton.js

```
var
  http = require('http'),
  events = require('events');

/**
 * создать наблюдатель за CouchDB, применяя параметры соединения;
 * следует паттерну EventEmitter в node.js, генерирует события 'change'.
 */
❶ exports.createWatcher = function(options) {

  ❷  var watcher = new events.EventEmitter();

  watcher.host = options.host || 'localhost';
  watcher.port = options.port || 5984;
  watcher.last_seq = options.last_seq || 0;
  watcher.db = options.db || '_users';

  ❸  watcher.start = function() {
    // ... детали, относящиеся к ленте обновлений
  };
  return watcher;
};

// начать наблюдение за изменениями в CouchDB, если запущен как
// главный скрипт
❹  if (!module.parent) {
  exports.createWatcher({
    db: process.argv[2],
    last_seq: process.argv[3]
  })
  .on('change', console.log)
  .on('error', console.error)
  .start();
}
```

- ❶ Экспортирует стандартный объект, предоставляемый CommonJS Module API, реализованного в Node.js. Добавление метода `createWatcher()` в объект `exports` делает его доступным другим скриптам Node.js, то есть он становится частью библиотеки. Аргумент `options` позволяет вызывающей программе указать, к какой базе данных подключаться и задать другие параметры соединения.
- ❷ Метод `createWatcher()` порождает объект `EventEmitter`, с помощью которого вызывающая программа может прослушивать события изменения. Метод `on()` объекта `EventEmitter` позволяет подписаться на

внешнее событие и генерировать внутреннее событие путем вызова метода `emit()`.

- ❸ Метод `watcher.start()` отвечает за отправку HTTP-запросов для наблюдения за изменениями в CouchDB. Когда произойдет изменение какого-либо документа, наблюдатель оповестит об этом программу путем генерации события изменения. Все специфические детали реализации должны быть помещены сюда.
- ❹ Это блок описывает, что должен делать скрипт, если он вызван непосредственно из командной строки. В данном случае скрипт вызовет метод `createWatcher()`, после чего настроит возвращенный объект, так чтобы результаты печатались на стандартный вывод. К какой базе данных подключаться и с какого порядкового номера начинать, задается в аргументах командной строки.

Пока в этом коде нет ничего, относящегося к CouchDB. Это просто стандартный способ программирования Node.js. Поначалу код может показаться необычным, особенно если вам раньше не доводилось работать с событийно-управляемыми серверами, но далее в этой книге мы будем часто использовать такую технологию.

Имея заготовку, добавим код для долгого опроса CouchDB и генерации событий. Показанный ниже код следует поместить в метод `watcher.start()` (туда, где находится комментарий *«детали, относящиеся к ленте обновлений»*). Результирующий файл назовите `watch_changes_longpolling.js`.

couchdb/watch_changes_longpolling_impl.js

```
var
❶ http_options = {
  host: watcher.host,
  port: watcher.port,
  path:
    '/' + watcher.db + '/_changes' +
    '?feed=longpoll&include_docs=true&since=' + watcher.last_seq
};

❷ http.get(http_options, function(res) {
  var buffer = '';
  res.on('data', function(chunk) {
    buffer += chunk;
  });
  res.on('end', function() {
    ❸ var output = JSON.parse(buffer);
    if (output.results) {
      watcher.last_seq = output.last_seq;
      output.results.forEach(function(change) {
        watcher.emit('change', change);
      });
    }
  });
});
```



```

    watcher.start();
  } else {
    watcher.emit('error', output);
  }
})
})
.on('error', function(err) {
  watcher.emit('error', err);
});

```

- ❶ Этот скрипт первым делом настраивает конфигурационный объект `http_options`, подготавливая его к отправке запроса. Параметр `path` указывает на уже известный нам `URL_changes`, в котором `feed` задает режим долгого опроса, а `include_docs=true`.
- ❷ Затем скрипт вызывает библиотечный метод Node.js `http.get()`, который отправляет GET-запрос с указанными параметрами. Второй параметр `http.get` – функция обратного вызова, которая получит объект `HTTPResponse`. Получая содержимое, объект ответа генерирует события `data`, в ответ на которые мы добавляем полученное содержимое в буфер.
- ❸ Наконец, в ответ на сгенерированное объектом ответа событие `end` мы разбираем содержимое буфера (это должен быть JSON-объект) и узнаем новое значение `last_seq`. Затем мы генерируем события изменения и повторно вызываем `watcher.start()` в ожидании новой порции изменений.

Из командной строки этот скрипт запускается следующим образом (выдача сокращена):

```

$ node watch_changes_longpolling.js music
{ seq: 1,
  id: '370255',
  changes: [ { rev: '1-a7b7cc38d4130f0a5f3eae5d2c963d85' } ] },
  doc:
    { _id: '370255',
      _rev: '1-a7b7cc38d4130f0a5f3eae5d2c963d85',
      albums: [ [Object] ],
      id: '370255',
      name: '""ATTIC""',
      url: 'http://www.jamendo.com/artist/ATTIC_(3)',
      mbgid: '',
      random: 0.4121620435325435 } }
{ seq: 2,
  id: '370254',
  changes: [ { rev: '1-2c7e0deec3ffca959ba0169b0e8bfcef' } ] },
  doc:
    { _id: '370254',
      _rev: '1-2c7e0deec3ffca959ba0169b0e8bfcef',
... еще 98 записей ...

```

Ура, наше приложение работает! После вывода записей для всех документов, процесс не завершается, а продолжает опрашивать CouchDB о новых изменениях.

Попробуйте модифицировать какой-нибудь документ в Futon непосредственно или увеличьте значение `@max` в скрипте `import_from_jamendo.rb` и запустите его еще раз. Вы увидите, что изменения напечатаются в окне терминала. Далее мы увидим, как можно включить «полный вперед» и с помощью непрерывного отслеживания изменения получать обновления еще быстрее.

Непрерывное отслеживание изменений

Опрос и долгий опрос службы `_changes` дают правильные результаты в формате JSON. Механизм непрерывного отслеживания работает несколько иначе. Вместо того чтобы помещать все доступные изменения в массив `results` и затем закрывать поток, он посылает каждое изменение по отдельности и оставляет соединение открытым. Таким образом, уведомления о новых изменениях, представленные в виде JSON-объектов, становятся доступны сразу после изменения.

Чтобы увидеть, как это работает, введите такую команду (выдача сокращена):

```
$ curl 'http://localhost:5984/music/_changes?since=97&feed=continuous'
{"seq":98,"id":"357999","changes":[{"rev":"1-0329f5c885...87b39beab0"}]}
{"seq":99,"id":"357998","changes":[{"rev":"1-79c3fd2fe6...1e45e4e35f"}]}
{"seq":100,"id":"357995","changes":[{"rev":"1-aa649aa53f...320c235aee"}]}
```

Если в течение некоторого времени никаких изменений не было, то CouchDB все-таки закроет соединение, но предварительно отправит такую строку:

```
{"last_seq":100}
```

Преимущество этого способа по сравнению с опросом и долгим опросом состоит в уменьшении накладных расходов благодаря тому, что соединение остается открытым. Не приходится тратить время на повторное установление HTTP-соединений. С другой стороны, вывод не является JSON-объектом в строгом смысле слова, поэтому для разбора требуется немного больше усилий. Наконец, это решение не годится, если в роли клиента выступает веб-браузер. При асинхронной загрузке обработчик не получит данных, пока сервер не закроет соединение (в таком случае лучше использовать долгий опрос).

Фильтрация изменений

Как мы только что видели, Changes API предоставляет уникальную возможность подглядывать за тем, что происходит в базе данных CouchDB. То, что он помещает изменения в общий поток, – это плюс, но иногда требуется знать не обо всех, а только о некоторых изменениях. Например, только об операциях удаления или об изменении документов, обладающих определенными характеристиками. Тут на помощь приходят *функции-фильтры*.

Фильтр – это функция, которая принимает на входе документ (и информацию из запроса) и решает, следует ли пропустить документ дальше. О своем решении она сообщает с помощью кода возврата. Посмотрим, как это выглядит на практике. В нашей базе данных `music` в большинстве документов об исполнителях имеется трехбуквенный код страны в свойстве `country`. Предположим, что нас интересуют только группы из России (RUS). Соответствующую функцию-фильтр можно записать так:

```
function(doc) {  
  return doc.country === "RUS";  
}
```

Если добавить ее в проектный документ с ключом `filters`, то мы сможем указывать ее в запросах к `URL_changes`. Но предварительно обобщим задачу. Вместо того чтобы всегда запрашивать только российские группы, было бы лучше иметь возможность параметрически задавать страну в URL. Вот как выглядит параметризованная функция-фильтр:

```
function(doc, req) {  
  return doc.country === req.query.country;  
}
```

На этот раз мы сравниваем свойство `country` документа с одноименным параметром, заданным в строке запроса. Чтобы увидеть, как это работает, создайте новый проектный документ, предназначенный исключительно для хранения фильтров по географическим характеристикам:

```
$ curl -X PUT \  
http://localhost:5984/music/_design/wherabouts \  
-H "Content-Type: application/json" \  
-d '{"language": "javascript", "filters": {"by_country":  
  "function(doc, req){return doc.country === req.query.country;}"  
}}'  
{  
  "ok": true,
```

```
"id": "_design/wherabouts",
"rev": "1-c08b557d676ab861957eaeb85b628d74"
}
```

Теперь можно отправить запрос об изменениях с фильтрацией по стране:

```
$ curl "http://localhost:5984/music/_changes?
filter=wherabouts/by_country&
country=RUS"
{"results": [
{"seq": 10, "id": "5987", "changes": [{"rev": "1-2221be...a3b254"}]},
{"seq": 57, "id": "349359", "changes": [{"rev": "1-548bde...888a83"}]},
{"seq": 73, "id": "364718", "changes": [{"rev": "1-158d2e...5a7219"}]},
...
}
```

Фильтры позволяют реализовать нечто вроде псевдосегментирования, когда на другие узлы реплицируется только подмножество записей. Это, конечно, не настоящее сегментирование, как в MongoDB или HBase, но все же позволяет распределить ответственность за обслуживание запросов определенного вида. Например, на главном сервере CouchDB можно определить фильтры для пользователей, заказов, сообщений и складских запасов. Результаты применения этих фильтров можно реплицировать на отдельные серверы CouchDB, каждый из которых отвечает за определенную сторону работы предприятия.

Поскольку функции-фильтры могут содержать произвольный код на JavaScript, в них можно поместить и более сложную логику, например, проверку глубоко вложенных полей, как мы делали при создании представлений. Можно также проверять свойства с помощью регулярных выражений или математических операций (скажем, фильтровать по диапазону дат). В объекте запроса имеется даже свойство `req.userCtx`, содержащее информацию о текущем пользователе, так что при фильтрации изменений можно учитывать, кто именно запрашивает данные.

Мы еще вернемся к серверу Node.js и интерфейсу CouchDB Changes API в главе 8, когда будем разрабатывать приложение с несколькими базами данных. А пока перейдем к последней особенности, отличающей CouchDB: репликации.

CouchDB или BigCouch?

Подход CouchDB во многих случаях вполне оправдан. Она, безусловно, заполняет нишу, которую большинство других рассматриваемых в этой книге баз данных оставляют без внимания. С другой стороны, иногда бывает удобно избирательно реплицировать данные на другие узлы, чтобы более рационально использовать место на диске. Иначе говоря, хранить копии данные

не на всех узлах, а только на некоторых – при условии, что общее число копий не меньше заданного порога. Этот порог – величина N в аббревиатуре NWR, обуславливающейся в разделе «Узлы и операции чтения-записи» в главе 3.

CouchDB не предоставляет такую функциональность в готовом виде, но не отчаивайтесь! На этот случай имеется система BigCouch. Эта система, разработанная и поддерживаемая компанией Cloudant, имеет совместимый с CouchDB интерфейс (с небольшими отличиями⁹). Но под капотом в ней реализована стратегия сегментирования и репликации в духе системы Dynamo – как в Riak.

Установка BigCouch – дело куда более сложное, чем установка CouchDB, но овчинка может стоить выделки, если речь идет о развертывании в большом центре обработки данных.

Репликация данных в CouchDB

CouchDB задумывалась, прежде всего, ради асинхронной обработки данных и обеспечения их долговечности. Согласно идеологии CouchDB, самое безопасное место хранения данных – повсюду, и она предоставляет для этого необходимые инструменты. В некоторых других СУБД для гарантии согласованности выделяется один главный узел. В других для этой цели используется кворум согласных между собой узлов. В CouchDB ничего этого нет; зато она поддерживает так называемую репликацию главный-главный, или репликацию с несколькими главными узлами.

Любой сервер CouchDB способен наравне с другими принимать обновления, отвечать на запросы и удалять данные, даже если он не может связаться ни с одним другим сервером. В такой модели изменения избирательно реплицируются в одном направлении, и все данные равно могут служить предметом репликации. Иными словами, сегментирования нет. Каждый сервер, участвующий в репликации, хранит все данные.

Репликация – последняя крупная особенность CouchDB, которую мы обсудим. Сначала посмотрим, как настроить одноразовую и непрерывную репликацию баз данных. А затем рассмотрим, к чему могут приводить конфликты в данных и как писать приложения, способные разрешать эти конфликты.

Для начала щелкните по ссылке Replicator в меню Tools, расположенном на странице справа. В результате откроется страница, показанная на рис. 29. В форме «Replicate changes from» (Реплицировать изменения из) выберите в левом раскрывающемся списке базу данных *music*, а в поле справа от него введите *music-repl*. Флажок

9 <http://bigcouch.cloudant.com/api>

Continuous (Непрерывно) не отмечайте и нажмите кнопку Replicate. Когда система спросит, создать ли базу данных `music-repl`, нажмите ОК. В результате в области журнала событий под формой появится сообщение о новом событии.

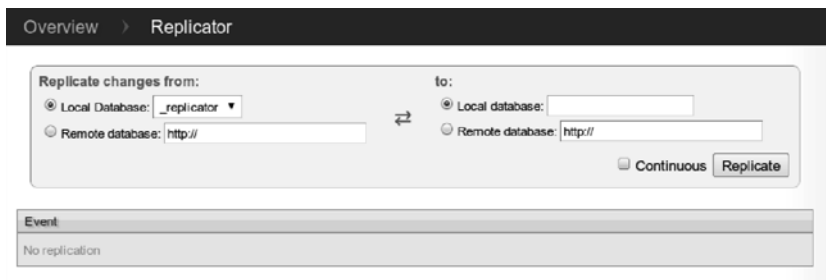


Рис. 29. CouchDB Futon: окно репликатора

Чтобы убедиться, что запрос репликации сработал, вернитесь на страницу Overview в Futon Overview. На ней должна появиться новая база данных `music-repl` с таким же числом документов, как в базе данных `music`. Если оказалось, что документов меньше, то немного подождите и обновите страницу – возможно, CouchDB еще не успела скопировать все данные. Не удивляйтесь, если значения в поле `Update Seq` (порядковый номер обновления) не совпадут. Это связано с тем, что в исходной базе данных `music` мы обновляли и удаляли документы, а при репликации в базу `music-repl` производятся только вставки – ради ускорения.

Создание конфликтов

Далее мы намеренно создадим конфликт и посмотрим, как его обрабатывать. Держите страницу Replicator открытой, поскольку мы будем часто запускать одноразовую репликацию между базами `music` и `music-repl`.

Введите в окне терминала следующую команду для создания документа в базе данных `music`:

```
$ curl -X PUT "http://localhost:5984/music/theconflicts" \
  -H "Content-Type: application/json" \
  -d '{ "name": "The Conflicts" }'
{
  "ok":true,
  "id":"theconflicts",
  "rev":"1-e007498c59e95d23912be35545049174"
}
```

На странице Replicator нажмите кнопку Replicate, чтобы снова запустить синхронизацию. Убедиться, что документ реплицирован успешно, можно, запросив его из базы данных `music-repl`.

```
$ curl "http://localhost:5984/music-repl/theconflicts"
{
  "_id": "theconflicts",
  "_rev": "1-e007498c59e95d23912be35545049174",
  "name": "The Conflicts"
}
```

Затем обновите базу данных `music-repl`, добавив альбом *Conflicts of Interest*.

```
$ curl -X PUT "http://localhost:5984/music-repl/theconflicts" \
-H "Content-Type: application/json" \
-d '{
  "_id": "theconflicts",
  "_rev": "1-e007498c59e95d23912be35545049174",
  "name": "The Conflicts",
  "albums": ["Conflicts of Interest"]
}'
{
  "ok": true,
  "id": "theconflicts",
  "rev": "2-0c969fbfa76eb7fcdf6412ef219fcac5"
}
```

А в базе `music` создайте конфликтующее обновление, добавив другой альбом: *Conflicting Opinions*.

```
$ curl -X PUT "http://localhost:5984/music/theconflicts" \
-H "Content-Type: application/json" \
-d '{
  "_id": "theconflicts",
  "_rev": "1-e007498c59e95d23912be35545049174",
  "name": "The Conflicts",
  "albums": ["Conflicting Opinions"]
}'
{
  "ok": true,
  "id": "theconflicts",
  "rev": "2-cab47bf4444a20d6a2d2204330fdce2a"
}
```

Сейчас в базах `music` и `music-repl` имеется документ с одним и тем же значением `_id: theconflicts`. У обоих документов версия равна 2, и оба основаны на одной и той же исходной редакции (1-e007498c59e95d23912be35545049174). Теперь возникает вопрос: что произойдет при попытке репликации из одной базы в другую?

Разрешение конфликтов

Теперь, когда базы находятся в конфликтующем состоянии, перейдите на страницу Replicator и снова запустите репликацию. Если вы ожидали ошибку, то будете поражены, увидев, что операция завершилась успешно. Так как же CouchDB справилась с расхождением?

Как выясняется, CouchDB просто выбирает одно обновление и объявляет его победителем. Благодаря использованию детерминированного алгоритма CouchDB при обнаружении конфликта всегда назначает победителем одно и то же обновление. Однако на этом история не заканчивается. CouchDB сохраняет и «проигравшие» документы, чтобы клиентское приложение впоследствии могло оценить ситуацию и разрешить конфликт самостоятельно.

Чтобы узнать, какая версия документа победила при последней репликации, мы можем запросить ее с помощью обычного GET-запроса. Если добавить в URL параметр `conflicts=true`, то CouchDB включит также сведения о конфликтующих редакциях.

```
$ curl http://localhost:5984/music-repl/theconflicts?conflicts=true
{
  "_id": "theconflicts",
  "_rev": "2-cab47bf4444a20d6a2d2204330fdce2a",
  "name": "The Conflicts",
  "albums": [ "Conflicting Opinions" ],
  "_conflicts": [
    "2-0c969fbfa76eb7fcdcf6412ef219fcac5"
  ]
}
```

Как видим, победило второе обновление. Обратите внимание на поле `_conflicts` в ответе. Оно содержит список редакций, конфликтующих с выбранной. Добавив в GET-запрос параметр `rev`, мы сможем получить конфликтующие редакции и решить, что с ними делать.

```
$ curl http://localhost:5984/music-repl/theconflicts?rev=2-0c969f...
{
  "_id": "theconflicts",
  "_rev": "2-0c969fbfa76eb7fcdcf6412ef219fcac5",
  "name": "The Conflicts",
  "albums": [ "Conflicts of Interest" ]
}
```

Вывод таков: CouchDB не пытается интеллектуально объединять конфликтующие изменения. Как это делать, зависит от приложения, и никакого общего решения не существует. В нашем случае имело бы смысл объединить оба массива `albums`, но легко придумать примеры, когда правильное действие не столь очевидно.

Например, рассмотрим базу данных событий в календаре. Одна ее копия хранится в вашем смартфоне, другая – в вашем же ноутбуке. Вы получаете от планировщика приемов текстовое сообщение с указанием места проведения приема, который вы организуете, и вносите данные в смартфонную базу данных. Позже, уже в офисе, вы получаете от планировщика сообщение по электронной почте, где указано *другое* место проведения. Теперь вы обновляете базу данных в ноутбуке и запускаете репликацию. CouchDB не знает, какое место проведения правильно. Максимум, что она может сделать, – поддержать согласованность, сохранив где-то старое значение, чтобы вы сами могли решить, какое из двух конфликтующих значений оставить. Приложение должно будет организовать интерфейс для представления такой ситуации, оставив окончательное решение на усмотрение пользователя.

День 3: итоги

На этом заканчивается наше краткое знакомство с CouchDB. Мы начали сегодняшний день с того, что добавили функции-редукторы к представлениям, генерируемым каркасом `mapreduce`. Затем мы погрузились в изучение интерфейса `Changes API` и совершили увлекательное путешествие в мир серверных событийно-управляемых программ на JavaScript на примере Node.js. И наконец, мы немного поговорили о том, как в CouchDB реализована стратегия репликации главный-главный и как клиентское приложение может обнаружить и разрешить конфликты.

День 3: домашнее задание

Информационный поиск

1. Какие редукторы изначально встроены в CouchDB? В чем преимущества использования встроенных редукторов по сравнению с пользовательскими, написанными на JavaScript?
2. Как отфильтровать изменения, поступающие от службы `_changes`, на стороне сервера?
3. Как и всё остальное в CouchDB, инициализация и отмена репликации управляются HTTP-запросами. Какие REST-команды применяются для настройки и удаления отношений репликации между серверами?
4. Как можно воспользоваться базой данных `_replicator` для сохранения отношений репликации?

Задачи

1. Создайте новый модуль `watch_changes_continuous.js` для Node.js на базе заготовки, приведенной в разделе «Опрос изменений в приложении для Node.js».
2. Реализуйте метод `watcher.start()`, так чтобы он непрерывно отслеживал изменения через интерфейс `_changes`. Убедитесь, что результаты получаются такими же, как при использовании модуля `watch_changes_longpolling.js`.

Подсказка: если у вас возникнут затруднения, имейте в виду, что пример реализации имеется в материалах, опубликованных на сопроводительном сайте.

3. У документов с конфликтующими редакциями имеется свойство `_conflicts`. Создайте представление, которое будет эмитировать конфликтующие редакции и отображать их на идентификатор документа `_id`.

6.5. Резюме

В этой главе мы рассмотрели довольно широкий спектр задач, решаемых с помощью CouchDB, — от простейших операций CRUD до построения представлений посредством `mapreduce`-функций. Мы видели, как отслеживать изменения, и познакомились с разработкой неблокирующих событийно-управляемых клиентских приложений. Наконец, мы научились выполнять одноразовую репликацию баз данных, а также обнаруживать и разрешать конфликты. И хотя еще осталось много нерассмотренных тем, пришло время подвести итоги и переходить к следующей базе данных.

Сильные стороны CouchDB

CouchDB — надежный и стабильный представитель семейства баз данных категории NoSQL. Предполагая, что сети принципиально ненадежны, а аппаратные сбои неизбежны, CouchDB предлагает полностью децентрализованный подход к организации хранилищ данных. Достаточно малая, чтобы уместиться в смартфоне, и достаточно большая для поддержки корпоративных решений, CouchDB может быть развернута на самых разных платформах.

CouchDB — в равной мере база данных и API. В этой главе мы уделили основное внимание каноническому проекту Apache CouchDB, но существуют альтернативные реализации и поставщики служб CouchDB, построенных на базе гибридных серверных решений, причем их

число постоянно растет. Поскольку CouchDB появилась «из веб и для веб», то она достаточно просто встраивается как отдельный слой в веб-технологии – подобно балансировщикам нагрузки и распределенным кэшам, – но при этом сохраняет уникальность своих API.

Слабые стороны CouchDB

Разумеется, CouchDB годится не для любой задачи. Основанные на технологии mapreduce представления, при всей своей новизне, не могут обеспечить столь же гибкие средства выборки данных, как реляционные СУБД. На самом деле, в производственной системе *вообще не рекомендуется* прибегать к произвольным запросам. Кроме того, принятая в CouchDB стратегия репликации не всегда является подходящим выбором. В CouchDB в основу репликации положен принцип «всё или ничего», то есть на всех реплицируемых серверах хранятся одни и те же данные. Не существует механизма сегментирования, позволяющего распределить данные по различным серверам в ЦОД. Основная причина добавления новых узлов в CouchDB – не столько распределить данные, сколько увеличить производительность операций чтения и записи.

Перед расставанием

Стремление CouchDB обеспечить надежность в условиях неопределенности делает ее подходящей системой для противостояния жесткой реальности дикого мира Интернета. Опираясь на стандартные веб-технологии, в частности HTTP/REST и JSON, CouchDB отлично встраивается в любое решение, где такие технологии активно применяются, то есть – с учетом современных тенденций – везде. Но и в отгороженном от внешнего мира садике ЦОД CouchDB тоже найдет применение, если вы готовы разрешать возникающие конфликты или не против альтернативной реализации типа BigCouch, но не ожидаете найти готовый механизм сегментирования.

У CouchDB есть немало других особенностей, делающих ее уникальной и заслуживающей внимания базой данных. К сожалению, мы не можем рассмотреть их все, но хотя бы перечислим некоторые: простота резервного копирования, двоичные вложения в документы и CouchApps – система для разработки и развертывания веб-приложения прямо через CouchDB без дополнительного ПО промежуточного слоя. Как бы то ни было, мы надеемся, что этого обзора было достаточно, чтобы у вас разыгрался аппетит и захотелось продолжить изучение. Попробуйте взять CouchDB за основу своего следующего веб-приложения, управляемого данными, – вы не будете разочарованы!



ГЛАВА 7.

Neo4J

Эластичный трос не похож на стандартный плотницкий инструмент, да и Neo4j не выглядит стандартной базой данных. Эластичный трос применяется, чтобы скрепить предметы – какой угодно формы. Если возможности прочно скрепить стол с колонной и с грузовичком-пикапом придается первостепенное значение, то это как раз то приспособление, которое вам нужно.

Neo4j – новый тип хранилищ данных NoSQL, получивший название *графовая база данных*. Как следует из названия, данные в ней хранятся в виде графа (в том смысле, в каком он рассматривается в математике). Отличительной особенностью таких баз данных является возможность рисовать их структуру на доске в виде прямоугольных блоков и соединяющих их линий. Всё, что можно изобразить в таком виде, можно сохранить в Neo4j. В Neo4j упор делается скорее на *связи между* значениями, чем на общие характеристики значений (как в коллекциях документов или в строках таблицы). Таким образом, совершенно разнородные данные можно хранить просто и естественно.

Размер Neo4j невелик – настолько, что ее можно внедрить практически в любое приложение. С другой стороны, в Neo4j можно хранить десятки миллиардов узлов и столько же ребер. А учитывая поддержку репликации главный-подчиненный на много серверов, эта СУБД способна справиться с задачей любого размера.

7.1. Neo4J дружит с доской

Предположим, нам необходимо создать систему рекомендаций вин, в которой для вина могут быть определены сорт, регион производства, виноградник, винтаж и десигнация¹. Быть может, требуется хранить статьи авторов, описывающих вина. Или дать пользователям возможность пометать свои любимые вина.

1 Классификация вина, обозначенная на этикетке, например: «легкое вино», «игристое вино», «цитрусовое вино» и т. д. *Прим. перев.*

В реляционной модели вы, наверное, создали бы таблицу категорий и связь многие-ко-многим между вином с одного виноградника и некоторой комбинацией категорий и других данных. Но человек мысленно моделирует данные иначе. Сравните рисунки 30 и 31. В мире реляционных баз данных ходит поговорка: *по прошествии достаточного времени все поля становятся необязательными*. В Neo4j эта проблема решается неявно – за счет того, что значения и структурные связи определяются только там, где необходимо. Если для купажного вина нет винтажа, то можно добавить год розлива по бутылкам и сослаться из винтажей на узел купажирования. Никакой схемы, которую надо было бы править, не существует.

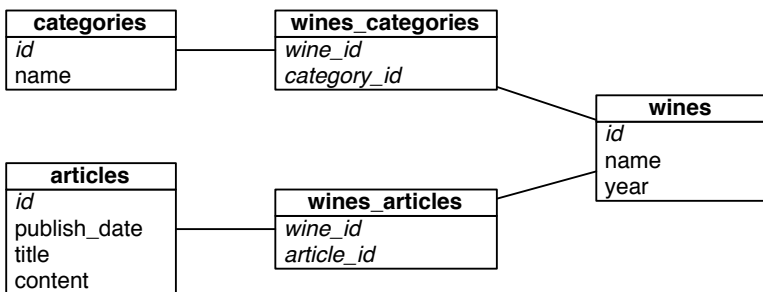


Рис. 30. Схема системы рекомендаций вин в виде UML-диаграммы реляционной базы данных

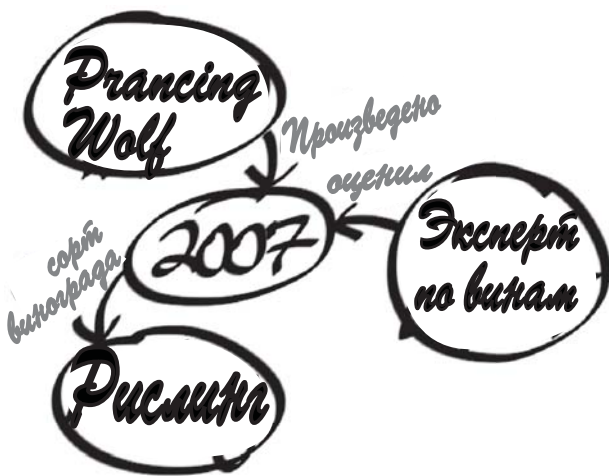


Рис. 31. Данные системы рекомендаций, нарисованные на доске

Следующие три дня мы посвятим изучению взаимодействия с Neo4j с помощью консоли, REST-интерфейса и поисковых индексов. Для работы с большими графами мы будем применять алгоритмы на графах. И в последний день познакомимся со средствами, которые Neo4j предлагает для критически важных приложений масштаба предприятия: от полной поддержки ACID-транзакций до организации высокодоступных кластеров и инкрементного резервного копирования.

В этой главе мы используем издание Neo4j 1.7 Enterprise. Почти все рассматриваемые действия можно было бы выполнить и в издании GPL Community, но на третий день нам понадобится функциональность масштаба предприятия: обеспечение высокой доступности.

7.2. День 1: графы, Groovy и операции CRUD

Сегодня мы собираемся прыгнуть, оттолкнувшись обеими ногами. Помимо знакомства с веб-интерфейсом Neo4j, мы рассмотрим терминологию графовых баз данных и операции CRUD. Этот день мы посвятим в основном изучению того, как опрашивать графовую базу данных с помощью процедуры *обхода*. Излагаемые здесь идеи существенно отличаются от тех, что встречаются в документных или реляционных базах данных. Всё в Neo4j крутится вокруг связей.

Но прежде чем приступать к намеченной программе, запустим веб-интерфейс и посмотрим, как Neo4j представляет данные в виде графа и как этот граф обходить. Скачав и распаковав пакет Neo4j, перейдите в установочный каталог и запустите сервер:

```
$ bin/neo4j start
```

Чтобы проверить, что сервер запущен и работает, отправьте с помощью `curl` запрос на такой URL-адрес:

```
$ curl http://localhost:7474/db/data/
```

Как и CouchDB, стандартный пакет Neo4j включает впечатляющий административный веб-интерфейс и средства просмотра данных; для экспериментов с простыми командами лучше не придумайшь. Мало того, в дистрибутиве имеется один из самых мощных встречавшихся нам обозревателей графовых данных. Для первого знакомства он идеален, потому что обход графов поначалу кажется очень неудобным делом.

Веб-интерфейс Neo4j

Запустите браузер и перейдите на страницу администрирования:

<http://localhost:7474/webadmin/>

Вы увидите цветной, но пока еще пустой граф, изображенный на рис. 32. Щелкните по ссылке Data Browser в верхней части страницы. В только что установленном экземпляре Neo4j имеется один предопределенный ссылочный узел: узел 0.

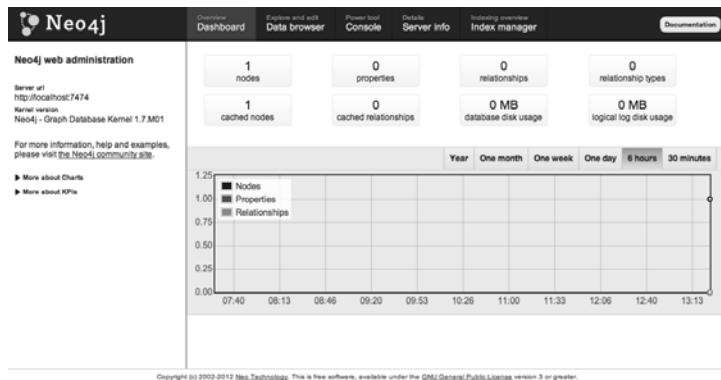
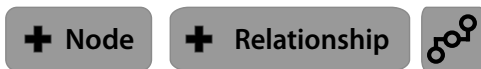


Рис. 32. Индикаторная панель на странице административного веб-интерфейса

Узел в графовой базе данных в какой-то мере напоминает узлы в том смысле, в каком мы употребляли это слово в предыдущих главах. Ранее, говоря об *узле*, мы имели в виду физический сервер в сети. Если рассматривать всю сеть как гигантский взаимосвязанный граф, то серверные узлы будут *вершинами* графа, а *связи* между ними – *ребрами*.

В Neo4j под узлом понимается то же самое: вершина, из которой исходят ребра, где могут храниться данные в виде пар ключ-значение. Нажмите кнопку + Property и задайте ключ *name* и значение *Prancing Wolf Ice Wine 2007*, представляющие конкретное вино и винтаж. Затем нажмите показанную ниже кнопку + Node, добавляющую узел:



Для вновь созданного узла добавьте свойство *name* со значением *Wine Expert Monthly* (мы будем коротко записывать это в виде [name : "Wine Expert Monthly"]). Номер узла автоматически увеличится.

Теперь у нас есть два узла, но их ничто не связывает. Поскольку журнал Wine Expert оценивал вино Prancing Wolf, то мы должны связать эти узлы, создав ребро. Нажмите кнопку + Relationship и установите связь, идущую из узла 1 в узел 0, присвоив ей тип `reported_on`.

Вы перейдете на страницу, соответствующую этой связи:

<http://localhost:7474/db/data/relationship/0>

где показано, что *Node 1 reported_on Node 0*.

Как и у узлов, у связей могут быть свойства. Нажмите кнопку + Add Property и введите свойство `[rating : 92]` – это оценка, выставленная вину.

Это конкретное ледяное вино изготовлено из винограда сорта *riesling*, добавим и эту информацию. Можно было бы добавить это свойство прямо в узел вина, но рислинг – это общая категория, которая может быть ассоциирована и с другими винами, поэтому создадим новый узел и зададим для него свойство `[name : "riesling"]`. Затем добавьте еще одну связь, идущую из узла 0 в узел 2, присвоив ей тип `grape_type` и задав свойство `[style : "ice wine"]`.

И как теперь выглядит наш граф? Нажав кнопку «переключить режим просмотра» (закорючка рядом с кнопкой + Relationship), вы увидите картину, изображенную на рис. 33. Кнопка Style открывает меню, в котором можно выбрать, какой профиль использовать для визуализации графа. Чтобы показывать на диаграмме более полезную информацию, нажмите Style и выберите пункт меню New Profile. Вы попадете на страницу «Create new visualization profile» (Создать новый профиль визуализации). Введите в поле сверху имя *wines*, после чего измените метку с `{id}` на `{id}: {prop.name}`. После нажатия кнопки Save вы окажетесь снова на странице визуализации. Теперь из меню Style можно выбрать пункт *wines* – получится картина, изображенная на рис. 34.

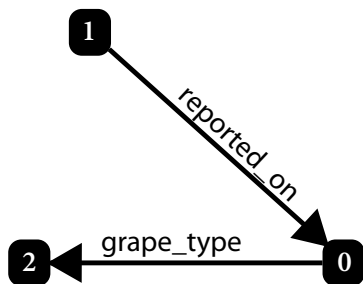


Рис. 33. Граф узлов, связанных с текущим

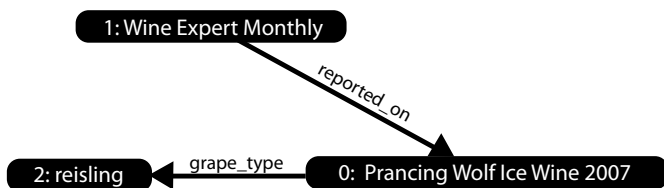


Рис. 34. Граф узлов, визуализированный по специальному профилю

Хотя описанного веб-интерфейса достаточно для внесения простых изменений, работа в производственной системе требует чего-то более развитого.

Neo4j и Gremlin

К Neo4j можно обращаться из нескольких языков: Java, REST-интерфейс, Cypher, Ruby и других. Но сегодня мы воспользуемся языком Gremlin, который написан на языке программирования Groovy и специально предназначен для обхода графов. Впрочем, чтобы использовать Gremlin, Groovy знать необязательно; можете считать, что это просто декларативный предметно-ориентированный язык типа SQL.

Как и другие встречавшиеся нам ранее консольные оболочки, Gremlin дает доступ к инфраструктуре языка, на котором основан. Это означает, что в программе на Gremlin можно использовать конструкции Groovy и библиотеки, написанные на Java. Нам это представляется более выразительным и естественным способом работы с графами, чем писать код на «родном» для Neo4j языке Java. Более того, консоль Gremlin доступна и в административном веб-интерфейсе; просто щелкните по ссылке Console в верхней части страницы и выберите Gremlin.

По принятому соглашению, переменная *g* представляет граф как объект. Действия над графом – это функции, вызываемые от имени данного объекта.

Поскольку Gremlin – универсальный язык обхода графов, то в нем используются термины из математической теории графов. То, что в Neo4j называется *узлом* (node), в Gremlin именуется *вершиной* (vertex), а вместо *связи* (relationship) употребляется термин *ребро* (edge).

Для доступа к множеству всех вершин графа имеется свойство с именем *v* (от слова *vertices*).

```
gremlin> g.v
==>v[0]
```

```
==>v[1]
==>v[2]
```

а для доступа к ребрам графа – свойство `E`.

```
gremlin> g.E
==> e[0][1-reported_on->0]
==> e[1][0-grape_type->2]
```

Получить конкретную вершину можно, передав номер узла методу `v` (в нижнем регистре).

```
gremlin> g.v(0)
==> v[0]
```

Убедиться, что получена нужная вершина, можно, перечислив ее свойства методом `map()`. Отметим, что в Groovy/Gremlin методы можно сцеплять:

```
gremlin> g.v(0).map()
==> name=Prancing Wolf Ice Wine 2007
```

Вызов `v(0)` возвращает конкретный узел, но можно было бы профильтровать всё множество узлов по указанному значению. Например, чтобы получить узел *riesling* по имени, можно воспользоваться синтаксисом фильтра `{...}`, который в Groovy называется *замыканием* (closure). Весь код внутри фигурных скобок определяет функцию; если она возвращает `true`, то программа посещает соответствующую вершину. Переменная `it` внутри замыкания представляет текущий объект, значение ей присваивается автоматически.

```
gremlin> g.V.filter{it.name=='riesling'}
==> v[2]
```

Имея вершину, мы можем получить исходящие из нее ребра, вызвав метод `outE()`. Для получения входящих ребер служит метод `inE()`, а для получения одновременно входящих и исходящих – метод `bothE()`.

```
gremlin> g.V.filter{it.name=='Wine Expert Monthly'}.outE()
==> e[0][1-reported_on->0]
```

Отметим, что в Groovy, как и в Ruby, скобки при вызове методов можно опускать, поэтому запись `outE` дает тот же самый результат.

```
gremlin> g.V.filter{it.name=='Wine Expert Monthly'}.outE
==> e[0][1-reported_on->0]
```

Зная исходящие ребра, мы можем пройти в соседние вершины, вызвав метод `inV`, то есть получить вершины, в которые ведут ребра.

Ребро `reported_on`, исходящее из узла `Wine Expert`, ведет в вершину *Prancing Wolf Ice Wine 2007*, поэтому вызов `outE.inV` ее и вернет. Затем можно прочитать свойство `name` этой вершины:

```
gremlin> g.V.filter{it.name=='Wine Expert Monthly'}.outE.inV.name
==> Prancing Wolf Ice Wine 2007
```

Выражение `outE.inV` возвращает все вершины, в которые ведут исходящие ребра. Обратная операция (получить все вершины, из которых ведут ребра в заданное множество вершин) реализуется выражением `inE.outV`. Поскольку эти операции употребляются очень часто, в Gremlin есть для них сокращенная нотация. Выражение `out` – это сокращенная запись для `outE.inV`, а выражение `in` – для `inE.outV`.

```
gremlin> g.V.filter{it.name=='Wine Expert Monthly'}.out.name
==> Prancing Wolf Ice Wine 2007
```

В одном винодельческом хозяйстве могут производиться разные вина, и, если мы планируем хранить несколько вин, то должны добавить виноградник в качестве связующей вершины, из которой ведет ребро в вершину *Prancing Wolf*.

```
gremlin> pwolf = g.addVertex([name : 'Prancing Wolf Winery'])
==> v[3]
gremlin> g.addEdge(pwolf, g.v(0), 'produced')
==> e[2][3-produced->0]
```

Теперь можно добавить еще два рислинга: *Kabinett* и *Spatlese*.

```
gremlin> kabinett = g.addVertex([name : 'Prancing Wolf Kabinett 2002'])
==> v[4]
gremlin> g.addEdge(pwolf, kabinett, 'produced')
==> e[3][3-produced->4]
gremlin> spatlese = g.addVertex([name : 'Prancing Wolf Spatlese 2007'])
==> v[5]
gremlin> g.addEdge(pwolf, spatlese, 'produced')
==> e[4][3-produced->5]
```

Пополним этот небольшой граф, добавив ребра, ведущие из вершины *riesling* к новым вершинам. Установим переменную *riesling*, отфильтровав вершину *riesling*; для получения первой вершины из конвейера необходим метод `next()`, о котором мы поговорим чуть ниже.

```
gremlin> riesling = g.V.filter{it.name=='riesling'}.next()
==> v[2]
gremlin> g.addEdge([style:'kabinett'], kabinett, riesling, 'grape_type')
==> e[5][4-grape_type->2]
```

Вершину Spatlese можно соединить ребром с riesling аналогично, только в свойство `style` следует записать `spatlese`. После всех этих действий можно визуализировать граф (рис. 35).



Рис. 35. Граф узлов после добавления данных с помощью Gremlin

Конвейеры

Операции в Gremlin можно рассматривать как последовательность каналов (*pipe*). Каждый канал на входе принимает коллекцию, а на выходе возвращает другую коллекцию. В коллекции может быть нуль, один или много элементов. Элементами могут быть вершины, ребра или значения свойств.

Например, канал `outE` принимает коллекцию вершин и выводит коллекцию ребер. Последовательность каналов называется *конвейером* (*pipeline*) и служит для *декларативного* описания задачи. Сравните это с типичным *императивным* программным подходом, в котором вы должны описать шаги, выполняемые для решения задачи. Конвейеры – один из самых кратких способов сформулировать запрос к графовой базе данных.

По сути своей Gremlin – язык для построения каналов. Точнее, он создан на базе проекта Pipes, написанного на Java. Для исследования идеи каналов вернемся к нашему графу вин. Предположим, что нужно найти вина, похожие на данное, – то есть изготовленные из того же сорта винограда. Мы можем проследовать из вершины, соответствующей ледяному вину, по ребру `grape_type` и затем найти вершины, из которых исходят ребра, ведущие в ту же вершину (пропустив узел, из которого вышли).

```
ice_wine = g.v(0)
ice_wine.out('grape_type').in('grape_type').filter{ !it.equals(ice_wine) }
```

Если вам доводилось писать на языке Smalltalk или работать с контекстами на платформе Rails, то такой способ сцепления методов вам знаком. А теперь сравните это с кодом, написанным с помощью стандартного Neo4j Java API, где для доступа к узлам вин с теми же сортовными признаками приходится обходить ведущие из узла связи:

```
enum WineRelationshipType implements RelationshipType {
    grape_type
}

import static WineRelationshipType.grape_type;

public static List<Node> same_variety( Node wine ) {
    List<Node> wine_list = new ArrayList<Node>();
    // пройти по всем ребрам, исходящим из данной вершины
    for( Relationship outE : wine.getRelationships( grape_type ) ) {
        // пройти по всем ребрам, входящим в противоположную вершину
        // данного ребра
        for( Edge inE : outE.getEndNode().getRelationships( grape_type ) ) {
            // добавлять только вершины, отличные от исходной
            if( !inE.getStartNode().equals( wine ) ) {
                wine_list.add( inE.getStartNode() );
            }
        }
    }
    return wine_list;
}
```

Вместо вложенных циклов и обхода ребер проект Pipes предлагает способ объявлять входящие и исходящие вершины. Сначала создается последовательность входных и выходных каналов, фильтров и запрашиваемых у конвейера значений. Затем мы цикле вызываем метод конвейера `next()`, который возвращает следующий подходящий узел. Иными словами, конвейер обходит дерево. Но до тех пор пока у конвейера ничего не запрошено, мы просто объявляем порядок обхода.

Для примера приведем еще одну реализацию метода `same_variety()`, в которой вместо явных циклов используется Pipes:

```
g.V().filter{it.id=='navigation'}.out().filter{it.tag=='li'}.
out().filter{it.name=='section1'}.text
```

```
public static void same_variety( Vertex wine ) {
    List<Vertex> wine_list = new ArrayList<Vertex>();
    Pipe inE = new InPipe( "grape_type" );
    Pipe outE = new OutPipe( "grape_type" );
    Pipe not_wine = new ObjectFilterPipe( wine, true );
    Pipe<Vertex,Vertex> pipeline =
        new Pipeline<Vertex,Vertex>( outE, inE, not_wine );
    pipeline.setStarts( Arrays.asList( wine ) );
    while( pipeline.hasNext() ) {
        wine_list.add( pipeline.next() );
    }
}
```

```
    return wine_list;
}
```

Глубоко в недрах Gremlin скрыт язык построения каналов. Задача обхода графа все равно решается на сервере Neo4j, но Gremlin упрощает формулирование запросов, которые Neo4j понимает.

Говорит Джим: jQuery и Gremlin

Пользователям популярной библиотеки jQuery для языка JavaScript метод обхода коллекций, принятый в Gremlin, вероятно, покажется знакомым. Рассмотрим следующий фрагмент HTML:

```
<ul id="navigation">
  <li>
    <a name="section1">section 1</a>
  </li>
  <li>
    <a name="section2">section 2</a>
  </li>
</ul>
```

Предположим, что нужно найти текст внутри всех тегов с именем `section1`, являющихся потомками элементов ``, которые находятся внутри элемента с идентификатором `navigation` (`id=navigation`). На jQuery для этого можно было бы написать такое выражение:

```
$('#[id=navigation]').children('li').children('[name=section1]').text()
g.V.filter{it.id=='navigation'}.out.filter{it.tag=='li'}.
out.filter{it.name=='section1'}.text
```

А теперь рассмотрим, как мог бы выглядеть запрос на Gremlin к аналогичному набору данных в предположении, что из каждого родительского узла ведут ребра во все его дочерние узлы. Похоже, правда?

Конвейер и вершина

Чтобы получить коллекцию, содержащую только одну указанную вершину, мы можем отфильтровать ее от списка всех узлов. Именно это мы и делаем в вызове `g.V.filter{it.name=='reisling'}`. Свойство `V` представляет список всех узлов, из которого мы выбираем подсписок. Но чтобы получить саму вершину, нужно вызвать метод `next()`, который возвращает первый элемент в конвейере. Это сродни различию между массивом из одного элемента и самим элементом.

Говорит Эрик: язык Cypher

Cypher – еще один поддерживаемый Neo4j язык запросов к графам. Он основан на сопоставлении с образцами и имеет SQL-подобный синтаксис. Фразы выглядят знакомо, что помогает понять смысл запроса. В особенности интуитивно понятна фраза `MATCH`, делающая выражения похожими на ставящие перед собой задачу ASCII-символами.

Поначалу мне не нравилась многословность Cypher, но со временем я на-
вострился читать его грамматику и стал горячим поклонником этого языка.

Взгляните, как на Cypher записывается эквивалент запроса «похожие
вина»:

```
START ice_wine=node(0)
MATCH (ice_wine) -[:grape_type]-> () <-[:grape_type]- (similar)
RETURN similar
```

Мы начинаем с привязки идентификатора `ice_wine` к узлу 0. Во фразе
`MATCH` используются идентификаторы в круглых скобках для обозначения
узлов и типизированные «стрелки» вида `-[:grape_type]->` для обозначения
двусторонних связей. Мне эта конструкция очень нравится, потому что
наглядно представляет обход узлов.

Однако можно пойти и гораздо дальше. Вот более реалистичный пример –
столь же мощный и многословный, как SQL.

```
START ice_wine=node:wines(name="Prancing Wolf Ice Wine 2007")
MATCH ice_wine -[:grape_type]-> wine_type <-[:grape_type]- similar
WHERE wine_type =~ /(?!riesl.*)/
RETURN wine_type.name, collect(similar) as wines, count(*) as wine_count
ORDER BY wine_count desc
LIMIT 10
```

Хотя в этой главе я решил остановиться на Gremlin, оба языка естествен-
но дополняют и мирно сосуществуют друг с другом. В повседневной рабо-
те можно использовать тот или другой – в зависимости от того, как удобнее
рассуждать о поставленной задаче.

Взглянув на класс, сконструированный в результате вызова свойст-
ва фильтра `class`, вы обнаружите, что фильтр возвращает объект
`GremlinPipeline`.

```
gremlin> g.V.filter{it.name=='Prancing Wolf Winery'}.class
==>class com.tinkerpop.gremlin.pipes.GremlinPipeline
```

Но сравните это с классом следующего узла, полученного из кон-
вейера. Это будет нечто иное – `Neo4jVertex`.

```
gremlin> g.V.filter{it.name=='Prancing Wolf Winery'}.next().class
==>class com.tinkerpop.blueprints.pgm.impls.neo4j.Neo4jVertex
```

Хотя на консоли удобно выводить списки узлов, извлеченных из
конвейера, конвейер остается таковым, пока из него не будет что-то
извлечено.

Бессхемная социальная сеть

Для привнесения в граф социального аспекта достаточно добавить
еще несколько узлов. Предположим, требуется добавить трех человек,
знакомых между собой, у каждого из которых есть любимые вина.

Алиса – сладстена, поэтому обожает ледяное вино.

```
alice = g.addVertex([name:'Alice'])
ice_wine = g.V.filter{it.name=='Prancing Wolf Ice Wine 2007'}.next()
g.addEdge(alice, ice_wine, 'likes')
```

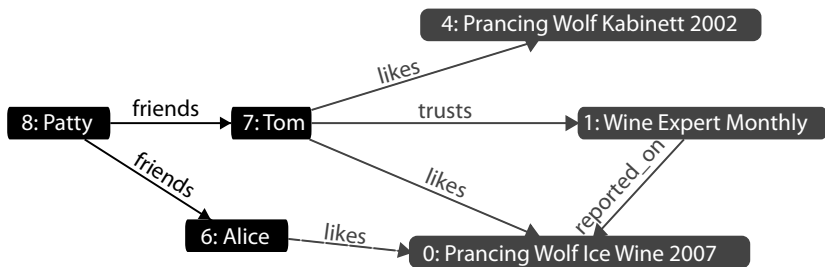
Том любит Кабинет (Kabinett) и ледяное вино и доверяет всему, что пишет *Wine Expert Monthly*.

```
tom = g.addVertex([name:'Tom'])
kabinett = g.V.filter{it.name=='Prancing Wolf Kabinett 2002'}.next()
g.addEdge(tom, kabinett, 'likes')
g.addEdge(tom, ice_wine, 'likes')
g.addEdge(tom, g.V.filter{it.name=='Wine Expert Monthly'}.next(), 'trusts')
```

Пэтти дружит с Томом и Алисой, но в мире вина новичок, ей еще предстоит определиться со своими пристрастиями.

```
patty = g.addVertex([name:'Patty'])
g.addEdge(patty, tom, 'friends')
g.addEdge(patty, alice, 'friends')
```

Не изменяя базовую структуру существующего графа, мы смогли наложить на него поведение, которое заранее не закладывалось. Новые узлы соединены, как показано на рисунке ниже.



Дорога меряется шагами

Мы рассмотрели несколько основных *шагов*, или блоков обработки каналов в Gremlin. Но это далеко не всё. Познакомимся с другими элементами, которые позволяют не только обходить граф, но также преобразовывать объекты, фильтровать шаги и производить побочные эффекты, например, подсчитывать узлы, сгруппированные по некоторому критерию.

Мы уже встречались с методами `inE`, `outE`, `inV` и `outV`, которые представляют собой *шаги трансформации*, позволяющие извлекать входящие и исходящие ребра и вершины. Предоставляются также ме-

тоды `bothE` и `bothV`, которые следуют вдоль ребра вне зависимости от того, является оно входящим или исходящим.

Следующий запрос находит Алису и всех ее друзей. В конец выражения мы добавили `name`, чтобы получить свойство `name` каждой вершины. Поскольку нам безразлично, куда направлено ребро `friend`, мы воспользовались шагами `bothE` и `bothV`.

```
alice.bothE('friends').bothV.name
==> Alice
==> Patty
```

Если Алиса нас не интересует, то можно включить в конвейер фильтр `except()`, передав ему список ненужных узлов.

```
alice.bothE('friends').bothV.except([alice]).name
==> Patty
```

Противоположностью `except()` является фильтр `retain()`, который, как легко догадаться, посещает только те узлы, которые ему переданы.

Можно поступить и по-другому – на последнем шаге отфильтровать вершину с помощью кода, проверяющего, что данная вершина не совпадает с `alice`.

```
alice.bothE('friends').bothV.filter{!it.equals(alice)}.name
```

А как узнать, с кем дружат друзья Алисы? Просто повторить шаги:

```
alice.bothE('friends').bothV.except([alice]).
bothE('friends').bothV.except([alice])
```

Точно так же можно было бы найти друзей друзей друзей Алисы, добавив в цепочку еще одну последовательность вызовов `bothE/bothV/except`. Но пришлось бы слишком долго печатать, и к тому же такой подход не обобщается на переменное число шагов. Для этой цели предназначен метод `loop()`. Он повторяет предыдущие шаги, пока заданное замыкание остается равным `true`.

Показанный ниже код в цикле повторяет три предшествующих шага, отсчитывая назад точки перед вызовом метода `loop`: шаг `except` – первый, шаг `bothV` – второй и шаг `bothE` – третий.

```
alice.bothE('friends').bothV.except([alice]).loop(3){
    it.loops <= 2
}.name
```

После каждой итерации `loop()` вызывает переданное ему замыкание, то есть код в фигурных скобках. В данном случае в свойстве

`it.loops` хранится количество уже выполненных итераций цикла. Мы сравниваем это число с 2 и возвращаем результат проверки, следовательно, цикл остановится после двух итераций. По существу, замыкание очень похоже на условие цикла `while` в типичном языке программирования.

```
==>Tom
==>Patty
==>Patty
```

Цикл правильно нашел Тома и Пэтти. Только у нас получилось две копии Пэтти: первая – потому что Пэтти дружит с Алисой, вторая – потому что она дружит с Томом. Стало быть, нужно как-то отфильтровать дубликаты. Для этого предназначен фильтр `dedup()`.

```
alice.bothE('friends').bothV.except([alice]).loop(3){
    it.loops <= 2
}.dedup.name
==>Tom
==>Patty
```

Чтобы понять, как были получены эти значения, можно проследовать по пути друг->друг, воспользовавшись преобразованием `paths()`.

```
alice.bothE('friends').bothV.except([alice]).loop(3){
    it.loops <= 2
}.dedup.name.paths
==> [v[7], e[12][9-friends->7], v[9], e[11][9-friends->8], v[8], Tom]
==> [v[7], e[12][9-friends->7], v[9], e[11][9-friends->8], v[9], Patty]
```

До сих пор мы обходили граф только в прямом направлении. Но иногда нужно сделать два шага вперед и два назад. Если, начав с узла Алисы, сделать два шага вперед, а потом два шага назад, то мы вернемся в исходный узел.

```
gremlin> alice.outE.inV.back(2).name
==> Alice
```

И последним мы рассмотрим еще один часто используемый шаг `groupCount()`, который обходит узлы и подсчитывает повторяющиеся значения, сохраняя их в ассоциативном массиве. В следующем примере мы обходим все вершины графа и подсчитываем, сколько раз встречаются различные значения свойства `name`.

```
gremlin> name_map = [:]
gremlin> g.V.name.groupCount( name_map )
gremlin> name_map
==> Prancing Wolf Ice Wine 2007=1
```

```
==> Wine Expert Monthly=1
==> riesling=1
==> Prancing Wolf Winery=1
==> Prancing Wolf Kabinett 2002=1
==> Prancing Wolf Spatlese 2007=1
==> Alice=1
==> Tom=1
==> Patty=1
```

В языках Groovy и Gremlin ассоциативный массив обозначается символом `[:]` и практически не отличается от объектов в литеральной нотации `{ }` в языках Ruby и JavaScript. Обратите внимание, что все значения равны 1. Этого и следовало ожидать, поскольку у нас не было повторяющихся имен, а в коллекцию `v` каждая вершина графа входит ровно один раз. Далее подсчитаем, сколько марок вина любит каждый зарегистрированный в системе человек. Для этого нужно получить все вершины, описывающие любимые вина, и подсчитать, сколько их соответствует каждому имени.

```
gremlin> wines_count = [:]
gremlin> g.V.outE('likes').outV.name.groupCount( wines_count )
gremlin> wines_count
==> Alice=1
==> Tom=2
```

Как и ожидалось, Алиса любит одну марку вина, а Том – две.

Переход на уровень Groovy

Помимо шагов на Gremlin, в нашем распоряжении широкий спектр конструкций и методов языка Groovy. В Groovy имеется функция распределения (в духе `mapreduce`), которая называется `collect()`, и функция редуцирования `inject()`. С их помощью можно заранее сформулировать запросы, как в технологии `mapreduce`.

Пусть требуется подсчитать, сколько вин еще не получили оценку. Для этого можно сначала построить список значений `true/false`, показывающих, оценено вино или нет. Затем этот список подается на вход редуктора, который подсчитывает количество `true` и `false`. На фазе распределения функция `collect` используется следующим образом:

```
rated_list = g.V.in('grape_type').collect{
    !it.inE('reported_on').toList().isEmpty()
}
```

Здесь выражение `g.V.in('grape_type')` возвращает вершины, в которые входит ребро типа `grape_type`. Поскольку такие ребра мо-

гут входить только в вершины, описывающие вина, то мы тем самым получаем список всех вин. Далее в замыкании `collect` мы смотрим, входит ли в текущую вершину ребро типа `reported_on`. Вызов метода `toList()` преобразует конвейер в настоящий список, который мы можем проверить на пустоту. В переменной `rated_list`, порождаемой этим кодом, будет находиться список значений `true` и `false`.

Чтобы подсчитать, сколько вин не оценено, редуцируем этот список, воспользовавшись методом `inject()`.

```
rated_list.inject(0){ count, is_rated ->
  if (is_rated) {
    count
  } else {
    count + 1
  }
}
```

В Groovy оператор «стрелка» (`->`) отделяет входные аргументы замыкания от его тела. В нашем редукторе мы хотим обработать значения, собранные на этапе распределения, и понять, оценено текущее вино или нет; для этого и нужны переменные `count` и `is_rated`. Аргумент `0` в `inject(0)` служит для инициализации `count` нулем перед первым обращением. Затем в теле замыкания мы либо возвращаем текущее значение счетчика `count` (если вино оценено), либо увеличиваем его на единицу (если вино не оценено). На выходе мы получим количество значений `false` в списке (то есть количество не оцененных вин).

```
==> 2
```

Как выясняется, оценки еще не получили два вина.

Имея в своем распоряжении описанные инструменты, мы можем составлять различные комбинации обходов и трансформаций графа. Пусть, например, требуется найти все пары друзей. Для этого нужно сначала найти все ребра типа `friends`, а затем вывести имена людей на обоих концах ребра, воспользовавшись операцией `transform`.

```
g.V.outE('friends').transform{[it.outV.name.next(), it.inV.name.next()]}
==> [Patty, Tom]
==> [Patty, Alice]
```

Здесь замыкание `transform` возвращает литеральный массив `[...]` из двух элементов: входящей и исходящей вершин ребра `friend`.

Чтобы найти всех людей и вина, которые они любят, мы преобразуем каждого найденного человека (вершина на любом конце ребра

типа `friends`) в список из двух элементов: имя человека и список его любимых вин.

```
g.V.both('friends').dedup.transform{
  [ it.name, it.out('likes').name.toList() ]
}
==> [Alice, [Prancing Wolf Ice Wine 2007]]
==> [Patty, []]
==> [Tom, [Prancing Wolf Ice Wine 2007, Prancing Wolf Kabinett 2002]]
```

К Gremlin, конечно, нужно привыкнуть, особенно если раньше вы не программировали на Groovy. Но, освоив его, вы согласитесь, что это мощный и выразительный способ предъявления запросов к Neo4j.

Предметно-ориентированные шаги

Обход графов – это прекрасно, но бизнес предпочитает говорить на предметно-ориентированных языках. Мы ведь обычно не спрашиваем: «Из какой вершины, в которую входит ребро типа `grape_type`, исходит ребро, ведущее в данную вершину, описывающую вино»? Вместо этого мы говорим: «Каковы сортовые характеристики этого вина»?

Gremlin уже является языком, ориентированным на определенную предметную область: запрос к графовым базам данных. Но нельзя ли сделать его еще более специфичным? Для этого Gremlin позволяет определять новые шаги, семантически связанные с хранящимися в графе данными.

Начнем с создания шага `varietal`, дающего ответа на поставленный выше вопрос. Когда метод `varietal()` вызывается для некоторой вершины, он ищет исходящие из нее ребра типа `grape_type` и следует по ним к противоположным вершинам.

Нам придется немного углубиться в Groovy, поэтому сначала приведем код шага, а затем подробно опишем его.

neo4j/varietal.groovy

```
Gremlin.defineStep( 'varietal',
  [Vertex, Pipe],
  {_( ).out('grape_type').dedup}
)
```

В начале мы сообщаем ядру Gremlin, что собираемся определить шаг с именем `varietal`. Во второй строке мы говорим, что новый шаг нужно присоединить к классам `Vertex` и `Pipe` (если сомневаетесь, указывайте оба). И в последней строке производится содержательная

работа. По сути дела, мы создаем замыкание, содержащее исполняемый шаг код. Знак подчеркивания и круглые скобки представляют текущий объект конвейера. Из этого объекта мы проследуем к соседним узлам по ребрам типа `grape_type`, то есть в узел, определяющий сортовые характеристики вина. И напоследок вызываем метод `dedup`, чтобы устранить дубликаты.

Новый шаг вызывается, как любой другой. Например, следующий запрос возвращает сорт винограда, из которого изготовлено ледяное вино:

```
g.V.filter{it.name=='Prancing Wolf Ice Wine 2007'}.varietal.name
==> riesling
```

Рассмотрим еще один пример. На этот раз напишем шаг для типичного действия: получить любимые вина всех друзей.

neo4j/friendsuggest.groovy

```
Gremlin.defineStep( 'friendsuggest',
    [Vertex, Pipe],
    {
        _().sideEffect{start = it}.both('friends').
        except([start]).out('likes').dedup
    }
)
```

Как и раньше, даем новому шагу имя – `friendsuggest` – и связываем его с классами `Vertex` и `Pipe`. Наш код должен отфильтровать текущего человека. Для этого мы сохраняем текущую вершину или канал в переменной (`start`), вызывая функцию `sideEffect{start = it}`. Затем мы получаем все узлы `friends`, кроме текущего (мы не хотим считать Алису другом самой себя).

И теперь новый шаг можно включить в конвейер как обычно.

```
g.V.filter{it.name=='Patty'}.friendsuggest.name
==> Prancing Wolf Ice Wine 2007
==> Prancing Wolf Kabinett 2002
```

Поскольку `varietal` и `friendsuggest` – обычные шаги, строящие каналы, то их можно сцеплять для получения более интересных запросов. Так, следующий запрос находит сорта винограда, предпочитаемые друзьями Пэтти:

```
g.V.filter{it.name=='Patty'}.friendsuggest.varietal.name
==> riesling
```

Использование имеющихся в Groovy средств метапрограммирования для создания новых шагов – мощное средство разработки пред-

метно-ориентированных языков. Но к этому подходу, как и к самому Gremlin, нужно привыкнуть.

Обновляем, удаляем, стираем

Обходить граф и вставлять в него данные мы научились, но как насчет обновления и удаления? Просто – нужно только найти вершину или ребро, которое вы собираетесь изменить. Давайте оценим степень любви Алисы к вину Prancing Wolf Ice Wine 2007 года, приписав ей вес.

```
gremlin> e=g.V.filter{it.name=='Alice'}.outE('likes').next()
gremlin> e.weight = 95
gremlin> e.save
```

Удалить значение ничуть не сложнее.

```
gremlin> e.removeProperty('weight')
gremlin> e.save
```

Прежде чем завершить трудный день и перейти к домашнему заданию, расскажем, как очистить базу данных.

Не выполняйте эти команды, пока не сделаете домашнее задание!

У объекта, представляющего граф, есть функции для удаления вершин и ребер: `removeVertex` и `removeEdge` соответственно. Чтобы уничтожить граф, мы можем удалить все вершины и ребра:

```
gremlin> g.V.each{ g.removeVertex(it) }
gremlin> g.E.each{ g.removeEdge(it) }
```

Проверить, что ничего не осталось, можно, вызвав методы `g.V` и `g.E`. Того же результата можно достичь с помощью метода `clear()` – если не боитесь.

```
gremlin> g.clear()
```

Если вы запустили консольную оболочку Gremlin (вне веб-интерфейса), то рекомендуется корректно закрывать соединение с графом методом `shutdown()`.

```
gremlin> g.shutdown()
```

В противном случае можно повредить базу данных. Но обычно сервер просто «обрушает» вас при следующем подключении к графу.



День 1: итоги

Сегодня мы познакомились с графовой базой данных Neo4j и поняли, насколько она отличается от всех остальных. Хотя мы не рассказывали о специфических паттернах проектирования, скажем честно: когда мы только начали работать с Neo4j, голова закружилась от открывающихся возможностей. Все, что можно нарисовать на доске, можно сохранить в графовой базе данных.

День 1: домашнее задание

Информационный поиск

1. Поставьте закладку на вики-сайт Neo4J.
2. Поставьте закладку на документацию по шагам Gremlin – на раздел вики-сайта или описание API.
3. Найдите еще две оболочки для Neo4J (например, оболочка Cypher на административной консоли).

Задачи

1. Запросите имена всех узлов в другой оболочке (например, на языке Cypher).
2. Удалите все узлы и ребра из базы данных.
3. Создайте новый граф, представляющий вашу семью.

7.3. День 2: REST, индексы и алгоритмы

Сегодняшний день мы начнем с изучения REST-интерфейса к Neo4j. Мы создадим с помощью REST узлы и связи между ними, а затем построим индексы и выполним полнотекстовый поиск. Затем мы рассмотрим подключаемый модуль, который позволяет выполнять на сервере Gremlin-запросы, отправленные через REST-интерфейс. Тем самым мы сможем обойтись вообще без консоли Gremlin и накладываемых ей ограничений – даже устанавливать Java на серверы приложений или клиенты необязательно.

REST-интерфейс

Как и Riak, HBase, Mongo и CouchDB, Neo4j поставляется с REST-интерфейсом. Одна из причин, по которым все эти базы данных поддерживают REST, заключается в желании избавиться от языковых

зависимостей, обеспечив стандартный интерфейс подключения. Мы можем подключиться к серверу Neo4j – для работы которого необходима Java – с машины, на которой никаких следов Java нет и в помине. А благодаря подключаемому модулю Gremlin мы сможем через REST воспользоваться выразительной мощностью лаконичного синтаксиса этого языка.

Прежде всего, проверьте, что REST-сервер запущен, для чего отправьте GET-запрос на базовый URL, который возвращает корневой узел. Сервер прослушивает тот же порт, что и рассмотренный вчера административный веб-интерфейс, только ему соответствует путь `/db/data/`. Для отправки REST-запросов мы воспользуемся нашим надежным другом – программой `curl`.

```
$ curl http://localhost:7474/db/data/
{
  "relationship_index" : "http://localhost:7474/db/data/index/relationship",
  "node" : "http://localhost:7474/db/data/node",
  "relationship_types" : "http://localhost:7474/db/data/relationship/types",
  "extensions_info" : "http://localhost:7474/db/data/ext",
  "node_index" : "http://localhost:7474/db/data/index/node",
  "extensions" : {
  }
}
```

В ответ мы получим JSON-объект с описанием URL-адресов других команд, в частности для операций с узлами и индексами.

Создание узлов и связей с помощью REST

Создать узел или связь с помощью интерфейса Neo4j REST столь же просто, как в CouchDB или Riak. Для создания узла нужно отправить на адрес `/db/data/node` POST-запрос, содержащий данные в формате JSON. Будет удобнее, если для каждого узла задать свойство `name`, – это позволит легко получить информацию об узле: достаточно просто вызвать `name`.

```
$ curl -i -X POST http://localhost:7474/db/data/node \
-H "Content-Type: application/json" \
-d '{"name": "P.G. Wodehouse", "genre": "British Humour"}'
```

В ответ сервер вернет путь к узлу в заголовке `Location` и метаданные узла в теле (мы приводим их в сокращенном виде). Все эти данные можно затем прочитать, отправив GET-запрос на адрес, указанный в заголовке `Location` (или в свойстве `self` в метаданных).

HTTP/1.1 201 Created

Location: http://localhost:7474/db/data/node/9

Content-Type: application/json

```
{
  "outgoing_relationships" :
    "http://localhost:7474/db/data/node/9/relationships/out",
  "data" : {
    "genre" : "British Humour",
    "name" : "P.G. Wodehouse"
  },
  "traverse" : "http://localhost:7474/db/data/node/9/traverse/{returnType}",
  "all_typed_relationships" :
    "http://localhost:7474/db/data/node/9/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/9/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/9",
  "properties" : "http://localhost:7474/db/data/node/9/properties",
  "outgoing_typed_relationships" :
    "http://localhost:7474/db/data/node/9/relationships/out/{-list|&|types}",
  "incoming_relationships" :
    "http://localhost:7474/db/data/node/9/relationships/in",
  "extensions" : {
  },
  "create_relationship" :
    "http://localhost:7474/db/data/node/9/relationships",
  "paged_traverse" :
    "http://localhost:7474/db/.../{returnType}{?pageSize,leaseTime}",
  "all_relationships" :
    "http://localhost:7474/db/data/node/9/relationships/all",
  "incoming_typed_relationships" :
    "http://localhost:7474/db/data/node/9/relationships/in/{-list|&|types}"
}
```

Если требуются только свойства узла (не метаданные), то отправьте GET-запрос на адрес, полученный дописыванием строки / `properties` в конец URL узла. А если дописать еще и имя конкретного свойства, то будет возвращено его значение.

```
$ curl http://localhost:7474/db/data/node/9/properties/genre
"British Humour"
```

Одного узла нам будет маловато, поэтому добавьте еще один со свойствами [`"name" : "JeeveTsakes Charge", "style" : "short story"`]. Поскольку рассказ «Дживз берет бразды правления в свои руки» (Jeeves Takes Charge) написал П. Дж. Вудхауз, между этими узлами можно установить связь.

```
$ curl -i -X POST http://localhost:7474/db/data/node/9/relationships \
-H "Content-Type: application/json" \
-d '{"to": "http://localhost:7474/db/data/node/10", "type": "WROTE",
  "data": {"published": "November 28, 1916"}}'
```

У REST-интерфейса есть замечательная особенность: он заранее, в свойстве `create_relationship` возвращенных в теле метаданных, сообщает, как следует создавать связь. Таким образом, REST-интерфейсы являются самодокументированными.

Найди свой путь

С помощью REST-интерфейса можно найти путь между двумя узлами, отправив POST-запрос на URL `/paths` начального узла. В теле запроса должна присутствовать строка в формате JSON, описывающая, в какой узел идти, по каким ребрам следовать и какой алгоритм поиска пути использовать.

Пусть, например, требуется найти пути по ребрам типа `WROTE` из узла 1, используя алгоритм `shortestPath` и ограничиваясь глубиной 10.

```
$ curl -X POST http://localhost:7474/db/data/node/9/paths \
-H "Content-Type: application/json" \
-d '{ "to": "http://localhost:7474/db/data/node/10",
      "relationships": { "type" : "WROTE" },
      "algorithm": "shortestPath", "max_depth": 10 }'
[ {
  "start" : "http://localhost:7474/db/data/node/9",
  "nodes" : [
    "http://localhost:7474/db/data/node/9",
    "http://localhost:7474/db/data/node/10"
  ],
  "length" : 1,
  "relationships" : [ "http://localhost:7474/db/data/relationship/14" ],
  "end" : "http://localhost:7474/db/data/node/10"
} ]
```

Допустимы также алгоритмы `allPaths`, `allSimplePaths` и `dijkstra`. Подробные сведения об этих алгоритмах можно найти в онлайн-овой документации², но их детальное рассмотрение выходит за рамки этой книги.

Индексирование

Как и другие рассмотренные базы данных, Neo4j поддерживает быстрый поиск по индексам. Однако тут есть одна тонкость. В отличие от других СУБД, где запрос формулируется одинаково вне зависимости от того, построены индексы или нет, в Neo4j применяется иной подход. Связано это с тем, что индексирование – это отдельная служба.

² <http://api.neo4j.org/current/org/neo4j/graphalgo/GraphAlgoFactory.html>

Простейшим из всех индексов является хеш-индекс, состоящий из пар ключ-значение. В роли ключа выступают какие-то хранящиеся в узле данные, а в роли значения – REST-совместимый URL, указывающий на узел в графе. Количество индексов не ограничено. Назовем интересующий нас индекс «authors». В конец URL поместим индексируемое имя автора, а в качестве значения передадим узел 9 (тот, в котором хранятся сведения о Вудхаузе, на вашей машине номер может отличаться).

```
$ curl -X POST http://localhost:7474/db/data/index/node/authors \
-H "Content-Type: application/json" \
-d '{ "uri" : "http://localhost:7474/db/data/node/9",
"key" : "name", "value" : "P.G.+Wodehouse" }'
```

Для выборки узла нужно просто обратиться к индексу, причем в ответ вы получите не заданный выше URL, а сами данные узла.

```
$ curl http://localhost:7474/db/data/index/node/authors/name/P.G.+Wodehouse
```

Помимо хеш-индексов, Neo4j поддерживает инвертированные индексы для полнотекстового поиска, позволяющие предъявлять запросы вида: «Дай мне все книги, названия которых начинаются со слова ‘Jeeves’». Для построения такого индекса необходим весь набор данных, а не отдельная запись, как в предыдущем примере. Как и Riak, Neo4j строит инвертированный индекс с помощью Lucene.

```
$ curl -X POST http://localhost:7474/db/data/index/node \
-H "Content-Type: application/json" \
-d '{ "name": "fulltext", "config": { "type": "fulltext", "provider": "lucene" } }'
```

Этот POST-запрос возвращает JSON-объект, содержащий сведения о вновь добавленном индексе.

```
{
  "template" : "http://localhost:7474/db/data/index/node/fulltext/{key}/{value}",
  "provider" : "lucene",
  "type" : "fulltext"
}
```

Добавить Вудхауза в полнотекстовый индекс можно следующим образом:

```
curl -X POST http://localhost:7474/db/data/index/node/fulltext \
-H "Content-Type: application/json" \
-d '{ "uri" : "http://localhost:7474/db/data/node/9",
"key" : "name", "value" : "P.G.+Wodehouse" }'
```

Теперь для поиска используется синтаксис запросов Lucene, а в качестве пути указывается URL-индекса.

```
$ curl http://localhost:7474/db/data/index/node/fulltext?query=name:P*
```

Индексы можно строить и по ребрам, достаточно заменить в URL слово *node* словом *relationship*, например: `http://localhost:7474/db/data/index/relationship/published/date/1916-11-28`.

REST и Gremlin

Вчера мы в основном занимались изучением языка Gremlin, а в первой половине сегодняшнего дня – использованием REST-интерфейса. Если вы пребываете в растерянности, чем же все-таки пользоваться, расслабьтесь. Для REST-интерфейса к Neo4j имеется подключаемый модуль Gremlin (в той версии, с которой мы работаем, он устанавливается по умолчанию³). Через REST-интерфейс можно посылать любые команды, которые можно выполнить в консольной оболочке Gremlin. Тем самым вы получаете в свое распоряжение мощь и гибкость обоих инструментов. Это замечательная комбинация, так как Gremlin лучше подходит для сложных запросов, а REST обеспечивает гибкость развертывания и языковую независимость.

Следующий запрос возвращает имена всех вершин. Требуется всего лишь отправить данные на URL подключаемого модуля в виде JSON-строки в поле `script`.

```
$ curl -X POST \
http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script \
-H "content-type:application/json" \
-d '{"script": "g.V.name"}'
```

```
[ "P.G. Wodehouse", "Jeeves Takes Charge" ]
```

Хотя, начиная с этого момента, мы будем использовать Gremlin, не забывайте, что с тем же успехом можно было бы остановиться на REST.

Большие данные

До сих пор мы имели дело с крохотными наборами данных, так что самое время узнать, как Neo4j поведет себя, когда объем данных велик.

Исследуем набор данных о кинофильмах, который можно скачать с сайта [Freebase.com](http://freebase.com). Мы будем работать с набором «performance», в котором поля разделены знаками табуляции⁴. Скачайте этот набор и выполните показанный ниже скрипт, который последовательно чита-

3 <http://docs.neo4j.org/chunked/stable/gremlin-plugin.html>

4 <http://download.freebase.com/datadumps/latest/browse/film/performance.tsv>



ет строки и создает связи между новыми или существующими узлами (совпадения ищутся по имени в индексе).

Имейте в виду, что этот набор данных содержит очень много информации о фильмах самых разных жанров – от блокбастеров до иностранного кино и – как бы это сказать? – фильмов для взрослых. Для запуска этого Ruby-скрипта вам понадобятся gem-пакеты `json` и `faraday`.

neo4j/importer.rb

```
REST_URL = 'http://localhost:7474/'
HEADER = { 'Content-Type' => 'application/json' }

%w{rubygems json cgi faraday}.each{|r| require r}

# подключиться к REST-серверу Neo4j
conn = Faraday.new(:url => REST_URL) do |builder|
  builder.adapter :net_http
end

# этот метод ищет существующий узел в индексе или создает новый
def get_or_create_node(conn, index, value)
  # ищем узел в индексе
  r = conn.get("/db/data/index/node/#{index}/name/#{CGI.escape(value)}")
  node = (JSON.parse(r.body).first || {})[self] if r.status == 200
  unless node
    # в индексе узел не найден, создаем новый
    r = conn.post("/db/data/node", JSON.unparse({"name" => value}), HEADER)
    node = (JSON.parse(r.body) || {})[self] if [200, 201].include? r.status
    # добавляем новый узел в индекс
    node_data = "{ \"uri\" : \"#{node}\", \"key\" : \"name\", \"value\" : \"#{CGI.escape(value)}\" }"
    conn.post("/db/data/index/node/#{index}", node_data, HEADER)
  end
  node
end

puts "начинается обработка..."

count = 0
File.open(ARGV[0]).each do |line|
  _, _, actor, movie = line.split("\t")
  next if actor.empty? || movie.empty?
  # строим узлы актера и фильма
  actor_node = get_or_create_node(conn, 'actors', actor)
  movie_node = get_or_create_node(conn, 'movies', movie)

  # создаем связь между актером и фильмом
  conn.post("#{actor_node}/relationships",
```

```
JSON.unparse({ :to => movie_node, :type => 'ACTED_IN' }), HEADER)

puts "загружено связей: #{count}" if (count += 1) % 100 == 0

end

puts "готово!"
```

Написав скрипт, запустите его, указав на скачанный файл `performance.tsv`.

```
$ ruby importer.rb performance.tsv
```

Для обработки всего файла данных может потребоваться несколько часов, но процесс можно в любой момент прервать, ограничившись неполным списком фильмов и актеров. Если вы работаете с версией Ruby 1.9, то можно ускорить обработку, заменив строку `builder.adapter :net_http` строкой `builder.adapter :em_synchrony`, которая создает неблокирующее соединение.

Интересные алгоритмы

Загрузив большой набор данных о фильмах, мы на время оставим REST-интерфейс и вернемся к Gremlin.

Здравствуй, Кевин Бэйкон

Немного развлечемся, реализовав один из самых известных алгоритмов на графах: шесть шагов до Кевина Бэйкона. Этот алгоритм основан на игре, смысл которой – не более чем за 6 переходов найти кратчайшее расстояние между загаданным актёром и Кевином Бэйконом через актёров, вместе с которыми они снимались. Например, Алек Гиннесс снимался в фильме «Кафка» с Терезой Рассел, а та – в фильме «Дикость» с Кевином Бэйконом.

Для начала откройте консоль Gremlin и подключитесь к графу. Далее создайте шаг `costars`, код которого приведен ниже. Он похож на созданный вчера шаг `friendsuggest` – ищет актеров, снимавшихся в тех же фильмах, что актер в указанном узле (то есть расположенных на другом конце любого ребра, исходящего из узлов фильмов, связанных с начальным узлом актера).

```
neo4j/costars.groovy
Gremlin.defineStep( 'costars',
  [Vertex, Pipe],
  {
    _().sideEffect{start = it}.outE('ACTED_IN').
```

```

    inV.inE('ACTED_IN').outV.filter{
        !start.equals(it)
    }.dedup
}
)

```

В Neo4j мы не «запрашиваем» множество значений, а «обходим» граф. Прелесть этой идеи в том, что обычно первым посещенным будет узел, ближайший к начальному (в терминах количества промежуточных ребер, а не взвешенного расстояния). Начнем с поиска начального и конечного узла.

```

gremlin> bacon = g.V.filter{it.name=='Kevin Bacon'}.next()
gremlin> elvis = g.V.filter{it.name=='Elvis Presley'}.next()

```

Далее мы ищем актеров, снимавшихся в одном фильме с начальным, затем – тех, кто снимался в одном фильме с каждым из найденных на предыдущем шаге, и т. д. В классическом варианте игры поиск прекращается после шести шагов, но можно ограничиться и четырьмя (если не найдете, можете повторить поиск, увеличив число шагов). После четырех итераций цикла мы найдем всех актеров, «отстоящих на четыре шага». Воспользуемся ранее созданным шагом `costars`.

```

elvis.costars.loop(1){it.loops < 4}

```

Следует оставить только вершины на пути к Бэйкону, а остальные отбросить.

```

elvis.costars.loop(1){
    it.loops < 4
}.filter{it.equals(bacon)}

```

Чтобы не попадать в узел Кевина Бэйкона дважды, мы досрочно прервем цикл, оказавшись в этом узле. Иными словами, цикл заканчивается после четырех итераций или когда обнаружен узел `bacon`. После этого можно вывести пути, по которым мы попали в узел `bacon`.

```

elvis.costars.loop(1){
    it.loops < 4 & !it.object.equals(bacon)
}.filter{it.equals(bacon)}.paths

```

Теперь осталось только взять из списка возможных путей первый элемент – это будет кратчайший путь, вычисленный раньше всех прочих. Оператор `>>` как раз и извлекает первый элемент из списка.

```

(elvis.costars.loop(1){
    it.loops < 4 & !it.object.equals(bacon)
}.filter{it.equals(bacon)}.paths >> 1)

```


И наконец, мы получаем имя каждой вершины и отфильтровываем пустые ребра, пользуясь командой Groovy `grep`.

```
(elvis.costars.loop(1){
    it.loops < 4 & !it.object.equals(bacon)
}).filter{it.equals(bacon)}.paths >> 1).name.grep{it}
==>Elvis Presley
==>Double Trouble
==>Roddy McDowall
==>The Big Picture
==>Kevin Bacon
```

Мы не знали, кто такой Родди Макдауэлл, но в этом и состоит красота графовой базы данных. Нам и не нужно было этого знать, чтобы получить правильный ответ. Можете поупражняться с Groovy, если хотите вывести не просто список, а что-то более изящное, но сами данные мы уже получили.

Случайный обход

Когда требуется представительная выборка из большого набора данных, удобно воспользоваться «случайным обходом». Сначала создаем генератор случайных чисел.

```
rand = new Random()
```

Затем отфильтровываем необходимую долю полного множества документов. Если, например, требуется вернуть только около трети от примерно 60 фильмов, в которых сыграл Кевин Бэйкон, то можно оставить лишь те, для которых случайное число оказалось меньше 0.33.

```
bacon.outE.filter{rand.nextDouble() <= 0.33}.inV.name
```

В результате должно получиться приблизительно 20 случайных названий фильмов.

Если взять актеров, отстоящих на два шага от Кевина Бэйкона, то получится огромный список (для нашего набора данных он содержит более 300 000 имен).

```
bacon.outE.inV.inE.outV.loop(4){
    it.loops < 3
}.count()
==> 316198
```

А чтобы оставить примерно один процент из этого списка, добавим фильтр. Но не забудьте, что фильтр сам является шагом, поэтому параметр метода `loop` следует увеличить на 1.

```
bacon.outE{
    rand.nextDouble() <= 0.01
```

```
.inV.inE.outV.loop(5){
    it.loops < 3
}.name
```

Мы получили Элиа Вуда. Можно ожидать, что наш алгоритм поиска актеров, близких к Бэйкону, вернет его через два шага. И это действительно так: Элиа Вуд играл в фильме «Столкновение с бездной» вместе с Роном Элдардом, а тот играл в «Спящих» вместе с Бэйконом.

Центрированность

Центрированностью называется мера относительной важности узла в графе. Например, если мы хотим измерить важность узла в сети, исходя из его расстояния до всех других узлов, то потребуется алгоритм вычисления центрированности.

Пожалуй, самым известным алгоритмом вычисления центрированности является Google PageRank, но есть несколько вариантов. Мы реализуем простой вариант *центрированности собственного вектора*, в котором просто вычисляется количество ребер, входящих в узел или исходящих из него. Мы сопоставим каждому актеру число сыгранных им ролей.

Нам необходим ассоциативный массив, который могла бы заполнить функция `groupCount()`, и переменная `count`, отсчитывающая число итераций, чтобы оно не превышало заданного порога.

```
role_count = [:]; count = 0
g.V.in.groupCount(role_count).loop(2){ count++ < 1000 }; ''
```

Ключами ассоциативного массива `role_count` будут вершины, а значениями — число ребер, инцидентных данной вершине. Для интерпретации результата массив лучше отсортировать.

```
role_count.sort{a,b -> a.value <= b.value}
```

Последним будет актер, с самым длинным послужным списком. В нашем наборе эта честь принадлежит легендарному актеру озвучивания Мелу Бланку, набравшему 424 пункта (все их можно просмотреть, выполнив запрос `g.V.filter{it.name=='Mel Blanc'}.out.name`).

Внешние алгоритмы

Написать свой алгоритм, конечно, интересно, но по большей части эта работа уже кем-то проделана. Каркас Java Universal Network/Graph (JUNG) представляет собой коллекцию стандартных алгорит-

мов на графах и других средств для моделирования и визуализации графов. Благодаря проекту Gremlin/Blueprint стало несложно получить доступ к таким алгоритмам из JUNG, как PageRank, HITS, Voltage, алгоритмы вычисления центрированности, и инструментам представления графа в виде матрицы.

Для использования JUNG, необходимо обернуть граф Neo4j, преобразовав его в граф JUNG⁵. Для доступа к графу JUNG у нас есть две возможности: скачать и установить все jar-файлы Blueprint и JUNG в каталог libs сервера Neo4j и перезапустить сервер или скачать уже сконфигурированную консоль Gremlin. Мы рекомендуем второй способ, поскольку это избавит вас от необходимости выискивать в сети различные jar-файлы.

В предположении, что вы скачали консоль gremlin, остановите сервер neo4j и запустите Gremlin. Вам понадобится создать объект Neo4jGraph, передав его конструктору путь к подкаталогу data/graph установочного каталога.

```
g = new Neo4jGraph('/users/x/neo4j-enterprise-1.7/data/graph.db')
```

Мы будем по-прежнему обозначать граф Gremlin буквой *g*. Объект Neo4jGraph необходимо обернуть объектом GraphJung, который мы назовем *j*.

```
j = new GraphJung( g )
```

Кевин Бэйкон был выбран в качестве «центра Вселенной» отчасти благодаря его относительной близости к другим актерам. Он действительно играл в фильмах с другими популярными звездами. Но вообще-то ему вовсе необязательно было играть много ролей самому, важно просто быть связанным с теми, кто связан со многими другими.

Но тогда возникает вопрос: а нельзя ли найти актера, который с точки зрения расстояния до других актеров был бы лучше Кевина Бэйкона?

В JUNG имеется алгоритм оценивания BarycenterScorer, который присваивает каждой вершине оценку, исходя из ее расстояния до всех остальных вершин. Если Кевин Бэйкон – действительно лучший выбор, то его оценка должна быть наименьшей, то есть он «ближе всего» ко всем остальным актерам.

Наш алгоритм из каркаса JUNG должен применяться только к актерам, поэтому сконструируем *трансформатор*, который будет отбрасывать все узлы, кроме актеров. Класс EdgeLabelTransformer передает алгоритму лишь узлы, из которых исходят ребра типа ACTED_IN.

5 <http://blueprints.tinkerpop.com>

```
t = new EdgeLabelTransformer(['ACTED_IN'] as Set, false)
```

Затем нужно импортировать сам алгоритм и создать его экземпляр, передав конструктору граф GraphJung и трансформатор.

```
import edu.uci.ics.jung.algorithms.scoring.BarycenterScorer
barycenter = new BarycenterScorer<Vertex,Edge>( j, t )
```

Теперь можно запросить у BarycenterScorer оценки всех узлов. Найдем, чему равна оценка Кевина Бэйкона.

```
bacon = g.V.filter{it.name=='Kevin Bacon'}.next()
bacon_score = barycenter.getVertexScore(bacon)
~0.0166
```

Получив оценку Бэйкона, мы можем обойти все вершины графа и сохранить те, для которых оценка меньше.

```
connected = [:]
```

Применение алгоритма BarycenterScorer к каждому актеру в базе данных может занять весьма длительное время. Поэтому поступим иначе – применим его лишь к актерам, игравшим с Бэйконом в одном фильме. Это может занять несколько минут, все зависит от мощности оборудования. Сам алгоритм BarycenterScorer работает быстро, но ведь его надо применить ко многим актерам.

```
bacon.costars.each{
  score = b.getVertexScore(it);
  if(score < bacon_score) {
    connected[it] = score;
  }
}
```

Все ключи, оказавшиеся в ассоциативном массиве connected, представляют актеров с лучшей оценкой, чем у Кевина Бэйкона. Но хорошо бы увидеть кого-то нам знакомого, поэтому выведем всех и выберем того, кто больше понравится. На вашей машине результат может отличаться, потому что открытая для общего пользования база данных постоянно меняется.

```
connected.collect{k,v -> k.name + " => " + v}
==> Donald Sutherland => 0.00925
==> Clint Eastwood => 0.01488
...
```

Дональд Сазерленд вошел в список с заслуживающей уважения оценкой ~0.00925. Поэтому гипотетически сыграть с друзьями в игру «Шесть шагов до Дональда Сазерленда» было бы проще, чем в традиционные «Шесть шагов до Кевина Бэйкона». Имея граф j, мы можем

выполнить над нашим набором данных любой алгоритм, имеющийся в JUNG, например PageRank. Как и в случае BarycenterScorer, сначала нужно импортировать класс.

```
import edu.uci.ics.jung.algorithms.scoring.PageRank
pr = new PageRank<Vertex,Edge>( j, t, 0.25d )
```

Полный перечень имеющихся в JUNG алгоритмов можно найти в онлайн-официальной документации в формате Javadoc. Постоянно добавляются новые алгоритмы, поэтому имеет смысл поинтересоваться, что доступно, прежде чем писать самому.

День 2: итоги

Во второй день мы узнали о новых способах взаимодействия с базой данных Neo4j, познакомившись с ее REST-интерфейсом. Мы видели, как с помощью подключаемого модуля Gremlin исполнять написанный на Gremlin код на сервере и получать через REST-интерфейс результаты. Мы поэкспериментировали с большим набором данных и закончили кратким обзором нескольких алгоритмов для исследования этих данных.

День 2: домашнее задание

Информационный поиск

1. Поставьте закладку на документацию по Neo4J REST API.
2. Поставьте закладку на API проекта JUNG и реализованным в нем алгоритмам.
3. Найдите привязку к REST-интерфейсу для своего любимого языка программирования.

Задачи

1. Преобразуйте часть поиска путей в алгоритме «Шесть шагов до Кевина Бэйкона» в отдельный шаг. Затем напишите на Groovy универсальную функцию (например, `def actor_path(g, name1, name2) {...}`), которая принимает граф и два имени актеров и сравнивает расстояния от них до Бэйкона.
2. Выберите какой-нибудь из многочисленных алгоритмов в JUNG и примените его к узлу (или к набору данных, если того требует API).
3. Установите драйвер по своему выбору и воспользуйтесь им для управления графом своей компании, в котором вершины

представляют людей и их роли, а ребра – рабочие отношения (подчиняется, работает вместе). Если компания очень велика, ограничьтесь работающими рядом группами; если очень мала, попробуйте включить еще и клиентов. Найдите в своей организации человека с «наилучшими связями», то есть с наименьшим расстоянием до всех остальных вершин.

7.4. День 3: распределенность и высокая доступность

В заключение мы рассмотрим, как приспособить Neo4j к решению критически важных задач. Мы увидим, что Neo4j может обеспечить надежное хранение данных с помощью ACID-совместимых транзакций. Затем мы установим и настроим высокодоступный кластер Neo4j, чтобы увеличить доступность при обслуживании большого потока запросов на чтение. И наконец, рассмотрим стратегии резервного копирования.

Транзакции

Neo4j – база данных с поддержкой атомарных, непротиворечивых, изолированных и долговечных (ACID) транзакций, как и PostgreSQL. Это делает ее пригодной для хранения важных данных, когда обычно выбирается реляционная СУБД. Как и всегда, в транзакциях Neo4j реализован принцип «всё или ничего». Либо успешно выполняются все операции, входящие в состав транзакции, либо ни одна из них.

Механизм обработки транзакций находится не на уровне Gremlin, а на более низком – в проекте Blueprint, обертывающем Neo4j. Конкретные детали могут меняться от версии к версии. Мы используем версию Gremlin 1.3, основанную на Blueprint 1.0. Если у вас установлена другая версия того или иного продукта, то ищите подробности в документации по Blueprint API.

Как и в PostgreSQL, простые однострочные функции автоматически погружаются в неявную транзакцию. Чтобы продемонстрировать многострочные транзакции, мы должны отключить для объекта графа режим автоматических транзакций, дав Neo4j знать, что мы собираемся обрабатывать транзакции вручную. Для изменения режима транзакций служит функция `setTransactionMode()`.

```
gremlin> g.setTransactionMode(TransactionalGraph.Mode.MANUAL)
```

Для начала и завершения транзакции в графе предназначены методы `startTransaction()` и `stopTransaction(conclusion)`. Завершая транзакцию, необходимо указать, была ли она выполнена успешно. Если нет, то Neo4j откатит все команды, выполненные с начала транзакции. Рекомендуем заключать транзакцию в блок `try/catch`, чтобы все исключения гарантированно приводили к откату.

```
g.startTransaction()
try {
    // выполнить несколько операций над графом...
    g.stopTransaction(TransactionalGraph.Conclusion.SUCCESS)
} catch(e) {
    g.stopTransaction(TransactionalGraph.Conclusion.FAILURE)
}
```

Если вы готовы выйти за рамки Gremlin и работать напрямую с объектом Neo4j `EmbeddedGraphDatabase`, то можете использовать синтаксис транзакций, определенный в Java API. Это может понадобиться, если вы пишете код на Java или на языке, построенном на базе виртуальной машины Java, например JRuby.

```
r = g.getRawGraph()
tx = r.beginTx()
try {
    // выполнить несколько операций над графом...
    tx.success()
} finally {
    tx.finish()
}
```

При любом варианте вы получаете все гарантии ACID. Даже в случае сбоя системы все операции записи будут откачены после перезапуска сервера. Если ручное управление транзакциями не требуется, то лучше оставить режим `TransactionalGraph.Mode.AUTOMATIC`.

Высокая доступность

Режим высокой доступности – это ответ Neo4j на вопрос: «Способна ли графовая база к масштабированию»? Да, но с некоторыми ограничениями. Запись на один подчиненный сервер не сразу синхронизируется с другими подчиненными серверами, поэтому в течение небольшого промежутка времени согласованность (в смысле теоремы CAP) может отсутствовать (но восстановится в конечном счете). Высокодоступный (HA) кластер не гарантирует свойств ACID в полном объеме. По этой причине HA-кластеры Neo4j рассматриваются в основном как решение, призванное повысить количество обслужива-

емых запросов на чтение. Как и в Mongo, входящие в сервер кластеры выбирают главный узел, на котором хранится «золотая копия» данных. Но в отличие от Mongo, на подчиненные серверы можно писать. Эти операции записи будут синхронизированы с главным узлом, который затем распространит изменения на другие подчиненные узлы.

HA-кластер

Для работы с механизмом высокой доступности в Neo4j мы должны сначала настроить кластер. В Neo4j используется внешняя служба координации кластера Zookeeper – еще один прекрасный проект, отпочковавшийся от Apache Hadoop. Это универсальная служба для координации распределенных приложений. В кластере Neo4j она применяется для управления жизненным циклом. С каждым сервером Neo4j связан отдельный координатор, в задачу которого входит управление его местом в кластере, как показано на рис. 36.

К счастью, в состав Neo4j Enterprise уже входит Zookeeper, а также ряд файлов для настройки кластера. Мы собираемся запустить три экземпляра сервера Neo4j Enterprise версии 1.7. Скачать дистрибутив для своей операционной системы можно с сайта (только выберите правильное издание)⁶. Затем распакуйте архив и создайте еще две копии каталога. Мы включили в состав их имен цифры 1, 2, 3 и так и будем в дальнейшем на них ссылаться.

```
tar fx neo4j-enterprise-1.7-unix.tar
mv neo4j-enterprise-1.7 neo4j-enterprise-1.7-1
cp -R neo4j-enterprise-1.7-1 neo4j-enterprise-1.7-2
cp -R neo4j-enterprise-1.7-1 neo4j-enterprise-1.7-3
```

Теперь у нас есть три идентичных копии базы данных.

Обычно следует установить всего одну копию на каждый сервер и настроить кластер так, чтобы все серверы знали друг о друге. Но поскольку мы запускаем все серверы на одной машине, то разнесли их по разным каталогам и назначили им разные порты.

Для создания кластера нужно выполнить пять шагов, начав с конфигурирования координаторов кластера Zookeeper и закончив конфигурированием самих серверов Neo4j.

1. Присвоить каждому координатору уникальный идентификатор.
2. Сконфигурировать каждый координатор, так чтобы он мог общаться с другими координаторами и со своим сервером Neo4j.

⁶ <http://neo4j.org/download/>

3. Запустить все три сервера-координатора.
4. Сконфигурировать каждый сервер Neo4j для работы в режиме высокой доступности, назначить им уникальные порты и сообщить о наличии координаторов.
5. Запустить все три сервера Neo4j.

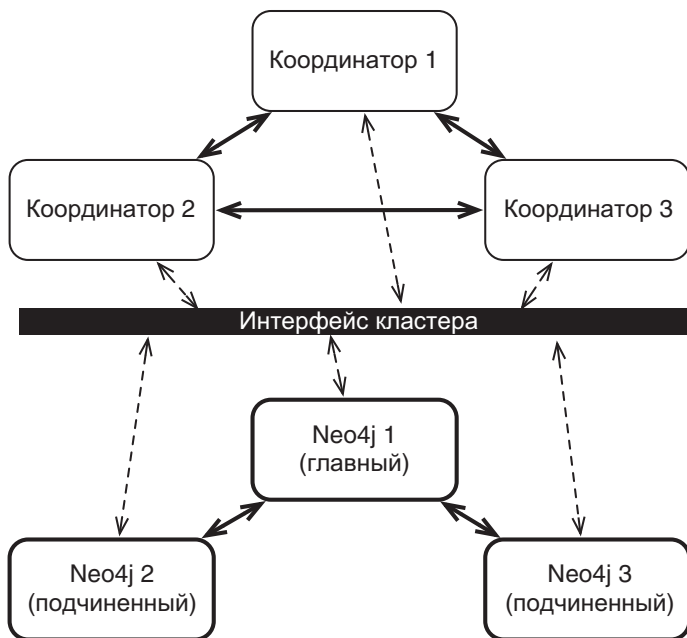


Рис. 36. Кластер из трех серверов Neo4j и их координаторов

Zookeeper отслеживает серверы по уникальному в пределах кластера идентификатору. Это число – единственное, что хранится в файле `data/coordinator/myid`. Для сервера 1 оставим подразумеваемое по умолчанию значение 1, для сервера 2 зададим идентификатор 2, а для сервера 3 – идентификатор 3.

```
echo "2" > neo4j-enterprise-1.7-2/data/coordinator/myid
echo "3" > neo4j-enterprise-1.7-3/data/coordinator/myid
```

Необходимо также задать некоторые внутренние для кластера коммуникационные параметры. У каждого сервера должен быть свой файл `conf/coord.cfg`. По умолчанию в переменную `server.1` записывается имя сервера `localhost` и номера двух портов: для выбора кворума (2888) и для выбора главного узла (3888).

Построение кластера

Кворумом Zookeeper называется группа входящих в кластер серверов и портов, через которые они взаимодействуют (не путать с кворумом в Riak, где этим термином обозначается минимальное большинство, необходимое для обеспечения согласованности). Порт для выбора главного узла используется, когда главный узел выходит из строя, — он нужен для того, чтобы оставшиеся серверы могли выбрать новый главный узел. Мы оставим переменную `server.1` без изменения и добавим переменные `server.2` и `server.3`, указав следующие по порядку номера портов. Файлы `coord.cfg` в каталогах серверов 1, 2 и 3 должны содержать одни и те же строки.

```
server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890
```

Наконец, мы должны определить порт, через который можно подключаться к серверу Neo4j. По умолчанию выбирается `clientPort` 2181, таким мы его и оставим для сервера 1. Для сервера 2 зададим `clientPort=2182`, а для сервера 3 — `clientPort=2183`. Если какие-то из этих портов на вашей машине уже заняты, можете выбрать любые другие, но мы далее будем предполагать, что порты заданы именно так.

Координаторы

Для запуска координатора Zookeeper мы воспользуемся скриптом, любезно предоставленным разработчиками Neo4j. Выполните следующую команду в каждом из трех каталогов серверов:

```
bin/neo4j-coordinator start
Starting Neo4j Coordinator...WARNING: not changing user
process [36542]... waiting for coordinator to be ready. OK.
```

Координаторы теперь работают, но Neo4j — еще нет.

Запускаем Neo4j

Далее мы должны сконфигурировать сервер Neo4j для работы в режиме высокой доступности и подключить его к серверу-координатору. Откройте файл `conf/neo4j-server.properties` и добавьте следующую строку (для каждого сервера):

```
org.neo4j.server.database.mode=HA
```

Тем самым мы говорим, что сервер Neo4j должен работать в режиме высокой доступности; до сих пор мы работали в режиме SINGLE.

Заодно уж давайте назначим порты веб-серверам. Обычно подразумеваемый по умолчанию порт 7474 вполне годится, но коль скоро мы запускаем три экземпляра neo4j на одной машине, то надо их развести по портам протоколов http/https. Серверу 1 назначим порты 7471/7481, серверу 2 – 7472/7482, а серверу 3 – 7473/7483.

```
org.neo4j.server.webserver.port=7471
org.neo4j.server.webserver.https.port=7481
```

Наконец, соединим каждый экземпляр Neo4j с соответствующим ему координатором. Открыв файл `conf/neo4j.properties` для сервера 1, вы обнаружите несколько закомментированных строк, начинающихся словом `ha`. Это параметры режима высокой доступности: номер машины данного сервера в кластере, список серверов Zookeeper и порт, по которому серверы neo4j могут общаться между собой. Для сервера 1 добавьте в файл `neo4j.properties` такие настройки:

```
ha.server_id=1
ha.coordinators=localhost:2181,localhost:2182,localhost:2183
ha.server=localhost:6001
ha.pull_interval=1
```

Настройки двух других серверов аналогичны с одним отличием: для сервера 2 `ha.server_id=2`, а для сервера 3 `ha.server_id=3`. Кроме того, значения `ha.server` тоже должны различаться (мы выбрали для сервера 2 порт 6002, а для сервера 3 – порт 6003). Еще раз отметим, что при запуске серверов на разных машинах менять номера портов необязательно. Таким образом, файл для сервера 2 (и аналогично для сервера 3) должен выглядеть так:

```
ha.server_id=2
ha.coordinators=localhost:2181,localhost:2182,localhost:2183
ha.server=localhost:6002
ha.pull_interval=1
```

Мы задали `pull_interval` равным 1, это означает, что каждый подчиненный сервер должен проверять наличие обновлений на главном один раз в секунду. Вообще говоря, задавать такое маленькое значение не стоит, но мы хотим видеть обновление данных в примере сразу после их вставки.

Сконфигурировав серверы Neo4j для высокодоступного кластера, мы можем их запустить. Запускайте каждый сервер neo4j из его собственного установочного каталога.

```
bin/neo4j start
```

Чтобы следить за сообщениями сервера, выполните команду `tail`, указав его файл журнала.

```
tail -f data/log/console.log
```

Каждый сервер присоединяется к назначенному ему координатору.

Проверка состояния кластера

Координатор, запущенный первым, – вероятно, сервер 1 – становится главным сервером. Чтобы убедиться в этом, откройте административный веб-интерфейс присоединенного экземпляра Neo4j (мы назначили серверу 1 порт 7471). Щелкните по ссылке **Server Info** (Сведения о сервере) в верхней части страницы, а затем по ссылке **High Availability** (Высокая доступность) в боковом меню⁷.

Свойства в списке **High Availability** содержат информацию о кластере. Если данный сервер является главным, соответствующее свойство будет равно `true`. В противном случае можно определить, какой сервер выбран главным, взглянув на список `InstancesInCluster`, где перечислены все подключенные серверы с указанием идентификатора машины, признака главного сервера и другой информации.

Проверка результатов репликации

Запустив кластер, мы можем проверить, правильно ли реплицируются серверы. Если все пойдет по плану, то любая операция записи на подчиненный сервер должна быть реплицирована на главный, а оттуда на все остальные подчиненные серверы. Открыв веб-консоли для каждого из трех серверов, мы сможем воспользоваться встроенной в административный интерфейс консолью Gremlin. Обратите внимание, что объект графа Gremlin теперь обернут в объект `HighlyAvailableGraphDatabase`.

```
g = neo4jgraph[HighlyAvailableGraphDatabase [../neo4j-ent-1.7-2/data/graph.db]]
```

Чтобы протестировать наши серверы, мы добавим в новый граф несколько узлов, содержащих названия знаменитых парадоксов. На консоли одного из подчиненных серверов запишите в корневой узел парадокс Зенона.

```
gremlin> root = g.v(0)
gremlin> root.paradox = "Zeno's"
gremlin> root.save
```

⁷ <http://localhost:7471/webadmin/#/info/org.neo4j/High%20Availability/>

Теперь перейдите на консоль главного сервера и выведите значение парадокса в вершине.

```
gremlin> g.V.paradox  
==> Zeno's
```

Перейдите на консоль другого подчиненного сервера и добавьте парадокс Рассела. Выведя список, мы обнаружим, что на втором подчиненном сервере существуют оба узла, хотя напрямую мы добавляли только один.

```
gremlin> g.addVertex(["paradox" : "Russell's"])  
gremlin> g.V.paradox  
==> Zeno's  
==> Russell's
```

Если на какой-то подчиненный сервер изменения еще не распространились, вернитесь на страницу Server Info, High Availability. Обратите внимание на значения идентификаторов `lastCommittedTransactionId` для всех серверов. Если они совпадают, значит, данные согласованы. Чем меньше число, тем старше версия данных на сервере.

Выборы главного сервера

Если остановить главный сервер и обновить информацию на одном из оставшихся, то обнаружится, что выбран новый главный сервер. Если снова запустить сервер, то он вернется в кластер, но останется подчиненным (до тех пор, пока новый главный сервер работает).

Благодаря механизму высокой доступности сильно нагруженные системы реплицируют данные на несколько серверов и за счет этого разделяют нагрузку. В конечном счете данные на всех серверах кластера будут согласованы, но есть несколько приемов, позволяющих уменьшить вероятность чтения устаревших данных в каждый момент времени, например, связывание сеанса с одним сервером. Если использовать подходящие инструменты, заранее всё спланировать и правильно настроить, то можно создать графовую базу данных, которая содержит миллиарды вершин и ребер и при этом обслуживает столько запросов, сколько вам необходимо. Добавьте сюда регулярное резервное копирование – и вот вам рецепт добротной производственной системы.

Резервное копирование

Резервное копирование – неотъемлемая составная часть любой профессиональной процедуры эксплуатации базы данных. Хотя репликацию можно в некотором смысле считать разновидностью резервного копирования, ежедневное снятие копии с хранением ее вне помещения вычислительного центра настоятельно рекомендуется на случай аварийного восстановления. Трудно планировать пожар в серверной или землетрясение, которое разрушит здание до основания. Издание Neo4j Enterprise включает простую программу резервного копирования под названием neo4j-backup.

При эксплуатации HA-кластера самый правильный подход – запускать команду полного резервного копирования, которая скопирует базу данных с кластера в снабженный временной меткой файл на смонтированном диске. В резервном каталоге будет создана полностью работоспособная копия. Если возникнет необходимость в восстановлении, достаточно будет заменить каталог данных на каждом сервере резервным каталогом – и можно продолжать работу.

Начинать всегда нужно с полного резервного копирования. Следующая команда копирует HA-кластер в каталог, имя которого заканчивается сегодняшней датой (ее возвращает команда `date`, имеющаяся в любом варианте `*nix`).

```
bin/neo4j-backup -full -from ha://localhost:2181,localhost:2182,\
localhost:2183 -to /mnt/backups/neo4j-`date +%Y.%m.%d`.db
```

Если сервер запущен не в режиме высокой доступности, просто измените схему `ha` в URI на `single`. После снятия полной копии вы можете дальше выполнять инкрементное резервное копирование, сохраняя только изменения, произошедшие с момента создания последней копии. Если требуется выполнить полное копирование на одном сервере в полночь, а затем инкрементно копировать изменения каждые два часа, то подойдет, например, такая команда:

```
bin/neo4j-backup -incremental -from single://localhost \
-to /mnt/backups/neo4j-`date +%Y.%m.%d`.db
```

Однако имейте в виду, что инкрементное копирование будет работать, только если в указанном каталоге уже есть полная копия. Поэтому показанная выше команда должна запускаться в тот же день, когда была снята полная резервная копия.

День 3: итоги

Сегодняшний день мы посвятили изучению механизмов обеспечения сохранности данных в Neo4j: ACID-совместимым транзакциям, высокой доступности и резервному копированию.

Важно отметить, что все рассмотренные сегодня инструменты включены только в издание Neo4j Enterprise, использование которого регламентируется схемой двойного лицензирования – GPL/AGPL. Если вы не хотите раскрывать исходный код своего приложения, то должны перейти на издание Community или получить статус OEM от компании Neo Technology, разрабатывающей Neo4j. Для получения более подробной информации обратитесь к команде Neo4j.

День 3: домашнее задание

Информационный поиск

1. Найдите руководство по лицензированию Neo4J.
2. Ответьте на вопрос: «Каково максимальное число поддерживаемых узлов?» (Подсказка: загляните в раздел «Вопросы и ответы» на сайте документации.)

Задачи

1. Реплицируйте Neo4J на три физических сервера.
2. Настройте балансировщик нагрузки на базе какого-нибудь веб-сервера, например Apache или Nginx, и подключитесь к кластеру через REST-интерфейс. Выполните какой-нибудь скрипт на языке Gremlin.

7.5. Резюме

Neo4j – лучшая реализация графовой базы данных (довольно редкий класс) с открытым исходным кодом. В графовых базах данных упор делается на связях между данными, а не на общности значений. Моделировать данные в виде графов просто. Нужно лишь создать узлы и связи между ними и при желании ассоциировать с ними пары ключ-значение. Запросы сводятся к описанию обхода графа, отправляясь из начального узла.

Сильные стороны Neo4j

Neo4j – один из лучших образчиков графовых баз данных с открытым исходным кодом. Такие базы прекрасно подходят для хранения неструктурированных данных – иногда даже лучше, чем документные хранилища. Мало того что в Neo4j нет ни типов, ни схемы, она еще и не накладывает никаких ограничений на взаимосвязи между данными. Это «свалка» в хорошем смысле слова. Текущая версия Neo4j поддерживает 34,4 миллиарда узлов и столько же связей – более чем достаточно для большинства задач (в одном графе Neo4j можно было бы хранить более 42 узлов для каждого из 800 миллионов пользователей Facebook).

В дистрибутив Neo4j входят инструменты для ускорения поиска (Lucene) и простые в использовании (хотя, быть может, не вполне привычные) языковые расширения, в частности Gremlin и REST-интерфейс. Но Neo4j не только проста в использовании, но и работает быстро. В отличие от операций соединения в реляционных базах данных или операций map-reduce в других базах, обход графа требует постоянного времени. Связанные данные находятся всего в одном шаге – не нужно производить массивное соединение с последующей фильтрацией результатов, как в большинстве рассмотренных выше СУБД. Каким бы большим ни был граф, переход из узла А в узел В требует всего одного шага, если между этими узлами имеется связь. Наконец, в издание Enterprise включены средства для создания высокодоступных сайтов с большим количеством запросов на чтение – путем создания HA-кластера Neo4j.

Слабые стороны Neo4j

Neo4j не лишена недостатков. Ребра в Neo4j не могут входить в ту же вершину, из которой исходят⁸. Нам также кажется, что выбранная терминология (*узел* вместо *вершины* и *связь* вместо *ребра*) только затрудняет взаимопонимание. HA-кластер прекрасно справляется с репликацией, но реплицировать разрешено только граф целиком. Невозможно сегментировать граф, что налагает ограничение на его размер (хотя будем честны – ограничение в десятки миллиардов не особенно обременительно). Наконец, если вы ищете продукт с открытым исходным кодом и лицензией, подходящей для корпоративного использования (например, MIT), то Neo4j, пожалуй, не для вас. Издание Community (включающее всё, чем мы пользовались в первые два

8 Такие ребра в теории графов называются петлями. *Прим. перев.*

дня) поставляется по лицензии GPL, но если вам нужны средства, входящие только в издание Enterprise (высокодоступный кластер и резервное копирование), для построения производственной системы, то, вероятно, за лицензию придется заплатить.

Neo4j и теорема CAP

Для тех, кто собирается строить распределенную систему, стратегия Neo4j должна быть ясна из фразы «высокодоступный кластер». Neo4j обладает свойствами доступности и устойчивости к потере связности (то есть относится к классу AP). Каждый подчиненный сервер возвращает только те данные, которыми располагает в данный момент, и они в течение некоторого периода времени могут отличаться от хранящихся в главном узле. Задержку обновления можно сократить, уменьшив интервал между опросами на подчиненном сервере, но технически система все равно обеспечивает только согласованность в конечном счете. Поэтому HA-кластер Neo4j рекомендуется использовать только в приложениях, ориентированных по большей части на чтение.

Перед расставанием

Простота Neo4j может обескуражить, если вы не привыкли моделировать данные в виде графов. Несмотря на мощный API с открытым исходным кодом и годы промышленной эксплуатации, у этой СУБД все еще относительно мало пользователей. Мы объясняем это только недостаточной информированностью, потому что графовые базы данных естественно отражают процесс интерпретации данных человеком. Мы представляем семьи в виде деревьев, друзей – в виде графов; мало кто рассматривает связи между людьми как самоотносимые типы данных. Для некоторых классов задач, например социальных сетей, выбор Neo4j напрашивается сам собой. Но и в не столь очевидных случаях имеет смысл присмотреться к этой СУБД повнимательнее – ее мощь и простота использования, возможно, удивят вас.



ГЛАВА 8.

Redis

Redis можно сравнить со смазкой. Чаще всего он применяется для смазывания движущихся деталей, чтобы уменьшить трение и повысить общую скорость работы. Какие бы механизмы ни использовались в системе, их функционирование можно улучшить, капнув немного масла. Иногда для решения проблемы достаточно просто разумного использования Redis.

Redis (REmote DIctionary Service – удаленная служба словарей) – простое в работе хранилище ключей и значений с развитым набором команд. Первая версия системы вышла в 2009 году. И в быстродействии у него практически нет соперников. Чтение производится быстро, а запись еще быстрее – на некоторых эталонных тестах продемонстрировано до 100 000 операций SET в секунду. Создатель Redis Сальваторе Санфилиппо (Salvatore Sanfilippo) называет свой проект «сервером структур данных», чтобы подчеркнуть заложенные в него возможности работы со сложными типами данных и другие особенности. Изучением этого сверхбыстрого «не просто хранилища ключей и значений» мы и завершим обзор современного ландшафта баз данных.

8.1. Хранилище сервера структур данных

Точно отнести Redis к какой-нибудь категории довольно трудно. На самом базовом уровне это, конечно, хранилище ключей и значений, но такой упрощенный ярлык несколько унижает. Redis поддерживает сложные структуры данных, хотя и не до такой степени, как документо-ориентированные базы данных. Она поддерживает запросы, возвращающие множества, но не таком уровне детальности, как реляционные СУБД, и без поддержки типов. И, разумеется, она *быстрая*,

хотя для достижения этой цели приносит долговечность данных в жертву быстрдействию.

Помимо сервера структур данных, Redis является еще и блокирующей очередью (или стеком), а также системой публикации-подписки. В ней можно настраивать политики срока хранения, уровни долговечности и параметры репликации. Все это делает Redis скорее комплектом инструментальных средств, в который входят полезные алгоритмы работы со структурами данных и процессы, нежели базой данных какого-то определенного жанра.

Обширный список клиентских библиотек для Redis позволяет использовать ее во многих языках программирования. Работа с ней не только проста, но и доставляет удовольствие. Если API для программиста – то же, что удовольствие работы для пользователя, то Redis следует поместить в Музей современного искусства рядом с Mac Cube.

В первые два дня мы будем изучать функции Redis, принятые в ней соглашения и возможности настройки. Начав, как обычно, с простых операций CRUD, мы быстро перейдем к операциям над более сложными структурами данных: списками, хешами, множествами и отсортированными множествами. Мы научимся создавать транзакции и манипулировать характеристиками, определяющими срок хранения данных. Мы воспользуемся Redis для создания простой очереди сообщений и исследуем встроенный механизм публикации-подписки. Затем мы обратимся к вопросу о настройке и параметрах репликации Redis и посмотрим, как поддержать подходящий для приложения баланс между долговечностью данных и быстродействием.

Базы данных часто используются в сочетании – и эта тенденция набирает силу. Мы оставили Redis напоследок, чтобы показать его применение именно в таком качестве. На третий день мы построим систему, которая станет кульминацией этой книги; в ней будут задействованы Redis, CouchDB, Neo4J и Postgres, а цементирующим раствором послужит Node.js.

8.2. День 1: операции CRUD и типы данных

Поскольку командный интерфейс ставится разработчиками Redis во главу угла – и любим всеми пользователями, – то первый день мы потратим на изучение многих из 124 доступных команд. Особый интерес представляют изошренные типы данных и способы их опроса, далеко выходящие за рамки примитивного «получить значение ключа».

Приступая к работе

Redis поддерживают некоторые менеджеры пакетов, в частности Homebrew для Mac, но и собрать его из исходного кода тоже нетрудно¹. Мы будем работать с версией 2.4. Установив ее, запустите сервер командой

```
$ redis-server
```

По умолчанию он работает не в фоновом режиме, но это можно исправить, добавив в конец команды знак `&` или просто открыв еще один терминал. Затем запустите командную утилиту, которая автоматически подключится к подразумеваемому по умолчанию порту 6379. Установив соединение, попробуйте прозвонить сервер.

```
$ redis-cli
redis 127.0.0.1:6379> PING
PONG
```

При невозможности подключиться будет выдано сообщение об ошибке. В ответ на команду *help* будет выведена инструкция по работе со встроенной справкой. Введите *help*, затем пробел, а затем начните вводить какую-нибудь команду. Если вы не знаете ни одной команды Redis, просто нажимайте клавишу Tab – будет циклически перебираться список команд.

```
redis 127.0.0.1:6379> help
Type: "help @<group>" to get a list of commands in <group>
"help <command>" for help on <command>
"help <tab>" to get a list of possible help topics
"quit" to exit
```

Сегодня мы воспользуемся Redis для построения серверной части системы сокращения URL-адресов – по типу tinyurl.com или bit.ly. Сократитель URL – это служба, которая принимает длинный URL и сопоставляет ему короткий адрес в собственном домене, например, <http://www.myveryververylongdomain.com/somelongpath.php> может быть сокращен до <http://bit.ly/VLD>. При переходе по короткому URL система переадресует на исходный URL, избавляя пользователя от необходимости набирать в текстовых сообщениях длинные строки. Заодно владелец службы получает некоторую статистику, в частности о посещаемости.

В Redis существует операция SET, которая сопоставляет короткий ключ, например `7wks`, длинному значению, например <http://www.sevenweeks.org>. Эта операция принимает ровно два параметра:

```
1 http://redis.io
```

ключ и значение. Для получения значения по ключу нужно выполнить операцию GET.

```
redis 127.0.0.1:6379> SET 7wks http://www.sevenweeks.org/
OK
redis 127.0.0.1:6379> GET 7wks
"http://www.sevenweeks.org/"
```

Для уменьшения трафика можно воспользоваться операцией MSET, которая сопоставляет сразу несколько ключей и значений. Так, следующая команда сопоставляет значению Google.com ключ gog, а значению Yahoo.com – ключ yah.

```
redis 127.0.0.1:6379> MSET gog http://www.google.com yah http://www.yahoo.com
OK
```

И наоборот, MGET получает на входе несколько ключей и возвращает упорядоченный список соответствующих им значений.

```
redis 127.0.0.1:6379> MGET gog yah
1) "http://www.google.com/"
2) "http://www.yahoo.com/"
```

Хотя Redis хранит строки, она распознает и целые числа и даже умеет выполнять с ними некоторые простые операции. Если мы хотим следить за тем, сколько коротких ключей хранится в нашем наборе данных, то можем создать счетчик и увеличивать его на единицу командой INCR.

```
redis 127.0.0.1:6379> SET count 2
OK
redis 127.0.0.1:6379> INCR count
(integer) 3
redis 127.0.0.1:6379> GET count
"3"
```

Хотя GET возвращает count в виде строки, INCR понимает, что это целое число и прибавляет к нему единицу. Попытка инкрементировать что-то, кроме целого, закончится плохо.

```
redis 127.0.0.1:6379> SET bad_count "a"
OK
redis 127.0.0.1:6379> INCR bad_count
(error) ERR value is not an integer or out of range
```

Если значение невозможно интерпретировать как целое число, Redis справедливо ругается. Можно также прибавлять (INCRBY) или вычитать (DECR, DECRBY) любое целое значение.

Транзакции

Мы уже встречались с транзакциями в других базах данных (Postgres и Neo4j), и команда Redis `MULTI`, открывающая атомарный блок команд, предназначена для той же цели. Если заключить, скажем, операции `SET` и `INCR` в один блок, то либо они обе выполнятся успешно, либо не выполнятся ни одна. Частичного результата мы никогда не увидим.

Выполним преобразование в ключ еще одного URL и увеличение счетчика в рамках одной транзакции. Транзакция начинается командой `MULTI` и завершается командой `EXEC`.

```
redis 127.0.0.1:6379> MULTI
OK
redis 127.0.0.1:6379> SET prag http://pragprog.com
QUEUED
redis 127.0.0.1:6379> INCR count
QUEUED
redis 127.0.0.1:6379> EXEC
1) OK
2) (integer) 2
```

При использовании `MULTI` команды не выполняются в момент ввода (как в транзакциях Postgres), а ставятся в очередь и затем выполняются последовательно.

Существует и аналог команды SQL `ROLLBACK` – прервать транзакцию позволяет команда `DISCARD`, которая очищает очередь транзакции. Но в отличие от `ROLLBACK`, она не откатывает базу данных в прежнее состояние; транзакция вообще не начинает выполняться. Эффект тот же самый, хотя механизмы совершенно различные (откат транзакции и отмена операции).

Составные типы данных

До сих пор мы видели довольно простое поведение. Сохранение строки и целого числа в качестве значений ключей – даже в рамках транзакции – это, конечно, замечательно, но в большинстве задач программирования и хранения данных приходится иметь дело с гораздо более разнообразными типами данных. Встроенные механизмы хранения списков, хешей, множеств и отсортированных множеств объясняют популярность Redis, и, познакомившись с операциями над этими типами, вы, наверное, согласитесь, что популярность заслужена.

В этих коллекциях можно хранить огромное количество значений (до 2^{32} , или свыше 4 миллиардов элементов). Более чем достаточно

для того, чтобы поместить все учетные записи на Facebook в список под одним ключом.

Хотя некоторые команды Redis выглядят загадочно, все они устроены по общему образцу. Команды над множествами начинаются с буквы S, над хешами – с буквы H, над отсортированными множествами – с буквы Z. Команды над списками обычно начинаются с буквы L (от left – левый) или R (от right – правый) – в зависимости от того, с какой стороны списка применяется операция (например, LPUSH).

Хеш

Хеши играют в Redis роль объектов-контейнеров, способных хранить произвольное число пар ключ-значение. Воспользуемся хешем для учета пользователей, зарегистрировавшихся в нашей службе сокращения URL-адресов.

Хеши хороши тем, что позволяют избежать сохранения данных с ключами, имеющими искусственные префиксы. (Обратите внимание, что внутри ключа используется знак двоеточия. Это допустимый символ, который часто применяется для логического разбиения ключа на сегменты. Но это не более чем соглашение, не имеющее в Redis глубокого смысла.)

```
redis 127.0.0.1:6379> MSET user:eric:name "Eric Redmond" user:eric:password s3cret
OK
```

```
redis 127.0.0.1:6379> MGET user:eric:name user:eric:password
1) "Eric Redmond"
2) "s3cret"
```

Вместо отдельных ключей мы можем создать хеш, в котором будут храниться пары ключ-значение.

```
redis 127.0.0.1:6379> HMSET user:eric name "Eric Redmond" password s3cret
OK
```

Чтобы получить все значения из хеша, нужно знать лишь один ключ.

```
redis 127.0.0.1:6379> HVALS user:eric
1) "Eric Redmond"
2) "s3cret"
```

Можно также получить все ключи из хеша.

```
redis 127.0.0.1:6379> HKEYS user:eric
1) "name"
2) "password"
```

Или запросить только одно значение, передав сначала ключ Redis, а затем ключ внутри хеша. В примере ниже мы получаем только пароль.

```
redis 127.0.0.1:6379> HGET user:eric password
"s3cret"
```

В отличие от документных хранилищ типа Mongo или CouchDB, хеши в Redis не могут быть вложенными (равно как и все прочие составные типы данных, в частности списки). Иными словами, в хеше можно хранить только строки.

Существуют также команды для удаления из хеша (HDEL), увеличения целого значения на заданную величину (HINCRBY) и получения общего числа элементов в хеше (HLEN).

Список

Список содержит упорядоченный набор значений и может выступать как в роли очереди (первым пришел, первым обслужен), так и в роли стека (последним пришел, первым обслужен). Поддерживаются также более сложные операции, например, вставка в середину списка, ограничение размера списка и перемещение значений из одного списка в другой.

Поскольку наша служба сокращения URL теперь умеет учитывать пользователей, то почему бы не предоставить им возможность хранить «списки пожеланий», то есть списки URL-адресов, которые они хотели бы посетить. Назначим списку коротких URL сайтов, куда мы хотели бы заглянуть, ключ USERNAME:wishlist и будем помещать значения в конец списка (справа).

```
redis 127.0.0.1:6379> RPUSH eric:wishlist 7wks gog prag
(integer) 3
```

Как и в большинстве команд вставки в коллекцию, Redis возвращает число вставленных значений. Иначе говоря, раз мы вставили три значения, то получаем в ответ число 3. В любой момент можно узнать длину списка с помощью команды LLEN.

Команда LRANGE позволяет получить любую часть списка, задав первую и последнюю позицию. В операциях со списками считается, что первый элемент имеет индекс 0. Отрицательная позиция означает, что отсчет нужно вести с конца списка.

```
redis 127.0.0.1:6379> LRANGE eric:wishlist 0 -1
1) "7wks"
2) "gog"
3) "prag"
```

Команда LREM удаляет из списка с заданным ключом указанные значения. Ей также необходимо передать число, сообщаемое, сколь-

ко элементов с указанным значением удалять. Если этот параметр равен 0, то удаляются все подходящие значения:

```
redis 127.0.0.1:6379> LREM eric:wishlist 0 gog
```

Если счетчик больше 0, то будет удалено не более заданного числа совпадений, а если меньше 0, то просмотр списка начнется с конца (с правой стороны).

Чтобы удалить и вернуть значения в том порядке, в котором они добавлялись (как в очереди), нужно извлекать их с левого конца списка (головы).

```
redis 127.0.0.1:6379> LPOP eric:wishlist  
"7wks"
```

Чтобы использовать список как стек, нужно добавлять значения командой `R PUSH`, а извлекать командой `R POP`. Все эти операции выполняются за постоянное время.

Поведения очереди можно добиться также комбинацией команд `L PUSH` и `R POP`, а поведения стека – комбинацией `L PUSH` и `L POP`.

Пусть требуется удалить значения из списка пожеланий и переместить их в список посещенных сайтов. Чтобы выполнить перемещение атомарно, мы можем заключить команды `pop` и `push` в блок `MULTI`. На языке Ruby эти шаги можно было бы записать, как показано ниже (командная оболочка здесь не подойдет, так как нам необходимо сохранить извлеченное значение, поэтому мы воспользовались `gem`-пакетом `redis-rb`):

```
redis.multi do  
  site = redis.rpop('eric:wishlist')  
  redis.lpush('eric:visited', site)  
end
```

Но Redis предоставляет также команду, которая позволяет за одну операцию удалить (`pop`) значение из конца одного списка и добавить (`push`) в начало другого. Она называется `RPOPLPUSH` (`pop` справа, `push` слева).

```
redis 127.0.0.1:6379> RPOPLPUSH eric:wishlist eric:visited  
"prag"
```

Если теперь просмотреть список пожеланий, то обнаружится, что элемента `prag` больше нет; он теперь находится в списке `visited`. Это полезный механизм для создания очередей команд.

Если вы поищите в документации по Redis команды `RPOPRPUSH`, `LPOPLPUSH` и `LPOPRPUSH`, то с удивлением обнаружите, что их нет. `RPOPLPUSH` – единственный вариант, поэтому стройте списки соответственно.

Блокирующие списки

Итак, наша служба сокращения URL успешно стартовала и можно подумать о добавлении в нее каких-нибудь социальных функций – например, систему комментирования в реальном времени, где пользователи могли бы оставлять свои замечания о посещенных сайтах.

Напишем простую систему обмена сообщениями, в которой несколько клиентов могут добавлять комментарии, а один клиент (аналитик) извлекает сообщения из очереди. Мы хотели бы, чтобы аналитик ожидал появления новых комментариев и извлекал их по мере поступления. Redis предлагает для решения подобных задач несколько блокирующих команд.

Откройте еще один терминал и запустите в нем клиент `redis-cli`. Этот экземпляр будет нашим аналитиком. Команда, которая ожидает поступления значения, называется `BRPOP`. Ей необходимо передать ключ списка, из которого извлекать значение, и таймаут в секундах; ниже мы задали пять минут.

```
redis 127.0.0.1:6379> BRPOP comments 300
```

Теперь перейдите на первую консоль и поместите сообщение в список `comments`.

```
redis 127.0.0.1:6379> LPUSH comments "Prag is great! I buy all my books there."
```

Вернувшись на консоль аналитика, вы увидите, что команда вернула две строки: ключ и извлеченное значение. Кроме того, на консоли печатается время, проведенное в ожидании.

```
1) "comments"  
2) "Prag is great! I buy all my books there."  
(50.22s)
```

Существует также блокирующая версия операции `pop` слева (`BLPOP`) и комбинированной операции «`pop` справа, `push` слева» (`BRPOPLPUSH`).

Множество

Наш сократитель URL постепенно оформляется, но хорошо было бы добавить еще средство группировки URL по какому-нибудь общему признаку.

Множеством называется неупорядоченная коллекция без дубликатов. Они отлично подходят для выполнения таких операций над значениями двух и более ключей, как объединение и пересечение.

Чтобы разбить URL на группы с общим ключом, мы можем создать множество и добавить в него несколько значений командой `SADD`.

```
redis 127.0.0.1:6379> SADD news nytimes.com pragprog.com  
(integer) 2
```

Redis добавил два значения. Команда `SMEMBERS` позволяет извлечь всё множество, не гарантируя какого-то определенного порядка.

```
redis 127.0.0.1:6379> SMEMBERS news  
1) "pragprog.com"  
2) "nytimes.com"
```

Добавим еще одну категорию *tech* для сайтов технической направленности.

```
redis 127.0.0.1:6379> SADD tech pragprog.com apple.com  
(integer) 2
```

Чтобы найти, какие сайты одновременно публикуют новости и имеют техническую тематику, выполним пересечение обоих множеств командой `SINTER`.

```
redis 127.0.0.1:6379> SINTER news tech  
1) "pragprog.com"
```

Ничуть не сложнее удалить из одного множества значения, встречающиеся в другом. Чтобы найти все новостные, но не технические сайты, воспользуемся командой `SDIFF`:

```
redis 127.0.0.1:6379> SDIFF news tech  
1) "nytimes.com"
```

Можно также построить объединение сайтов – то есть множество как новостных, так и технических сайтов. Поскольку это множество, дубликаты автоматически удаляются.

```
redis 127.0.0.1:6379> SUNION news tech  
1) "apple.com"  
2) "pragprog.com"  
3) "nytimes.com"
```

Это множество значений можно сразу же сохранить в новом множестве (`SUNIONSTORE destination key [key ...]`).

```
redis 127.0.0.1:6379> SUNIONSTORE websites news tech
```

С помощью этой команды можно реализовать полезный прием: копирование значений одного ключа в другой ключ – `SUNIONSTORE news_copy news`. Аналогичные команды существуют для сохранения результатов пересечения (`SINTERSTORE`) и разности (`SDIFFSTORE`).

Если команда `RPOPLPUSH` перемещает значения из списка в список, то `SMOVE` делает то же самое для множеств, только ее название проще запомнить.

И как `lLEN` возвращает длину списка, так `SCARD` (кардинальное число множества) подсчитывает число элементов в множестве; правда, это название запомнить труднее.

Поскольку множества не упорядочены, не существует команд для выполнения операций слева, справа или вообще с каким-либо указанием позиции. Для извлечения случайного значения из множества достаточно просто команды `SPOP key`, а для удаления значений – команды `SREM key value [value ...]`.

В отличие от списков, блокирующих команд для множеств не существует.

Отсортированные множества

Рассмотренные до сих пор типы данных Redis легко отображаются на стандартные конструкции языков программирования. Отсортированные же множества заимствуют всего понемножку у прочих типов. Они упорядочены, как списки, и не содержат дубликатов, как множества. В них хранятся пары поле-значение, как в хешах, но поле – не строка, а число, обозначающее порядок следования значения в множестве. Можно считать, что отсортированное множество – аналог очереди с приоритетами и с произвольным доступом. Но у такой гибкости есть цена. Поскольку отсортированные множества поддерживают упорядоченность значений, то вставка элемента производится за время $\log(N)$ (где N – размер множества), а не за постоянное время, как в случае хешей и списков.

Далее мы хотим отслеживать популярность конкретных коротких адресов. Всякий раз, как кто-то посещает URL-адрес, его оценка увеличивается. Как и в случае хеша, для добавления элемента в отсортированное множество требуется указать ключ Redis и два значения: оценку и сам элемент.

```
redis 127.0.0.1:6379> ZADD visits 500 7wks 9 gog 9999 prag
(integer) 3
```

Чтобы увеличить оценку, мы можем либо вновь добавить элемент, указав новое значение, – при этом оценка обновится, но новое значение не добавится, – либо увеличить ее на некоторую величину, в результате чего будет возвращено новое значение.

```
redis 127.0.0.1:6379> ZINCRBY visits 1 prag
"10000"
```

Чтобы уменьшить значение, нужно точно так же, с помощью команды `ZINCRBY`, прибавить отрицательную величину.

Диапазоны

Для получения значений из нашего множества посещений можно выполнить команду `ZRANGE`, которая возвращает диапазон, заданный своими границами, — как и команда списка `LRange`. Но только в случае отсортированного множества диапазон еще и упорядочивается по величине оценки в порядке возрастания. Таким образом, чтобы найти два самых посещаемых сайта (нумерация начинается с 0), выполните следующую команду:

```
redis 127.0.0.1:6379> ZRANGE visits 0 1
1) "gog"
2) "7wks"
```

Чтобы получить еще и оценки для каждого элемента, добавьте уточнение `WITHSCORES`. А если требуется отсортировать элементы в порядке убывания, воспользуйтесь командой со словом `REV`, например `ZREVRANGE`.

```
redis 127.0.0.1:6379> ZREVRANGE visits 0 -1 WITHSCORES
1) "prag"
2) "10000"
3) "7wks"
4) "500"
5) "gog"
6) "9"
```

Но раз мы выбрали отсортированное множество, то, наверное, хотим запрашивать диапазон значений оценки, а не позиций. Команда `ZRANGEBYSCORE` имеет несколько иной синтаксис, чем `ZRANGE`. По умолчанию нижняя и верхняя граница *включаются*; чтобы их исключить, следует в начале диапазона поставить открывающую скобку: `(`. Таким образом, следующая команда вернет все оценки, для которых $9 \leq \text{score} \leq 10000$:

```
redis 127.0.0.1:6379> ZRANGEBYSCORE visits 9 9999
1) "gog"
2) "7wks"
```

А такая команда вернет оценки, для которых $9 < \text{score} \leq 10000$:

```
redis 127.0.0.1:6379> ZRANGEBYSCORE visits (9 9999
1) "7wks"
```

Можно также задавать диапазоны с бесконечными границами. Следующая команда вернет всё множество.

```
redis 127.0.0.1:6379> ZRANGEBYSCORE visits -inf inf
```



Чтобы перечислить значения в обратном порядке, воспользуйтесь командой `ZREVRANGEBYSCORE`.

Помимо поиска диапазона значений по рангу (индексу) или оценке, существуют также соответствующие команды удаления диапазона: `ZREMRANGEBYRANK` и `ZREMRANGEBYSCORE`.

Объединения

Как и для множеств, можно создать новый ключ, который будет содержать объединение или пересечение одного или более ключей. Это более сложная команда, потому что она должна не только объединить ключи – эта операция сравнительно проста, – но и пересчитать возможно различающиеся оценки. Операция объединения записывается так:

```
ZUNIONSTORE destination numkeys key [key ...]
[WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]
```

Здесь `destination` – ключ, в котором сохраняется результат, а `key` – один или несколько объединяемых ключей. Параметр `numkeys` задает количество объединяемых ключей, а параметр `weight` – необязательное число (весовой коэффициент), на которое следует умножить оценку в соответственном ключе (если есть два ключа, то должно быть и два веса). Наконец, `aggregate` – необязательное правило агрегирования взвешенных оценок; по умолчанию производится суммирование, но можно задать также вычисление минимума или максимума.

Воспользуемся этой командой для вычисления меры важности отсортированного множества коротких адресов.

Сначала создадим новый ключ, в котором будут храниться оценки коротких адресов, вычисленные по количеству голосов. Каждый посетитель сайта, может проголосовать за или против сайта, и его голос добавляется к итоговой оценке.

```
redis 127.0.0.1:6379> ZADD votes 2 7wks 0 gog 9001 prag
(integer) 3
```

Мы хотим найти самые важные из зарегистрированных в системе сайтов, применяя для оценки комбинацию количества голосов и количества посещений. Голоса важнее, но и посещения дают какой-то вклад, хотя и меньший (часто человек бывает настолько очарован сайтом, что просто забывает проголосовать). Мы хотим скомбинировать обе оценки и вычислить на их основе оценку важности сайта. Для этого назначим количеству голосов вдвое больший вес, чем количеству посещений.

```
ZUNIONSTORE importance 2 visits votes WEIGHTS 1 2 AGGREGATE SUM
(integer) 3
redis 127.0.0.1:6379> ZRANGEBYSCORE importance -inf inf WITHSCORES
1) "gog"
2) "9"
3) "7wks"
4) "504"
5) "prag"
6) "28002"
```

У этой команды есть и другие применения. Например, чтобы удвоить все оценки в множестве, мы можем построить объединение с единственным ключом, задав вес 2 и сохранив результат в исходном множестве.

```
redis 127.0.0.1:6379> ZUNIONSTORE votes 1 votes WEIGHTS 2
(integer) 2
redis 127.0.0.1:6379> ZRANGE votes 0 -1 WITHSCORES
1) "gog"
2) "0"
3) "7wks"
4) "4"
5) "prag"
6) "18002"
```

Для отсортированных множеств имеется аналогичная команда (`ZINTERSTORE`), вычисляющая пересечение.

Срок хранения

Хранилища ключей и значений типа Redis часто применяются как быстрый кэш для хранения данных, которые трудно или долго вычислять каждый раз заново. Задание срока хранения позволяет избежать неограниченного роста множества ключей, поскольку Redis автоматически удаляет пару ключ-значение по истечении указанного времени.

Чтобы задать срок хранения ключа, нам понадобится команда `EXPIRE`, которой передается уже существующий ключ и время его жизни в секундах. Ниже мы задаем срок хранения 10 секунд. Если в течение этого промежутка проверить существование ключа с помощью команды `EXISTS`, то мы получим в ответ 1 (true). Но если немного подождать, то в конечном итоге команда вернет 0 (false).

```
redis 127.0.0.1:6379> SET ice "I'm melting..."
OK
redis 127.0.0.1:6379> EXPIRE ice 10
(integer) 1
redis 127.0.0.1:6379> EXISTS ice
```

```
(integer) 1
redis 127.0.0.1:6379> EXISTS ice
(integer) 0
```

Установка и задание срока хранения ключей – настолько частая операция, что в Redis есть для нее специальная команда SETEX.

```
redis 127.0.0.1:6379> SETEX ice 10 "I'm melting..."
```

Команда TTL позволяет запросить время жизни ключа. Если задать срок хранения ключа ice, как показано выше, а затем проверить его TTL, то мы узнаем, сколько секунд ему осталось жить.

```
redis 127.0.0.1:6379> TTL ice
(integer) 4
```

В любой момент до истечения срока хранения таймаут можно отменить, выполнив команду PERSIST ключ.

```
redis 127.0.0.1:6379> PERSIST ice
```

Можно также указать не относительное (в виде числа секунд, начиная с текущего момента), а абсолютное время удаления ключа. Для этого предназначена команда EXPIREAT, которая принимает временную метку Unix (количество секунд с полуночи 1 января 1970).

Типичный прием, позволяющий хранить в кэше только часто используемые ключи, состоит в том, чтобы обновлять срок хранения при каждом извлечении значения. Этот алгоритм, называемый MRU (most recently used – последние использованные) гарантирует, что востребованные ключи Redis оставит, а те, к которым обращения производятся редко, по истечении таймаута удалит обычным порядком.

Пространства имен

До сих пор мы имели дело только с одним пространством имен. Но иногда требуется размещать ключи в разных пространствах имен – по аналогии с сегментами в Riak. Например, если вы пишете интернационализированное хранилище ключей и значений, то можете хранить переводы ответов на разные языки в разных пространствах имен. В немецком пространстве имен ключу greeting (приветствие) будет сопоставлено значение «guten tag», а во французском – «bonjour». После того как пользователь выберет предпочтительный язык, приложение будет извлекать значения из соответствующего пространства имен.

В терминологии Redis пространство имен называется *базой данных*, и каждому такому пространству сопоставляется числовой ключ.

Пока что мы работали с подразумеваемым по умолчанию пространством имен 0 (оно также называется базой данных 0). Следующие команды сопоставляют ключу `greeting` английское слово `hello`.

```
redis 127.0.0.1:6379> SET greeting hello
OK
redis 127.0.0.1:6379> GET greeting
"hello"
```

Но если переключиться на другую базу данных командой `SELECT`, то этот ключ станет недоступен.

```
redis 127.0.0.1:6379> SELECT 1
OK
redis 127.0.0.1:6379[1]> GET greeting
(nil)
```

А присваивание ему значения в новом пространстве имен не изменит прежнего значения.

```
redis 127.0.0.1:6379[1]> SET greeting "guten tag"
OK
redis 127.0.0.1:6379[1]> SELECT 0
OK
redis 127.0.0.1:6379> GET greeting
"hello"
```

Поскольку все базы данных работают в одном и том же экземпляре сервера, то Redis позволяет перебрасывать ключи с помощью команды `MOVE`. Следующие команды перемещают ключ `greeting` в базу данных 2.

```
redis 127.0.0.1:6379> MOVE greeting 2
(integer) 2
redis 127.0.0.1:6379> SELECT 2
OK
redis 127.0.0.1:6379[2]> GET greeting
"hello"
```

Это бывает полезно, когда требуется исполнять на одном сервере Redis разные приложения, разрешая им тем не менее обмениваться данными.

И это еще не всё

В Redis есть много других команд, например: переименование ключей (`RENAME`), определение типа значения ключа (`TYPE`) и удаление пары ключ-значение (`DEL`). Имеется также весьма опасная команда `FLUSHDB`, которая удаляет все ключи из одной базы данных Redis, и

ее апокалиптическая родственница – команда `FLUSHALL`, удаляющая все ключи из всех баз данных. Полный перечень команд Redis можно найти в онлайн-официальной документации.

День 1: итоги

Многообразие типов данных в Redis и поддержка сложных запросов превращают эту систему в нечто гораздо большее, чем стандартное хранилище ключей и значений. Она может выступать в роли стека, очереди или очереди с приоритетами, служить хранилищем объектов (благодаря хешам) и даже умеет выполнять сложные операции над множествами (объединение, пересечение и разность). В системе имеется много атомарных команд и механизм транзакций для выполнения многошаговых команд. Встроена также возможность задания срока хранения ключей, позволяющая использовать хранилище в качестве кэша.

День 1: домашнее задание

Информационный поиск

1. Найдите полную документацию по командам Redis, где приведена в частности оценка сложности в нотации «О-большое» ($O(x)$).

Задачи

1. Установите драйвер своего любимого языка программирования и подключитесь к серверу Redis. Вставьте и инкрементируйте числовое значение в рамках одной транзакции.
2. Пользуясь драйвером по своему выбору, напишите программу, которая читает данные из блокирующего списка и куда-то выводит их (на консоль, в файл, в канал `Socket.io` и т. д.), а также другую программу, которая помещает данные в этот список.

8.3. День 2: более сложные применения, распределенные вычисления

В первый день мы рассматривали Redis как сервер структур данных. Сегодня мы продолжим возводить здание на этом фундаменте и познакомимся с некоторыми более продвинутыми функциями Redis, в

частности, с конвейерами, моделью публикации-подписки, настройкой системы и репликацией. Кроме того, мы научимся строить кластер Redis, быстро сохранять большие массивы данных и использовать фильтры Блума.

Простой интерфейс

Насчитывающий всего 20 000 строк исходного кода, Redis является довольно простым проектом. И его интерфейс также прост – в том смысле, что принимает буквально те строки, которые вводятся на консоли.

Telnet

Мы можем взаимодействовать с Redis без всякого командного интерфейса, просто посылая команды по TCP-соединению или через telnet. Каждая команда должна завершаться знаками возврата каретки и перехода на новую строку (CRLF, или `\r\n`).

redis/telnet.sh

```
$ telnet localhost 6379
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
SET test hello
❶ +OK
GET test
❷ $5
hello
SADD stest 1 99
❸ :2
SMEMBERS stest
❹ *2
$1
1
$2
99
CTRL-]
```

Как видите, мы вводим те же самые команды, что при работе с консолью, только консоль производила незначительное форматирование ответов.

- ❶ Redis возвращает состояние `OK`, добавляя в начало префикс `+`.
- ❷ Перед тем как вернуть строку `hello`, Redis послал `$5`, что означает «длина следующей строки составляет 5 символов».

- ③ Числу 2, возвращенному после добавления двух значений в ключ `test`, предшествует знак `:`, показывающий, что это целое число (успешно были добавлены два значения).
- ④ Запросив два значения, мы получили в первой строке число 2, которому предшествует звездочка, – это означает, что далее следуют два составных значения. Следующие две строки такие же, как при возврате *hello*, только в первой мы видим число 1, а во второй – строку 99.

Конвейеры

Можно также передавать строки по одной, воспользовавшись программой `netcat` (`nc`), которая первоначально была написана для операционной системы BSD, но теперь включается по умолчанию во многие дистрибутивы Unix. В этом случае мы должны явно завершать каждую строку знаками CRLF (`telnet` это делает автоматически). Мы также ожидаем в течение одной секунды после появления введенной команды на консоли, чтобы дать серверу Redis время для ответа. В некоторых, но не во всех реализациях `nc` имеется флаг `-q`, позволяющий обойтись без ожидания, – проверьте.

```
$ (echo -en "ECHO hello\r\n"; sleep 1) | nc localhost 6379
$5
hello
```

Этим можно воспользоваться, чтобы организовать *конвейер* команд, то есть посылать несколько команд в одном запросе.

```
$ (echo -en "PING\r\nPING\r\nPING\r\n"; sleep 1) | nc localhost 6379
+PONG
+PONG
+PONG
```

Это гораздо эффективнее, чем отправлять по одной команде, поэтому такой вариант всегда следует иметь в виду – особенно при работе с транзакциями. Не забывайте только завершать каждую команду символами `\r\n`, которые сервер интерпретирует как разграничитель.

Публикация-подписка

Вчера нам удалось реализовать примитивную блокирующую очередь с помощью списка. Мы помещали в очередь данные, которые считывались блокирующей командой `pop`. Это позволило нам построить очень простую модель публикации-подписки. Сообщения в очередь могут помещать различные клиенты, а извлекает их единственный читатель по мере поступления. Это хорошо, но недостаточно. Во мно-

гих случаях нам необходимо обратное поведение, когда несколько подписчиков желают читать сообщения, опубликованные одним издателем, как показано на рис. 37. Redis предлагает несколько специализированных команд для организации публикации-подписки.

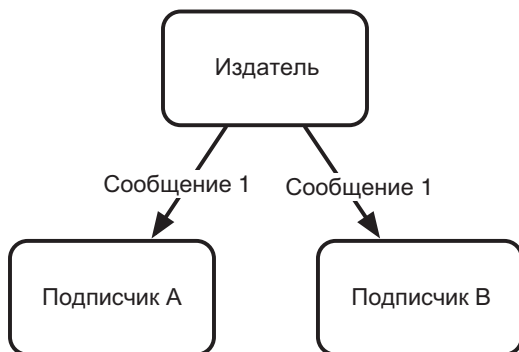


Рис. 37. Издатель отправляет сообщение всем подписчикам

Усовершенствуем механизм комментирования, разработанный вчера с помощью блокирующих списков, позволив пользователю отправлять комментарий нескольким подписчикам (а не только одному). Начнем с подписчиков, которые устанавливают соединение с ключом, который в терминологии публикации-подписки называется *каналом*. Запустите еще двух клиентов и подпишитесь в них на канал комментариев. Подписка приводит к блокировке командного интерфейса.

```
redis 127.0.0.1:6379> SUBSCRIBE comments
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "comments"
3) (integer) 1
```

Имея двух подписчиков, мы можем опубликовать через канал `comments` произвольное строковое сообщение. Команда `PUBLISH` вернет число 2, означающее, что сообщение было получено двумя подписчиками.

```
redis 127.0.0.1:6379> PUBLISH comments "Check out this shortcoded site! 7wks"
(integer) 2
```

Оба подписчика получают *многочастный ответ* (multibulk reply) (список), состоящий из трех элементов: строка «message», имя канала и само опубликованное сообщение.

- 1) "message"
- 2) "comments"
- 3) "Check out this shortcoded site! 7wks"

Когда клиент потеряет интерес к получению корреспонденции, он может выполнить команду `UNSUBSCRIBE comments`, чтобы отключиться от канала `comments`, или команду `UNSUBSCRIBE` без параметров – чтобы отключиться от всех каналов. Отметим, однако, что в оболочке `redis-cli` нужно нажать `CTRL+C` для разрыва соединения.

Информация о сервере

Перед тем как изменять системные настройки Redis, полезно посмотреть, что выдает команда `INFO`, так как в результате изменения настроек изменяются и некоторые возвращаемые ей значения. `INFO` выводит сведения о сервере, в том числе номер версии, идентификатор процесса, объем занятой памяти и время с момента последнего перезапуска.

```
redis 127.0.0.1:6379> INFO
redis_version:2.4.5
redis_git_sha1:00000000
redis_git_dirty:0
arch_bits:64
multiplexing_api:kqueue
process_id:54046
uptime_in_seconds:4
uptime_in_days:0
lru_clock:1807217
...
```

Рекомендуем вернуться к этой команде позже, поскольку она дает полезный мгновенный снимок состояния сервера и его настроек. Выводится даже информация о долговечности, фрагментации памяти и состоянии репликации

Настройка Redis

До сих пор мы запускали Redis с настройками по умолчанию. Но в значительной мере Redis обязан своей мощью гибкости конфигурирования, позволяющей адаптировать его к конкретному применению. Файл `redis.conf`, входящий в состав дистрибутива – в системах `*nix` он устанавливается в каталог `/etc/redis` – практически не нуждается в пояснениях, поэтому мы рассмотрим только его часть. Расскажем о некоторых наиболее употребительных параметрах по порядку.

```
daemonize no
port 6379
loglevel verbose
logfile stdout
database 16
```

По умолчанию параметр `daemonize` равен *no*, поэтому-то сервер и запускается в приоритетном режиме. Для тестирования это удобно, но в производственной системе – не очень. Если задать значение *yes*, то сервер будет работать в фоновом режиме и запишет идентификатор своего процесса в `pid`-файл.

В следующей строке задается номер порта сервера – 6379. Эта настройка особенно полезна при запуске нескольких экземпляров Redis на одной машине. Параметр `loglevel` по умолчанию равен *verbose* (подробно), но в производственной системе лучше задать режим *notice* или *warning*. Параметр `logfile` настроен так, что диагностическая информация печатается на `stdout` (стандартный вывод, то есть консоль), но в режиме демона лучше указать имя файла журнала.

Параметр `database` задает количество доступных баз данных Redis. О том, как переключать базу данных, мы говорили вчера. Если вы планируете использовать только одно пространство имен, то лучше установить этот параметр в 1.

Долговечность

Redis поддерживает несколько режимов сохранения. Прежде всего, это отсутствие сохранения вообще, когда все значения хранятся в оперативной памяти. Если сервер используется для кэширования, то это разумный выбор, потому что за долговечность приходится расплачиваться задержками. Одна из особенностей Redis, отличающих ее от других быстрых кэшей, например `memcached`², – встроенная поддержка хранения значений на диске. По умолчанию пары ключ-значение сохраняются лишь периодически. Команда `LASTSAVE` возвращает временную метку Unix, соответствующую последней успешной записи на диск. То же значение можно прочитать из поля `last_save_time`, возвращаемого командой `INFO`.

Принудительно сбросить данные на диск позволяет команда `SAVE` (или `BGSAVE`, которая производит сохранение асинхронно в фоновом режиме).

```
redis 127.0.0.1:6379> SAVE
```

После этого в журнале сервера появятся строки вида:

² <http://www.memcached.org/>

```
[46421] 10 Oct 19:11:50 * Background saving started by pid 52123
[52123] 10 Oct 19:11:50 * DB saved on disk
[46421] 10 Oct 19:11:50 * Background saving terminated with success
```

Еще один способ обеспечения долговечности – изменение параметров мгновенных снимков в конфигурационном файле.

Мгновенные снимки

Мы можем изменить частоту сохранения на диск, добавив, удалив или изменив одно из полей сохранения. По умолчанию таких полей 3, и все они начинаются ключевым словом `save`, за которым следует время в секундах и минимальное количество ключей, которые должны быть изменены, чтобы началась запись на диск.

Например, чтобы выполнять сохранение каждые 5 минут (300 секунд) при условии, что изменился хотя бы один ключ, нужно написать:

```
save 300 1
```

В конфигурационном файле настроены разумные умолчания, а именно: если изменилось 10 000 ключей, сохранять раз в 60 секунд; если изменилось 10 ключей, сохранять раз в 300 секунд; если изменился хотя бы один ключ, сохранять не реже, чем раз в 900 секунд (15 минут).

```
save 900 1
save 300 10
save 60 10000
```

Можно добавить дополнительные поля сохранения, определяющие более точные пороги.

Файл с дозаписью

По умолчанию Redis обеспечивает *долговечность в конечном счете*, то есть асинхронно сбрасывает значения на диск с интервалами, определяемыми настройками сохранения, или по явной команде от клиента. Это приемлемо для кэша второго уровня или сервера сеансов, но недостаточно, когда долговечность данных – неперемное условие, например при работе с финансовыми данными. Пользователи не поймут вас, потеряв деньги из-за аварийного завершения сервера Redis.

Redis поддерживает файл с дозаписью (`appendonly.aof`), в котором сохраняются все команды записи. Это подобие протоколирования с упреждающей записью, о котором шла речь в главе 4, пос-

вященной HBase. Если сервер «грохнется», не записав значение, то при перезапуске он выполнит сохраненные команды, восстановив последнее состояние. Чтобы активировать этот режим, следует присвоить значение `yes` параметру `appendonly` в файле `redis.conf`.

```
appendonly yes
```

Далее мы должны решить, как часто сбрасывать команды в файл. Режим `always` обеспечивает максимальную надежность, так как каждая команда сохраняется сразу же. Но он и самый медленный, что обеспечивает те преимущества, из-за которых люди выбирают Redis. По умолчанию устанавливается режим `everysec`, в котором команды записываются раз в секунду. Это приемлемый компромисс, так как, с одной стороны, сервер работает достаточно быстро, а, с другой, вы в худшем случае потеряете только изменения, произошедшие в течение одной секунды. Наконец, в режиме `no` сбросом на диск управляет операционная система. Но это может происходить не слишком часто, так что файл дозаписи теряет смысл и, быть может, его тогда лучше вообще не использовать.

```
# appendfsync always
appendfsync everysec
# appendfsync no
```

У режима дозаписи есть и другие параметры, о которых можно прочесть в самом конфигурационном файле. При определенных способах эксплуатации они могут оказаться полезны.

Безопасность

Вообще-то Redis не проектировался как абсолютно безопасный сервер, но в документации вы можете наткнуться на описание параметра `requirepass` и команды `AUTH`. Их можно благополучно игнорировать, так как поддерживается лишь задание пароля в открытом виде. Поскольку клиент может проверить почти 100 000 паролей в секунду, то игра не стоит свеч, и это даже не говоря о том, что открытые пароли в любом случае небезопасны. Если требуется обеспечить безопасность Redis, то лучше установить достойный брандмауэр и настроить SSH.

Интересно, что Redis поддерживает безопасность на уровне команд путем запутывания, позволяя скрывать или подавлять команды. Следующая команда превращает команду `FLUSHALL` (удалить все ключи из системы) в трудно угадываемую строку `c283d93ac9528f986023793b411e4ba2:`

```
rename-command FLUSHALL c283d93ac9528f986023793b411e4ba2
```

Если теперь отправить серверу команду FLUSHALL, он вернет ошибку. Напротив, секретная команда работает.

```
redis 127.0.0.1:6379> FLUSHALL
(error) ERR unknown command 'FLUSHALL'
redis 127.0.0.1:6379> c283d93ac9528f986023793b411e4ba2
OK
```

Более того, мы можем вообще запретить некоторые команды, переименовав их в пустую строку:

```
rename-command FLUSHALL ""
```

Таким образом, можно подавить сколько угодно команд, что позволяет организовать некое подобие специализированного окружения.

Дополнительные параметры

Существует ряд более специальных параметров для ведения журнала медленных запросов, задания кодировки, настройки задержки и импорта внешних конфигурационных файлов. Но предупреждаем – встретив в документации упоминания о виртуальной памяти Redis, не обращайтесь на них внимания, так как в версии 2.4 они объявлены нерекомендуемыми и впоследствии могут быть удалены.

Для исследования конфигурации сервера Redis предлагает отличный инструмент эталонного тестирования. По умолчанию он подключается к порту 6379 на локальной машине и отправляет 10 000 запросов от имени 50 параллельно работающих клиентов. Увеличить количество запросов до 100 000 можно с помощью аргумента -n.

```
$ redis-benchmark -n 100000
===== PING (inline) =====
100000 requests completed in 3.05 seconds
50 parallel clients
3 bytes payload
keep alive: 1
5.03% <= 1 milliseconds
98.44% <= 2 milliseconds
99.92% <= 3 milliseconds
100.00% <= 3 milliseconds
32808.40 requests per second
...
```

Можно протестировать и другие команды, например SADD и LRange; но чем команда сложнее, тем больше времени она исполняется.

Репликация главный-подчиненный

Как и другие рассмотренные выше базы данных NoSQL (например, MongoDB и Neo4j), Redis поддерживает репликацию в режиме главный-подчиненный. По умолчанию сервер является главным, если не подчинить его какому-то другому. Данные реплицируются на произвольное число подчиненных серверов.

Настроить подчиненные серверы несложно. Для начала понадобится скопировать файл `redis.conf`.

```
$ cp redis.conf redis-sl.conf
```

В файл нужно внести всего два изменения:

```
port 6380
slaveof 127.0.0.1 6379
```

Если всё пойдет по плану, то при запуске подчиненного сервера в его журнале появятся строки вида:

```
$ redis-server redis-sl.conf
[9003] 16 Oct 23:51:52 * Connecting to MASTER...
[9003] 16 Oct 23:51:52 * MASTER <-> SLAVE sync started
[9003] 16 Oct 23:51:52 * Non blocking connect for SYNC fired the event.
[9003] 16 Oct 23:51:52 * MASTER <-> SLAVE sync: receiving 28 bytes from master
[9003] 16 Oct 23:51:52 * MASTER <-> SLAVE sync: Loading DB in memory
[9003] 16 Oct 23:51:52 * MASTER <-> SLAVE sync: Finished with success
```

А в журнале главного сервера – строка 1 slaves. Если выполнить на главном сервере команду

```
redis 127.0.0.1:6379> SADD meetings "StarTrek Pastry Chefs" "LARPers Intl."
```

а потом подключиться к подчиненному, то можно будет получить список встреч:

```
redis 127.0.0.1:6380> SMEMBERS meetings
1) "StarTrek Pastry Chefs"
2) "LARPers Intl."
```

В производственной система репликация обычно включается для повышения доступности или резервного копирования, поэтому подчиненные серверы запускаются на разных машинах.

Загрузка данных

Мы много говорили о том, какая Redis быстрая система, но прочувствовать это, не имея солидного массива данных, сложно.

Давайте загрузим на сервер Redis большой набор данных. Если хотите, можете не останавливать подчиненный сервер, но настоль-

ный компьютер или ноутбук будут работать быстрее, когда запущен только главный сервер. Мы собираемся скачать с сайта Freebase.com список, содержащий более 2,5 миллионов названий изданных книг, идентифицируемых международным стандартным книжным номером (ISBN)³.

Предварительно установите `gem`-пакет `redis` для Ruby.

```
$ gem install redis
```

Существует несколько способов вставить большой набор данных. Они варьируются по скорости и соответственно по сложности.

Самый простой метод – просто обойти в цикле весь список данных и выполнить для каждого значения команду `SET` с помощью стандартной клиентской библиотеки `redis-rb`.

redis/isbn.rb

```
LIMIT = 1.0 / 0 # 1.0/0 в Ruby обозначает бесконечность
# %w{rubygems hiredis redis/connection/hiredis}.each{|r| require r}
#w{rubygems time redis}.each{|r| require r}

$redis = Redis.new(:host => "127.0.0.1", :port => 6379)
$redis.flushall
count, start = 0, Time.now
File.open(ARGV[0]).each do |line|
  count += 1
  next if count == 1
  isbn, _, _, title = line.split("\t")
  next if isbn.empty? || title == "\n"

  $redis.set(isbn, title.strip)

  # установите значение LIMIT, если не хотите загружать весь набор данных
  break if count >= LIMIT
end

puts "#{count} элементов за #{Time.now - start} секунд"

$ ruby isbn.rb isbn.tsv
2456384 элементов за 266.690189 секунд
```

Чтобы ускорить вставку и не использовать JRuby, можно установить дополнительный `gem`-пакет `hiredis`. Это написанный на C драйвер, который работает значительно быстрее драйвера на чистом Ruby. Затем раскомментируйте строку `hiredis require`, чтобы загрузить драйвер. Возможно, вы не заметите серьезного улучшения

3 <http://download.freebase.com/datadumps/latest/browse/book/isbn.tsv>

на такого рода задаче, загружающей прежде всего процессор, но при использовании Ruby в производственной системе мы настоятельно рекомендуем установить hiredis.

Заметного ускорения можно добиться за счет конвейеризации. Следующий скрипт создает пакет из 1000 строк и вставляет его целиком, передавая конвейеру. В результате время вставки на нашей машине уменьшилось более чем в три раза.

redis/isbn_pipelined.rb

```
BATCH_SIZE = 1000
LIMIT = 1.0 / 0 # 1.0/0 в Ruby обозначает бесконечность

# %w{rubygems hiredis redis/connection/hiredis}.each{|r| require r}
# %w{rubygems time redis}.each{|r| require r}

$redis = Redis.new(:host => "127.0.0.1", :port => 6379)
$redis.flushall

# отправить данные в виде одного пакета обновления
def flush(batch)
  $redis.pipelined do
    batch.each do |saved_line|
      isbn, _, _, title = line.split("\t")
      next if isbn.empty? || title == "\n"
      $redis.set(isbn, title.strip)
    end
  end
  batch.clear
end

batch = []
count, start = 0, Time.now
File.open(ARGV[0]).each do |line|
  count += 1
  next if count == 1

  # поместить строки в массив
  batch << line

  # когда в массиве окажется BATCH_SIZE элементов, сбросить его
  if batch.size == BATCH_SIZE
    flush(batch)
    puts "#{count-1} элементов"
  end

  # установите значение LIMIT, если не хотите загружать весь набор данных
  break if count >= LIMIT
end

# сбросить оставшиеся значения
```

```
flush(batch)
```

```
puts "#{count} элементов за #{Time.now - start} секунд"
```

```
$ ruby isbn_pipeline.rb isbn.tsv
2666642 элементов за 79.312975 секунд
```

Таким образом мы уменьшили количество соединений с Redis, но у построения набора данных для передачи по конвейеру имеются собственные накладные расходы. Организуя конвейер в производственной системе, поэкспериментируйте, задавая разное количество операций в одном пакете.

Для пользователей Ruby попутно отметим, что если приложение работает в неблокирующем режиме, применяя Event Machine, то драйвер Ruby может задействовать `em-synchrony` за счет вызова `EM::Protocols::Redis.connect`.

Кластер Redis

Помимо простой репликации, многие клиенты Redis предоставляют интерфейс для построения простого «рукодельного» распределенного кластера. Написанный на Ruby клиент `redis-rb` поддерживает управляемый кластер с согласованным хешированием. Возможно, вы еще помните, как мы обсуждали согласованное хеширование в главе, посвященной Riak, где можно добавлять и удалять узлы без устаревания большинства ключей. Здесь идея такая же, только кластер управляется клиентом, а не самими серверами.

Для начала нам понадобится еще один сервер. В отличие от репликации главный-подчиненный, здесь оба сервера играют роль главного (как в конфигурации по умолчанию). Мы просто скопировали файл `redis.conf` и заменили номер порта на 6380. Больше с серверами ничего делать не надо.

redis/isbn_cluster.rb

```
LIMIT = 10000
%w{rubygems time redis}.each{|r| require r}
require 'redis/distributed'

$redis = Redis::Distributed.new([
  "redis://localhost:6379/", "redis://localhost:6380/"
])
$redis.flushall

count, start = 0, Time.now
File.open(ARGV[0]).each do |line|
  count += 1
```

```
next if count == 1
isbn, _, _ , title = line.split("\t")
next if isbn.empty? || title == "\n"

$redis.set(isbn, title.strip)

# установите значение LIMIT, если не хотите загружать весь набор данных
break if count >= LIMIT
end
puts "#{count} items in #{Time.now - start} seconds"
```

Для создания моста между двумя и более серверами пришлось внести лишь незначительные изменения в существующий клиент для загрузки набора данных о книгах. Во-первых, нужно затребовать файл `redis/distributed` из `gem`-пакета `redis`.

```
require 'redis/distributed'
```

Затем мы заменяем клиент `Redis` на `Redis::Distributed` и передаем ему массив URI-адресов серверов. Во всех URI указана схема `redis` и далее имя сервера (`localhost`) и порт.

```
$redis = Redis::Distributed.new([
  "redis://localhost:6379/",
  "redis://localhost:6380/"
])
```

Запуска клиента производится так же, как и раньше.

```
$ ruby isbn_cluster.rb isbn.tsv
```

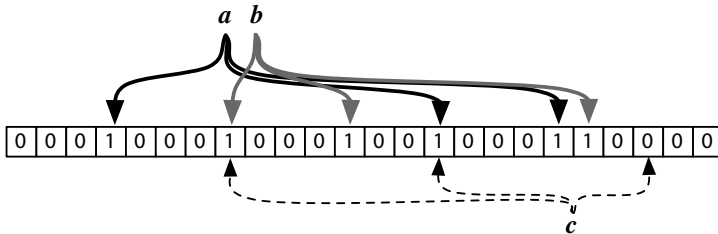
Но теперь на клиента возлагается гораздо больше работы, потому что он вычисляет, какие ключи на каких серверах хранятся. Убедиться в том, что ключи действительно хранятся на разных серверах, можно, попытавшись запросить ключ ISBN с каждого сервера в командной оболочке. В ответ на команду `GET` значение вернет только один сервер. Но если запрашивать ключи через объект `Redis::Distributed`, то клиент будет получать значение от правильного сервера.

Фильтры Блума

Стать владельцем уникального термина – замечательный способ гарантировать успешность поиска в сети. Если бы вы написали книгу «The Jabbyredis», то практически наверняка любая поисковая система указала бы на вас. Давайте напишем скрипт, который будет проверять, является ли некое слово уникальным среди всех слов, встречающихся в названиях книг из каталога ISBN. Для проверки того, встречалось ли слово ранее, можно воспользоваться фильтром Блума.

Фильтр Блума – это вероятностная структура данных, которая проверяет существование элемента в множестве. Впервые мы встретились с ней в разделе «Сжатие и фильтры Блума» в главе, посвященной HBase. Фильтр Блума может возвращать ложноположительный ответ, но никогда – ложноотрицательный. Это полезно, если требуется быстро установить, что значение не существует.

Задачу доказательства несуществования фильтр Блума решает за счет преобразования значения в сильно разреженную последовательность битов, которая сравнивается с объединением битов, соответствующих всем существующим значениям. Иными словами, при добавлении нового значения производится операция ИЛИ с текущей последовательностью битов, хранящейся в фильтре Блума. Если требуется проверить, существует ли уже данное значение, то мы выполняем операцию И с хранимой последовательностью битов. Если в данном значении поднят хотя бы один бит, который не поднят в соответствующем сегменте фильтра Блума, значит, это значение ранее не добавлялось, то есть его заведомо нет в множестве. Ниже эта идея представлена графически.



Напишем программу, которая перебирает все книги из набора ISBN, извлекает и нормализует название книги, а затем разбивает его на отдельные слова. Каждое слово проверяется по фильтру Блума. Если фильтр возвращает false, значит, слово еще не существует, и мы добавляем его в фильтр. Попутно можно выводить все встретившиеся новые слова.

```
$ gem install bloomfilter-rb
```

```
redis/isbn_bf.rb
# LIMIT = 1.0 / 0 # 1.0/0 в Ruby обозначает бесконечность
LIMIT= 10000
%w{rubygems time bloomfilter-rb}.each{|r| require r}
bloomfilter = BloomFilter::Redis.new(:size => 1000000)

$redis = Redis.new(:host => "127.0.0.1", :port => 6379)
```



```
$redis.flushall

count, start = 0, Time.now
File.open(ARGV[0]).each do |line|
  count += 1
  next if count == 1
  _, _, _, title = line.split("\t")
  next if title == "\n"

  words = title.gsub(/[^\w\s]+/, '').downcase
  # выделяем слова
  words = words.split(' ')
  words.each do |word|
    # пропустить слово, если оно уже есть в фильтре Блума
    next if bloomfilter.include?(word)
    # вывести ранее не встречавшееся слово
    puts word
    # добавить новое слово в фильтр Блума
    bloomfilter.insert(word)
  end
  # установите значение LIMIT, если не хотите загружать весь набор данных
  break if count >= LIMIT
end
puts "Содержит Jabbyredis? #{bloomfilter.include?('jabbyredis')}"
puts "#{count} строк за #{Time.now - start} секунд"
```

Этот фильтр Блума на основе Redis написал на Ruby вундеркинд Илья Григорик (Ilya Grigorik), но сама идея переносится на любой другой язык.

При запуске клиента используется тот же самый файл ISBN, но анализируем мы только названия книг.

```
$ ruby isbn_bf.rb isbn.tsv
```

Поначалу будет встречаться много общеупотребительных слов типа *and* и *the*. Но ближе к концу набора появляются все более странные слова, например *unindustria*.

Плюс этого подхода – возможность обнаруживать слова-дубликаты. А минус в том, что время от времени просачиваются ложноположительные ответы – фильтр Блума может пропустить слово, которое никогда не встречалось. Поэтому в реальных приложениях выполняется дополнительная проверка, например, по более медленной базе данных в системе, куда записываются все данные. Но это будет происходить достаточно редко в предположении, что размер фильтра достаточно велик, а этот размер можно оценить заранее⁴.

⁴ http://en.wikipedia.org/wiki/Bloom_filter

SETBIT и GETBIT

Как мы уже говорили, принцип работы фильтра Блума основан на манипуляциях с битами в разреженном битовом векторе. В реализации фильтра Блума в Redis, которой мы только что воспользовались, применяются две сравнительно недавно появившиеся команды, которые и выполняют эти манипуляции: SETBIT и GETBIT.

Как и во всех командах Redis, название SETBIT говорит само за себя. Команда устанавливает один бит в некоторой позиции битового вектора, причем позиции нумеруются с нуля. Этот способ часто применяется для быстрой пометки в многопараметрических системах – проще завести несколько битов, чем описывать всё строками.

Если бы нам понадобилось создать систему учета приправ к гамбургерам, то мы могли бы сопоставить каждой приправе позицию бита, например: кетчуп = 0, горчица = 1, лук = 2, салат-латук = 3. Тогда гамбургер с горчицей и луком можно было описать битовым вектором 0110, который в командной строке сформируется так:

```
redis 127.0.0.1:6379> SETBIT my_burger 1 1
(integer) 0
redis 127.0.0.1:6379> SETBIT my_burger 2 1
(integer) 0
```

Затем мы могли бы проверить, должны ли прилагаться к моему гамбургеру латук или горчица. Если возвращен 0, то ответ – нет, если 1 – то да.

```
redis 127.0.0.1:6379> GETBIT my_burger 3
(integer) 0
redis 127.0.0.1:6379> GETBIT my_burger 1
(integer) 1
```

В реализации фильтра Блума эта идея используется путем представления значения в виде битового вектора. При обращении к методу insert() вызывается команда SETBIT X 1 для каждой позиции X, в которой бит поднят, а при обращении к include?() существование проверяется путем вызова GETBIT X – и возвращается false, если хотя бы для одной позиции GETBIT возвращает 0.

Фильтр Блума прекрасно справляется с задачей уменьшения избыточного трафика с более медленной системой, стоящей за ним, будь это медленная база данных, ограниченный ресурс или сетевой запрос. Если имеется медленная база IP-адресов и требуется отслеживать новых пользователей сайта, то можно предварительно проверить с помощью фильтра Блума, существует ли уже IP-адрес в систе-

ме. Если фильтр вернул `false`, то можно с уверенностью сказать, что адреса еще нет, и отреагировать соответствующим образом. Если же фильтр вернул `true`, то адрес, может быть, существует, а, может быть, и нет, поэтому необходимо дополнительное обращение к основному хранилищу. Именно поэтому так важно правильно вычислить размер фильтра – при подходящем размере фильтр Блума может уменьшить частоту ошибок, то есть вероятность ложноположительного ответа (но не исключить их вовсе).

День 2: итоги

Сегодня мы продолжили изучение Redis, перейдя от простых операций к вопросу о том, как выжать максимум производительности из и без того очень быстрой системы. Вчера мы убедились, что Redis обеспечивает быстрые и гибкие структуры данных и простые манипуляции с ними, но он также успешно справляется и с более сложными задачами, предоставляя встроенные функции публикации-подписки и битовые операции. Кроме того, он настраивается в широких пределах, поддерживая разнообразные параметры долговечности и репликации, что позволяет адаптировать его к самым разным потребностям. Наконец, для Redis имеется ряд написанных сторонними разработчиками расширений, в частности фильтры Блума и кластеры.

На этом мы завершаем рассмотрение основных операций сервера структур данных Redis. Завтра мы займемся другим делом, а именно воспользуемся Redis как краеугольным камнем в системе многостороннего хранения данных, куда войдут еще CouchDB и Neo4j.

День 2: домашнее задание

Информационный поиск

1. Найдите сведения о различных паттернах обмена сообщениями и выясните, сколько из них реализованы в Redis.

Задачи

1. Запустите скрипт загрузки данных о ISBN, отключив мгновенные снимки и дозапись в файл. Затем попробуйте запустить его, задав параметр `appendfsync` равным `always`, и посмотрите на различие в скорости.
2. Возьмите свой любимый каркас для разработки веб-приложений и попытайтесь с его помощью разработать простую службу сокращения URL-адресов, используя Redis в качестве храни-

лица данных. Сайт должен предлагать поле для ввода URL и реализовывать переадресацию с короткого URL на исходный. Повысьте надежность хранения с помощью кластера Redis из нескольких узлов с репликацией главный-подчиненный.

8.4. День 3: комбинирование с другими базами данных

Сегодня мы завершим главу о последней базе данных, пригласив на съемки некоторые ранее рассмотренные системы. Но главную роль будет играть именно Redis, с помощью которой мы упростим и ускорим взаимодействие с другими базами данных.

В этой книге мы узнали, что у каждой базы данных есть свои сильные стороны, поэтому многие современные системы развиваются в направлении модели многостороннего хранения (polyglot persistence), когда используется несколько баз данных и каждая играет свою роль. Мы построим проект, в котором CouchDB будет выступать в роли системы, куда записываются исходные данные (канонический источник данных), Neo4j будет управлять связями, а Redis – использоваться для загрузки данных и кэширования. Считайте это выпускным экзаменом.

Сразу скажем, что этот проект не отражает приверженность авторов к какому-то конкретному набору баз данных, языков или каркасов. Это всего лишь демонстрация совместной работы нескольких баз данных, где особые умения каждой вносят свой вклад в достижение общей цели.

Служба многостороннего хранения

Наша служба многостороннего хранения будет работать как фронтальная система для хранения информации о музыкальных группах. Мы хотим хранить список названий групп, имена участвовавших в них исполнителей и роли каждого исполнителя в группе – от ведущего вокалиста до аккомпаниатора на клавиатуре. Каждая из трех баз данных – Redis, CouchDB и Neo4j – будет отвечать за определенные аспекты системы.

На Redis в нашей системе возлагаются три важные функции: помощь в заполнении базы CouchDB данными, кэширование недавно внесенных в Neo4j изменений и быстрый поиск по неполным значениям. Скорость Redis и способность хранить различные структуры

данных позволяют с успехом применить ее для заполнения системы данными, а встроенный механизм срока хранения – то что надо для организации кэширования.

CouchDB будет у нас системой первичной записи, то есть авторитетным (каноническим) источником данных. Поскольку CouchDB – документная база, в ней удобно хранить данные о группах, в которые вложены сведения об исполнителях и их ролях. А имеющимся в CouchDB интерфейсом Changes API мы воспользуемся для синхронизации с нашим третьим источником данных.

Neo4j будет выступать в роли хранилища связей. Хотя прямые запросы к CouchDB вполне возможны и разумны, графовая база данных намного превосходит всех конкурентов в простоте и скорости обхода связей между узлами. Мы будем хранить в ней связи между группами, их членами и ролями отдельных членов.

Восход многосторонней модели хранения

В последнее время все чаще наблюдается явление многоязыкового программирования, а теперь набирает силу и многостороннее хранение.

Для тех, кто не знаком с этой практикой, поясним, что под многоязыковым программированием понимается использование в одном проекте нескольких языков программирования. Ранее считалось нормальным разрабатывать проект на каком-то одном языке общего назначения. Но многоязыковое программирование полезно, потому что позволяет задействовать сильные стороны каждого языка. Например, язык Scala лучше приспособлен для организации транзакций без состояния на стороне веб-сервера, а на Ruby проще писать бизнес-логику. При совместном использовании возникает синергетический эффект. Одной из широко известных многоязыковых систем является Twitter. Некоторые из рассмотренных нами баз данных сами по себе поддерживают многоязыковое программирование – так Riak позволяет писать mapreduce-функции на JavaScript и Erlang, причем при обработке одного запроса они могут смешиваться.

Аналогично многостороннее хранение позволяет задействовать сильные стороны разных баз данных в одной системе, что резко контрастирует с привычной практикой использования единственной, как правило реляционной, СУБД. Самый простой вариант такой системы уже широко применяется: хранилище ключей и значений (типа Redis) используется как кэш запросов к относительно медленной реляционной базе данных (например, PostgreSQL). Как мы видели в предыдущих главах, реляционные базы далеко не оптимальны для решения широкого и постоянно расширяющегося класса задач, например обхода графа. Но даже новые базы данных – лишь одинокие, пусть и ярко сияющие, звезды в галактике требований.

Почему так внезапно возник интерес к многосторонности? Мартин Фаулер как-то отметил⁵, что раньше при проектировании программного обеспечения общепринятым подходом считалось наличие одной центральной базы

данных, обслуживающей несколько приложений. Эта когда-то популярная схема интеграции уступила место структуре с промежуточным слоем, когда приложения общаются не с базой данных, а со слоем служб по протоколу HTTP. Это дает самому промежуточному слою свободу выбора любого числа или, как в случае многостороннего хранения, любого типа баз данных.

В нашей системе у каждой базы данных своя роль, но непосредственно они между собой не контактируют. Для заполнения баз данных, организации взаимодействия между ними и в качестве простого фронтального сервера мы будем использовать каркас Node.js, основанный на JavaScript. Поскольку для сопряжения разных баз данных без программирования не обойтись, в этот последний день мы напишем больше кода, чем во всей книге.

Заполнение данными

Наша первая задача – заполнить хранилища необходимыми данными. Мы сделаем это в два шага: сначала заполним данными Redis, а потом – наш канонический источник, CouchDB.

Как и раньше, скачаем набор данных с сайта Freebase.com. На этот раз нас мы возьмем набор `group_membership` с табуляторами в качестве разделителей⁶. В этом файле очень много информации, но нас будут интересовать только поля `member` (имя исполнителя), `group` (название группы) и `roles` (роли в группе, перечисленные через запятую). Например, Джон Купер (*John Cooper*) был в группе *Skillet* ведущим вокалистом (*Lead vocalist*), басистом (*Bassist*) и играл на акустической гитаре (*Acoustic guitar*).

/m/0654bxy John Cooper Skillet Lead vocalist,Acoustic guitar,Bass 1996

В конечном итоге мы хотим поместить сведения о Джоне Купере и других участниках группы *Skillet* в один документ CouchDB с показанной ниже структурой. Это документ будет доступен по URL-адресу `http://localhost:5984/bands/Skillet`.

```
{
  "_id": "Skillet",
  "name": "Skillet"
  "artists": [
    {
      "name": "John Cooper",
      "role": [
        "Acoustic guitar",
        "Lead vocalist",
```

6 http://download.freebase.com/datadumps/latest/browse/music/group_membership.tsv

```
    "Bass"
  ],
  ...
  {
    "name": "Korey Cooper",
    "role": [
      "backing vocals",
      "Synthesizer",
      "Guitar",
      "Keyboard instrument"
    ]
  }
]
```

В файле хранятся сведения более чем о 30000 групп и 100000 исполнителей. Это не так уж много, но как отправная точка для построения своей системы вполне годится. Отметим, что документированы роли не всех исполнителей. Набор данных неполон, но об этом можно будет подумать позже.

Фаза 1: трансформация данных

Вы, наверное, недоумеваете, почему нужно сначала загружать данные в Redis, а не сразу в CouchDB. Играя роль посредника, Redis наделяет плоские TSV-данные структурой, поэтому последующая загрузка в другую базу происходит быстрее. Мы планируем создать по одной записи для каждой группы, и Redis позволит сделать это за один проход по TSV-файлу (в котором одна группа упоминается столько раз, сколько в ней есть участников, — каждый участник представлен в отдельной строке). Если добавлять участников по одному в CouchDB, то возникнет пробуксовка при обновлении, поскольку при вставке двух участников одной группы мы будем пытаться одновременно создать или обновить один и тот же документ, в результате чего системе придется повторять вставку, когда одна из попыток завершится ошибкой, обнаруженной механизмом контроля версий, встроенным в CouchDB.

У этой стратегии есть один недостаток — ограничение, связанное с тем, что Redis хранит весь набор данных в памяти. Впрочем, эту проблему можно решить с помощью простого кластера с согласованным хешированием, с которым мы познакомились во второй день.

Скачав данные, убедитесь, что установлен Node.js и его менеджер пакетов Node Package Manager (npm). После этого установите три

пакета NPM: redis, csv и hiredis (необязательный написанный на C драйвер для Redis, который мы вчера рассматривали; он способен существенно ускорить взаимодействие в Redis).

```
$ npm install hiredis redis csv
```

Затем убедитесь, что сервер Redis прослушивает порт по умолчанию 6379, или измените функцию `createClient()` во всех скриптах, прописав в ней правильный порт. Для загрузки данных в Redis запустите показанный ниже скрипт Node.js из того же каталога, в котором находится TSV-файл (мы предполагаем, что он называется `group_membership.tsv`). Все написанные на JavaScript скрипты довольно объемисты, поэтому полностью мы их не приводим. Полный код можно скачать с сайта Pragmatic Bookshelf. Здесь же мы остановимся только на основных моментах. Скачайте и выполните следующий файл:

```
$ node pre_populate.js
```

Этот скрипт перебирает все строки TSV-файла, извлекая имя исполнителя, название группы и роли исполнителя в данной группе. Затем эти данные добавляются в базу Redis (пустые значения пропускаются).

В Redis ключ группы имеет вид `"band:Band Name"`. Скрипт добавляет имя исполнителя в множество. Таким образом, ключ `"band:Beatles"` будет ссылаться на множество значений `["John Lennon", "Paul McCartney", "George Harrison", "Ringo Starr"]`. Точно так же, ключи, соответствующие исполнителям, содержат название группы и множество ролей. Например, ключ `"artist:Beatles:Ringo Starr"` содержит множество `["Drums"]` (ударные).

Прочий код нужен просто для подсчета числа обработанных строк и вывода результатов на экран.

redis/pre_populate.js

```
csv().
fromPath( tsvFileName, { delimiter: '\t', quote: '' }).
on('data', function(data, index) {
  var
    artist = data[2],
    band = data[3],
    roles = buildRoles(data[4]);
  if( band === '' || artist === '' ) {
    trackLineCount();
    return true;
  }
})
```



```
redis_client.sadd('band:' + band, artist);
roles.forEach(function(role) {
  redis_client.sadd('artist:' + band + ':' + artist, role);
});
trackLineCount();
}).
```

Убедиться в том, что этот скрипт действительно загрузил данные в Redis, можно запустив программу `redis-cli` и выполнив команду `RANDOMKEY`. Мы ожидаем получить ключ, начинающийся с `band:` или `artist:`. Значение нас устраивает любое, кроме (`nil`).

Загрузив данные в Redis, сразу же переходите к следующей части. Если в этот момент остановить Redis, то данные, скорее всего, будут потеряны, если вы, конечно, не установили режим долговечности, отличный от умалчиваемого, или не выполнили команду `SAVE`.

Фаза 2: вставка в каноническую систему

CouchDB у нас играет роль канонической системы, в которую записываются данные. В любом конфликте между Redis, CouchDB или Neo4j выходит победителем CouchDB. Хорошая каноническая система должна хранить все данные, необходимые для перестройки любого источника данных на подведомственной «территории».

Проверьте, что CouchDB прослушивает порт по умолчанию 5984, или измените номер порта в строке `require('http').createClient(5984, 'localhost')` показанного ниже скрипта. Redis должен находиться в том состоянии, в котором мы оставили его в предыдущем разделе. Скачайте и выполните следующий файл.

```
$ node populate_couch.js
```

Если в фазе 1 мы читали TSV-файл и загружали данные из него в Redis, то теперь будем читать данные из Redis и загружать их в CouchDB. Нам не понадобятся специальные драйверы для CouchDB, потому что взаимодействие с ней происходит через простой REST-интерфейс, а в Node.js уже встроена библиотека для работы с протоколом HTTP.

Говорит Эрик: неблокирующий код

Начиная писать эту книгу, мы имели лишь поверхностное представление о том, как писать событийно-управляемые неблокирующие приложения. Слово «неблокирующий» означает следующее: вместо того, чтобы ждать завершения долго работающего процесса, главная программа продолжает работать. Всё, что необходимо сделать в ответ на блокирующее событие, по-

мещается внутрь функции или блока, который будет выполнен позже. Делать это можно путем запуска отдельного потока или – как в нашем случае – путем реализации паттерна реактора.

В блокирующей программе код, который запрашивает базу данных, ожидает ответа и обрабатывает результаты, пишется следующим образом:

```
results = database.some_query()
for value in results
  # обработать каждое значение
end
# этот код выполняется только после завершения цикла обработки результатов...
```

В событийно-управляемой программе цикл передается в виде функции или блока кода. Пока база данных занимается своим делом, основная часть программы продолжает работать. И лишь после того, как будет получен результат, переданная функция или блок выполняется.

```
database.some_query do |results|
  for value in results
    # обработать каждое значение
  end
end
# этот код продолжает работать, пока база данных выполняет запрос...
```

На то, чтобы осознать скрытые в этом подходе преимущества, у нас ушло некоторое время. Да, программа может продолжать работу вместо того, чтобы ожидать ответа от базы данных, но так ли часто этим можно воспользоваться? Очевидно, да, потому что, начав программировать в таком стиле, мы добились уменьшения задержек на несколько порядков.

Мы старались сделать код максимально простым, однако неблокирующая работа с базой данных – дело принципиально сложное. Впрочем, как мы поняли, в общем случае это очень эффективный метод обращения к базам. Почти во всех популярных языках программирования имеется та или иная неблокирующая библиотека. В Ruby есть EventMachine, в Python – Twisted, в Java – библиотека NIO, в C# – Interlace, ну а в JavaScript – Node.js.

В показанном ниже блоке мы выполняем команду Redis `KEYS bands:*`, чтобы получить список всех названий групп. Если бы набор данных был *по-настоящему* большим, то можно было бы ограничить запрос (например, опросить сначала ключ `bands:A*`, чтобы выбрать только названия, начинающиеся на *a*, и т. д.). Затем для каждой группы мы запрашиваем множество исполнителей и выделяем из ключа название группы, удаляя префикс *bands:*.

redis/populate_couch.js

```
redisClient.keys('band:*', function(error, bandKeys) {
  totalBands = bandKeys.length;
  var
    readBands = 0,
    bandsBatch = [];
  bandKeys.forEach(function(bandKey) {
```

```
// подстрока 'band:'.length содержит название группы
var bandName = bandKey.substring(5);
redisClient.smembers(bandKey, function(error, artists) {
```

Затем мы получаем роли каждого исполнителя, Redis возвращает их в виде массива массивов (роли каждого исполнителя в отдельном массиве). Для этого мы можем собрать команды `SMEMBERS` в одном массиве `roleBatch` и выполнить их пакетно в рамках одной команды `MULTI`. По существу, это один конвейерный запрос вида:

```
MULTI
  SMEMBERS "artist:Beatles:John Lennon"
  SMEMBERS "artist:Beatles:Ringo Starr"
EXEC
```

Далее мы создаем пакет из 50 документов CouchDB. Мы решили поступить именно так, потому что затем передаем весь пакет команде CouchDB / `_bulk_docs`, благодаря чему вставка производится очень-очень быстро.

redis/populate_couch.js

```
redisClient.
  multi(roleBatch).
  exec(function(err, roles)
  {
    var
      i = 0,
      artistDocs = [];

    // построить поддокументы для исполнителей
    artists.forEach( function(artistName) {
      artistDocs.push({ name: artistName, role : roles[i++] });
    });

    // добавить новый документ, описывающий группу, в пакет,
    // который будет выполнен позже
    bandsBatch.push({
      _id: couchKeyify( bandName ),
      name: bandName,
      artists: artistDocs
    });
```

Заполнив базу данных о музыкальных группах, мы теперь имеем место, где хранятся все необходимые системе данные. Мы знаем названия групп, имена участвующих в них музыкантов и их роли в группе. Можно сделать перерыв и поиграться с только что заполненной канонической базой данных CouchDB, отправив запрос по адресу `http://localhost:5984/_utils/database.html?bands`.

Хранилище связей

Следующей в нашем списке значится служба на основе Neo4j, которую мы планируем использовать для отслеживания связей между исполнителями и их ролями. Конечно, можно было бы опрашивать CouchDB напрямую, создав представления, но сформулировать сложные запросы, касающиеся связей, так не удастся. Если Уэйн Койн из Flaming Lips потеряет перед концертом свой терменвокс, то он сможет одолжить его у Чарли Клоузера из Nine Inch Nails, который тоже играет на терменвоксе. Точно так же можно было бы найти музыкантов, обладающих одновременно несколькими талантами, даже если в разных группах они выступают в разных ампула, – для этого достаточно простого обхода узлов.

Подготовив исходные данные, мы теперь должны позаботиться о синхронизации Neo4j с CouchDB в случае, когда данные в канонической системе изменяются. Мы собираемся убить одним выстрелом двух зайцев, написав службу, которая будет передавать в Neo4j все изменения, происходящие в CouchDB с момента создания базы.

Мы также хотим занести в Redis ключи всех групп, исполнителей и ролей, чтобы можно было быстро находить эти данные впоследствии. По счастью, это как раз те данные, которые мы уже поместили в CouchDB, что избавляет нас от отдельного шага для начального заполнения Neo4j и Redis.

Проверьте, что Neo4j прослушивает порт 7474, или измените номер порта в функции `createClient()`. CouchDB и Redis должны быть уже запущены. Скачайте и выполните нижеупомянутый файл. Этот скрипт будет работать, пока вы не прервете его явно.

```
$ node graph_sync.js
```

Это сервер, который непрерывно опрашивает CouchDB на предмет произошедших изменений (как это делается, мы видели в главе, посвященной CouchDB). Обнаружив изменение, мы делаем две вещи: копируем данные в Redis и в Neo4j. Следующий код заполняет Redis, выполняя каскад функций обратного вызова. Сначала копируется группа с ключом `"band-name:Band Name"`. И далее точно так же копируется имя исполнителя и его роли.

Это позволит нам искать по неполным строкам. Например, команда `KEYS band-name:Bea*` вернет следующие группы: Beach Boys, Beastie Boys, Beatles и т. д.

redis/graph_sync.js

```
function feedBandToRedis(band) {
  redisClient.set('band-name:' + band.name, 1);
  band.artists.forEach(function(artist) {
    redisClient.set('artist-name:' + artist.name, 1);
    artist.role.forEach(function(role) {
      redisClient.set('role-name:' + role, 1);
    });
  });
}
```

В следующем блоке мы заполняем Neo4j. В файле `neo4j_caching_client.js`, который можно скачать с сайта этой книги, находится соответствующий драйвер. Он пользуется встроенной в Node.js библиотекой для работы с HTTP, чтобы подключиться к REST-интерфейсу Neo4j; дополнительно мы ограничили количество одновременно открываемых клиентом соединений. Наш драйвер также использует Redis, чтобы отслеживать изменения, произведенные в графе Neo4j, не отправляя отдельного запроса. Это третье применение Redis в нашей системе; первое – трансформация данных для заполнения базы CouchDB, второе – быстрый поиск по названиям групп.

В этом коде создаются узлы групп (если они еще не созданы), затем узлы исполнителей (если необходимо), а затем роли. На каждом шаге попутно создаются новые связи, так что узел The Beatles связывается с узлами Джона, Пола, Джорджа и Ринго, которые в свою очередь связаны с узлами ролей.

redis/graph_sync.js

```
function feedBandToNeo4j(band, progress) {
  var
    lookup = neo4jClient.lookupOrCreateNode,
    relate = neo4jClient.createRelationship;

  lookup('bands', 'name', band.name, function(bandNode) {
    progress.emit('progress', 'band');
    band.artists.forEach(function(artist) {
      lookup('artists', 'name', artist.name, function(artistNode) {
        progress.emit('progress', 'artist');
        relate(bandNode.self, artistNode.self, 'member', function() {
          progress.emit('progress', 'member');
        });
      });
      artist.role.forEach(function(role) {
        lookup('roles', 'role', role, function(roleNode) {
          progress.emit('progress', 'role');
          relate(artistNode.self, roleNode.self, 'plays', function() {
            progress.emit('progress', 'plays');
          });
        });
      });
    });
  });
}
```

Запустите эту службу в отдельном окне. Всякое изменение в CouchDB, при котором добавляется новый исполнитель или новая

роль существующего исполнителя, приводит к созданию новой связи в Neo4j и, возможно, новых ключей в Redis. Пока эта служба работает, все три базы остаются синхронизированными.

Зайдите в веб-консоль CouchDB и откройте группу. Внесите в базу данных какое-нибудь изменение, например, добавьте в группу нового участника (сделайте самого себя участником «Битлз»!) или назначьте новую роль исполнителю. Следите за тем, что выводит скрипт `graph_sync`. Затем откройте консоль Neo4j и попробуйте найти новые связи в графе. Если вы добавили нового участника группы, то должна появиться связь между его узлом и узлом группы. В текущей реализации связи не удаляются, но добавление операции Neo4j `DELETE` не потребует полной переделки скрипта.

Веб-служба

Это как раз то, ради чего всё затевалось. Мы собираемся написать простое веб-приложение, которое позволит пользователям искать музыкальные группы. Для любой группы будут в виде ссылок перечислены ее участники, а щелчок по ссылке на участника выведет сведения о нем, точнее, все его амплуа в группе. Кроме того, для каждой роли исполнителя будут выведены все прочие имеющиеся в системе исполнители, играющие такую же роль в своей группе.

Например, поиск группы Led Zeppelin вернет Джимми Пейджа (Jimmy Page), Джона Пола Джонса (John Paul Jones), Джона Бонэма (John Bonham) и Роберта Планта (Robert Plant). Щелкнув по ссылке на Джимми Пейджа, мы узнаем, что он играет на гитаре, а заодно получим кучу других гитаристов, например The Edge (Эдж) из группы U2.

Чтобы немного упростить создание веб-приложения, установим еще два пакета: `bricks` (простой веб-каркас) и `mustache` (библиотека для работы с шаблонами).

```
$ npm install bricks mustache
```

Убедитесь, что все базы данных работают, после чего запустите веб-сервер. Скачайте и выполните следующий файл:

```
$ node band.js
```

Сервер прослушивает порт 8080, так что если вы зайдете в браузере на адрес `http://localhost:8080/`, то увидите простую поисковую форму.

Ниже приведен код построения веб-страницы с информацией о группе. Каждый URL выполняет в нашем крохотном HTTP-сервере

свою роль. URL `http://localhost:8080/band` принимает в качестве параметра имя группы.

redis/bands.js

```
appServer.addRoute("/band$", function(req, res) {
  var
    bandName = req.param('name'),
    bandNodePath = '/bands/' + couchUtil.couchKeyify( bandName ),
    membersQuery = 'g.V[[name:"'+bandName+'"]]'
      + '.out("member").in("member").uniqueObject.name';
  getCouchDoc( bandNodePath, res, function( couchObj ) {
    gremlin( membersQuery, function(graphData) {
      var artists = couchObj && couchObj['artists'];
      var values = { band: bandName, artists: artists, bands: graphData };
      var body = '<h2>{{band}} Band Members</h2>';
      body += '<ul>{{#artists}}';
      body += '<li><a href="/artist?name={{name}}">{{name}}</a></li>';
      body += '{{{/artists}}}</ul>';
      body += '<h3>You may also like</h3>';
      body += '<ul>{{#bands}}';
      body += '<li><a href="/band?name={{.}}">{{.}}</a></li>';
      body += '{{{/bands}}}</ul>';
      writeTemplate( res, body, values );
    });
  });
});
```

Если вы введете в форме название группы *Nirvana*, то серверу будет отправлен запрос на адрес `http://localhost:8080/band?name=Nirvana`. Показанная выше функция отрисует HTML-страницу (ее шаблон находится во внешнем файле `template.html`). На этой странице будут выведены данные обо всех входящих в группу исполнителях, взятые непосредственно из CouchDB. Кроме того, показан список рекомендуемых групп; для его получения мы выполним Gremlin-запрос к графу Neo4j. Для группы Nirvana этот запрос выглядит следующим образом:

```
g.V.filter{it.name=="Nirvana"}.out("member").in("member").dedup.name
```

Иными словами, выходя из узла Nirvana, мы получаем уникальные названия групп, участники которых связаны с участниками группы Nirvana. Например, Дэйв Грол (Dave Grohl) играл как в Nirvana, так и в Foo Fighters, поэтому группа Foo Fighters будет включена в список.

На следующей странице – `http://localhost:8080/artist` – выводятся сведения об исполнителе.

redis/bands.js

```
appServer.addRoute("/artist$", function(req, res) {
  var
    artistName = req.param('name'),
```

```
rolesQuery = 'g.V[[name:"'+artistName+'']].out("plays").role. \
  uniqueObject',
bandsQuery = 'g.V[[name:"'+artistName+'']].in("member").name. \
  uniqueObject';
gremlin( rolesQuery, function(roles) {
  gremlin( bandsQuery, function(bands) {
    var values = { artist: artistName, roles: roles, bands: bands };
    var body = '<h3>{{artist}} Performs these Roles</h3>';
    body += '<ul>{{#roles}}';
    body += '<li>{{.}}</li>';
    body += '{{{/roles}}}</ul>';
    body += '<h3>Play in Bands</h3>';
    body += '<ul>{{#bands}}';
    body += '<li><a href="/band?name={{.}}">{{.}}</a></li>';
    body += '{{{/bands}}}</ul>';
    writeTemplate( res, body, values );
  });
});
```

Здесь мы выполняем два Gremlin-запроса. Первый выводит все роли исполнителя, второй – список групп, в которых он участвовал. Например, для Джеффа Уорда (Jeff Ward) (<http://localhost:8080/artist?name=Jeff%20Ward>) будет показано, что он играет на ударных (Drummer) и участвовал в группах Nine Inch Nails и Ministry.

Обратите внимание, что на двух описанных выше страницах мы выводим перекрестные ссылки. Ссылки в списке исполнителей на странице /bands ведут на страницу /artist выбранного исполнителя и наоборот. Но можно немного упростить поиск.

redis/bands.js

```
appServer.addRoute("^/search$", function(req, res) {
  var query = req.param('term');
  redisClient.keys("band-name:"+query+"*", function(error, keys) {
    var bands = [];
    keys.forEach(function(key) {
      bands.push(key.replace("band-name:", ''));
    });
    res.write( JSON.stringify(bands) );
    res.end();
  });
});
```

Здесь мы запрашиваем у Redis все ключи, соответствующие начальной части строки, как, например, "Bea*" выше. Redis выводит данные в формате JSON. Шаблон `template.html` включает код на jQuery, который наделяет поисковое поле функцией автозавершения.

Развитие веб-службы

Этот сравнительно небольшой скрипт реализует только самые необходимые функции. Его можно развить во многих направлениях. Например, в состав рекомендуемых входят только группы первого

порядка (те, в которых участвовали музыканты, входящие в текущую группу). Но можно получить интересные результаты, написав запрос для получения групп второго порядка, например: `g.V.filter{it.name=='Nine Inch Nails'}.out('member').in('member').dedup.loop(3){ it.loops <= 2 }.name`.

Кроме того, у нас нет формы, позволяющей изменить информацию о группе. Добавить такую функциональность не слишком сложно, потому что мы уже написали код заполнения CouchDB в скрипте `populate_couch.js`, а любое изменение в CouchDB автоматически отражается в Neo4j и Redis, если работает служба `graph_sync.js`.

Если забавы с многосторонним хранением вам понравились, то можно пойти еще дальше. Например, добавить хранилище данных на основе PostgreSQL⁷ для преобразования этих данных в схему типа «звезда», позволяющую анализировать данные по разным измерениям, например: самый распространенный музыкальный инструмент или сравнение среднего числа участников группы со средним числом инструментов. Можно также добавить сервер Riak для хранения образчиков произведений группы или сервер HBase для построения системы обмена сообщениями, с помощью которой пользователи могут отслеживать историю своих симпатий и антипатий, или расширение MongoDB, привносящее в систему географический аспект.

Или можете вообще переписать проект на другом языке, с использованием другого веб-каркаса или набора данных. Возможностей развития столько, сколько существует комбинаций баз данных и технологий их создания, то есть декартово произведение всех проектов с открытым исходным кодом.

День 3: итоги

Сегодня у нас был трудный день – настолько трудный, что мы не удивимся, если на самом деле он растянулся на несколько дней. Зато вы почувствовали, как выглядит будущее систем управления данными в мире, который движется в направлении от *одной большой реляционной СУБД* к модели с *несколькими специализированными базами данных*. Мы объединили эти базы с помощью неблокирующего кода, и, хотя эта технология не является предметом данной книги, стоит отметить, что именно в эту сторону смещаются предпочтения разработчиков в части взаимодействия с базами данных.

Важность Redis в этой модели не следует недооценивать. Безусловно, Redis не обладает никакой функциональностью, которой не

⁷ http://en.wikipedia.org/wiki/Data_warehouse

могла бы предоставить любая из рассмотренных баз данных по отдельности, зато в ней есть быстрые структуры данных. Мы смогли преобразовать плоский файл в ряд осмысленных структур данных, что необходимо как при заполнении системы данными, так и при их передаче. Причем реализовать это удалось легко и быстро.

Даже если в целом модель многостороннего хранения не вызвала у вас интереса, при разработке любой системы есть смысл иметь в виду Redis.

День 3: домашнее задание

Задачи

1. Измените программу импорта, так чтобы в состав данных о группе включались еще даты прихода и ухода каждого участника. Сохраните эти сведения в поддокументе исполнителя в CouchDB и отображайте на странице исполнителя.
2. Добавьте в системе еще и базу данных Riak для хранения нескольких образчиков творчества группы в GridFS, так чтобы пользователь мог прослушать одну-две композиции. Если для группы хранится хотя бы одна композиция, включите ссылку на нее в веб-приложение. Обеспечьте синхронизацию между Riak и CouchDB.

8.5. Резюме

Redis – компактное хранилище ключей и значений (или структур данных), потребляющее немного ресурсов. Оно сродни многофункциональным инструментам, включающим нож, открывалку, штопор и другие приспособления. Как и они, Redis пригодна для решения самых разных, часто неожиданных, задач. Ко всему прочему, хранилище Redis работает быстро, просто в эксплуатации и обеспечивает такую долговечность данных, какую вы настроите. Redis редко используется как автономная база данных, чаще она входит в состав многосторонней экосистемы, где применяется для трансформации данных, кэширования запросов или управления очередями сообщений (благодаря встроенным блокирующим командам).

Сильные стороны Redis

Очевидное достоинство Redis – быстрое действие, чем могут похвастаться многие подобные хранилища ключей и значений. Но, в отличие

от большинства таких хранилищ, Redis еще умеет хранить составные значения – списки, хеши и множества – и применять к ним специальные операции. Мало того, в Redis еще можно тонко настраивать долговечность данных, отдавая предпочтение либо надежности хранения, либо скорости. Встроенная репликация типа главный-подчиненный – еще один удобный способ повысить долговечность, не принося быстроедействие в жертву медленной дозаписи в файл при каждой операции. К тому же, репликация повышает производительность систем, в которых основная нагрузка приходится на операции чтения.

Слабые стороны Redis

Своим быстродействием Redis обязана, прежде всего, тому, что хранит все данные в памяти. Кто-то назовет это обманом, потому что база данных, которая вообще не обращается к диску, естественно, будет работать быстро. Любому хранилищу в оперативной памяти присуща проблема долговечности данных; если остановить базу, не сбросив данные на диск, то данные будут потеряны. Даже если установить синхронный режим дозаписи в файл при каждой операции, все равно при воспроизведении файла восстановление данных с истекшим сроком хранения не вполне надежно – это характерно для любого механизма воспроизведения событий, привязанных ко времени. Впрочем, по совести говоря, эта проблема скорее теоретическая, нежели практическая.

Redis также не поддерживает наборы данных, размер которых превышает объем доступной оперативной памяти (поддержка виртуальной памяти из Redis будет исключена). В настоящее время разрабатывается кластер Redis, который позволит выйти за пределы ограничения на объем ОЗУ в одном компьютере, но пока пользователи, желающие организовать кластер, должны сами написать соответствующий клиент (как драйвер на Ruby, с которым мы работали во второй день).

Перед расставанием

Redis не испытывает недостатка в командах – всего их более 120. Названия большинства команд объясняют их назначение, нужно лишь привыкнуть к мысли, что некоторые избыточные буквы опущены (как, например, в команде `INCRBY`) и что математическая точность иногда не столько помогает, сколько запутывает (сравните, например, `ZCOUNT` – число элементов в отсортированном множестве и `SCARD` – кардинальное число множества).

Redis уже стал неотъемлемой частью многих систем. На его основе создано несколько проектов с открытым исходным кодом – от Resque (написанная на Ruby служба асинхронных очередей задач) до SocketStream (система управления сеансами для Node.js). Вне зависимости от того, какая база данных выбрана в качестве канонической системы, подумать об использовании совместно с ней Redis несомненно имеет смысл.



ГЛАВА 9.

Подводя итоги

Проработав материал обо всех семи базах данных, вы, безусловно, заслужили поздравления!

Надеемся, что вы оценили достоинства этих систем. Если вы решите воспользоваться какой-нибудь из них в своем проекте, мы будем рады. А если решите применить сразу несколько, как в конце главы о Redis, наша радость достигнет степеней восторга. Мы полагаем, что будущее систем управления данными принадлежит модели многостороннего хранения (когда в одном проекте используется сразу несколько баз данных), а монопольное преобладание универсальных РСУБД сойдет на нет.

Воспользуемся возможностью и посмотрим, как наши семь баз данных укладываются в более крупную экосистему. Мы уже изучили детали каждой базы и отметили некоторые сходства и различия. Теперь поговорим об их вкладе в широкую и постоянно расширяющуюся панораму систем хранения данных.

9.1. Снова о жанрах

Мы видели, что по способу хранения базы данных можно приблизительно разбить на пять жанров: реляционные, хранилища ключей и значений, столбцовые, документные и графовые. Ненадолго задержимся и вспомним, чем они отличаются друг от друга и для каких задач подходят или не подходят – когда их имеет смысл использовать, а когда нужно держаться подальше.

Реляционные базы данных

Это классический и самый распространенный вариант. Реляционные системы управления базами данных (РСУБД) основаны на теории множеств, данные в них хранятся в виде двумерных таблиц, состо-

ящих из строк и столбцов. Реляционные базы строго контролируют типы данных и обычно допускают хранение чисел, строк, дат и неинтерпретируемых двоичных объектов (BLOB'ов), хотя, как мы видели, PostgreSQL поддерживает такие расширения, как массив и куб.

Хороши для

Благодаря структурированной природе данных реляционные базы имеет смысл использовать, когда структура данных заранее известна, а способы их возможного использования – нет. Иными словами, вы готовы авансом заплатить за сложность организации, надеясь, что в будущем это окупится гибкостью запросов. Многие задачи бизнеса удобно моделируются подобным образом: заказы, поставки, учет складских запасов, корзины покупок и другие. Заранее неизвестно, как впоследствии потребуются извлекать данные (например, сколько заказов мы обработали в феврале?), но данные по природе своей регулярны, и контролировать эту регулярность весьма полезно.

Не очень хороши для

Если структура данных переменна или характеризуется глубокой вложенностью, то реляционная СУБД – не лучший помощник. В ней необходимо заранее определить схему, что очень трудно сделать, если все записи структурно отличаются друг от друга. Подумайте, как бы вы стали разрабатывать базу данных для описания всех живых существ в природе. Попытка создать полный список признаков («имеет волосы», «количество ног», «откладывает яйца» и т. д.) обречена на провал. В таком случае нужна база данных, которая не налагает таких строгих ограничений на хранимые данные.

Хранилища ключей и значений

Хранилище ключей и значений – простейшая из всех рассмотренных нами моделей. В нем простые ключи отображаются на, возможно, более сложные значения, как в гигантской хеш-таблице. Благодаря относительной простоте базы данных этого жанра обеспечивают максимальную гибкость реализации. Поиск в хеш-таблице производится быстро, поэтому в случае Redis быстроедействие было поставлено во главу угла. Кроме того, поиск в хеш-таблицах легко распределяется на несколько машин, и в Riak этот факт использован для построения простых в управлении кластеров. Разумеется, простота может оказаться минусом, если к моделированию данных предъявляются сложные требования.

Хороши для

Поскольку необходимость поддерживать индексы отсутствует или невелика, то при проектировании хранилищ ключей и значений часто закладывается горизонтальная масштабируемость, очень высокое быстродействие или то и другое вместе. Особенно хороши они для задач, где между данными нет большого количества связей. Например, в веб-приложении этому критерию отвечают сеансы пользователей; действия одного пользователя в сеансе практически никак не связаны с действиями других пользователей.

Не очень хороши для

Из-за отсутствия индексов и средств сканирования КЗ-хранилища мало чем помогут, если требуется предъявлять запросы, отличные от простейших операций CRUD (создание, чтение, обновление, удаление).

Столбцовые базы данных

Столбцовые базы данных (также *базы данных с семействами столбцов*) имеют много общих черт с КЗ-хранилищами и РСУБД. Как и в КЗ-хранилищах, значения запрашиваются по ключу. Как и в реляционных базах, значения группируются в нуль или более столбцов, хотя в каждой строке может быть заполнено произвольное число столбцов. Но в отличие от того и другого, в столбцовых базах данные хранятся по столбцам, а не по строкам. Добавление новых столбцов обходится дешево, версионирование реализуется тривиально, и на хранение отсутствующих значений место не расходуется. Рассмотренная нами база данных HBase – классический пример этого жанра.

Хороши для

Столбцовые базы данных традиционно разрабатывались с целью обеспечить горизонтальную масштабируемость. Поэтому они особенно хороши для задач, связанных с «большими данными», когда база размещается в кластере из десятков, сотен, а то и тысяч узлов. Кроме того, в них обычно встраивается поддержка сжатия и версионирования. Канонический пример задачи, ориентированной на хранение данных по столбцам, – индексирование веб-страниц. Страницы в веб содержат много текста (и тут очень полезно сжатие), в какой-то мере взаимосвязаны и изменяются со временем (вот где на помощь приходит версионирование).

Не очень хороши для

У разных столбцовых баз данных разные возможности и, следовательно, разные недостатки. Но всем им свойственна одна общая черта: лучше проектировать схему с учетом будущих запросов. Это означает, что нужно априори представлять себе, как будут использоваться данные, а не только, что они содержат. Если способы использования данных заранее неизвестны – например, требуются произвольные отчеты, – то столбцовая база данных – не оптимальный вариант.

Документные базы данных

В документной базе данных хранимый объект может содержать произвольный набор полей и даже допускает вложенность любого уровня. Обычно такие объекты представляются в нотации JavaScript Object Notation (JSON), принятой как в MongoDB, так и в CouchDB, хотя это требование ни в коей мере не является концептуально значимым. Поскольку документы не связаны друг с другом так тесно, как в реляционных базах данных, их сравнительно просто сегментировать и реплицировать на несколько серверов, поэтому часто встречаются распределенные реализации. Система MongoHQ ставит целью повысить доступность за счет создания ЦОД, способных поддерживать гигантские наборы данных масштаба веб. С другой стороны, CouchDB ориентирована на простоту и долговечность данных, а доступность достигается за счет репликации типа главный-главный на узлы с высоким уровнем автономности. Эти проекты в значительной мере перекрываются.

Хороши для

Документные базы данных хороши для задач, где предметная область характеризуется высокой степенью изменчивости. Если заранее неизвестно, как выглядят данные, то документная база – как раз то, что нужно. Кроме того, документы по самой своей природе часто хорошо ложатся на объектно-ориентированные модели программирования. А, значит, уменьшается рассогласование интерфейсов при переходе от модели базы данных к объектной модели приложения.

Не очень хороши для

Если вы привыкли к сложным запросам с соединениями в реляционной базе данных с хорошо нормализованной схемой, то в документной базе этих возможностей будет не хватать. Документ обычно содержит всю или большую часть информации об объекте. Если в ре-

ляционной базе естественно стремление нормализовать данные с целью устранения дублирования, которое может привести к рассогласованию, то в документных базах денормализованное хранение – это норма.

Графовые базы данных

Графовые базы данных – новый, быстро развивающийся класс систем, в которых упор делается скорее на установление произвольных связей между данными, чем на фактические значения. Примером может служить проект с открытым исходным кодом Neo4j, который все чаще используется во многих социальных сетях. В отличие от других стилей баз данных, где коллекции схожих объектов группируются в общих контейнерах, в графовых базах данные хранятся в более свободной форме; обработка запроса сводится к следованию по ребрам, соединяющим две вершины, иначе говоря, к *обходу* узлов. По мере использования во все большем числе проектов графовые базы данных начинают применяться не только для построения простых социальных приложений, но и для более специфических целей, например: системы рекомендаций, списки управления доступом и географические данные.

Хороши для

Графовые базы данных как будто специально придуманы для приложений, характеризующихся наличием сети данных. Типичный пример – социальная сеть, где узлы представляют пользователей, между которыми существуют различные связи. Моделирование таких данных с помощью других средств зачастую является сложной проблемой, но графовые базы данных справляются с этой задачей на ура. К тому же, они прекрасно ложатся на объектно-ориентированную систему. Все, что можно смоделировать на доске, можно смоделировать и с помощью графа.

Не очень хороши для

Из-за наличия большого числа связей между узлами графовые базы данных как правило плохо приспособлены к размещению на нескольких машинах. Быстрый обход графа был бы невозможен, если бы пришлось отправлять по сети запросы другим узлам, поэтому графовые базы данных масштабируются плохо. Скорее всего, графовая база будет лишь одной из частей вашей системы, в которой хранятся только связи, тогда как основные данные находятся в другом месте.

9.2. Как сделать выбор?

В начале книги мы уже говорили, что данные – все равно, что сырая нефть. Мы сидим на берегу безбрежного океана данных, но они бесполезны, пока из них не извлечена информация (более грубо можно сказать, что сегодня в данных зарыты огромные деньги). От того, какую базу данных вы выберете, зависит простота сбора, хранения, анализа и очистки данных.

Определиться с выбором базы данных нередко оказывается сложнее, чем просто решить какой жанр лучше всего подходит для данных из конкретной предметной области. Очевидно, что социальный граф проще всего хранить в графовой базе данных, но если речь идет о Facebook, то данных слишком много – в графовой базе они попросту не поместятся. Скорее, вы остановитесь на какой-нибудь системе, приспособленной к «большим данным», например, HBase или Riak. Таким образом, вы будете вынуждены выбрать столбцовую базу или хранилище ключей и значений. Или взять другой пример: реляционная база данных вроде бы идеально подходит для реализации банковских транзакций, но ведь и Neo4j поддерживает ACID-транзакции, так что выбор есть.

Эти примеры показывают, что при выборе базы данных – или нескольких баз данных, – оптимальных для предметной области, следует принимать во внимание не только жанр. Вообще говоря, по мере роста объема данных, предельная емкость некоторых баз становится ограничением. Столбцовые базы данных часто распределяются по нескольким ЦОД и поддерживают «большие данные» максимального размера, тогда как емкость графовых баз наименьшая из всех. Впрочем, это правило не универсально. Так, Riak – крупномасштабное хранилище ключей и значений, предназначенное для сегментирования на сотни и тысячи узлов, тогда как Redis проектировалась для работы в одном узле, но с возможностью репликации типа главный-подчиненный и сегментирования, реализованного на уровне клиента.

При выборе базы данных нужно учитывать и такие характеристики, как долговечность, доступность, согласованность, масштабируемость и безопасность. Вы должны решить, нужна ли возможность предъявлять произвольные запросы или будет достаточно технологии mapreduce. Вы предпочитаете HTTP/REST-интерфейс или нуждаетесь в драйвере для специализированного двоичного протокола? Даже такие, на первый взгляд, второстепенные вопросы, как существование программы массовой загрузки данных, могут оказаться важными в конкретном случае.

Чтобы облегчить сравнение различных баз данных, мы включили в приложение 1 таблицу. Мы не ставили целью составить полный список всех возможностей, а хотели лишь предложить средство для быстрого сравнения тех баз данных, которые рассмотрены в этой книге. Обращайте внимание на номера версий. Функциональность меняется очень быстро, поэтому мы рекомендуем проверять, что появилось в более свежих версиях.

9.3. В каком направлении двигаться дальше?

В современных приложениях проблема масштабируемости сместилась преимущественно в область управления данными. Мы достигли точки, когда выбор языка программирования, каркаса, операционной системы и даже оборудования и порядка эксплуатации (благодаря хостингу виртуальных машин и облаку) настолько упростился и удешевился, что практически перестал быть проблемой и определяется личными вкусами в той же мере, что и необходимостью. Если вы хотите в этом столетии создавать масштабируемые приложения, то должны задуматься прежде всего о выборе базы (или баз) данных – именно здесь теперь находится узкое место. Помочь вам сделать правильный выбор – именно в этом мы видели главную цель этой книги.

Хотя эта книга подошла к концу, мы верим, что ваш интерес к многостороннему хранению далеко не удовлетворен. Следующим шагом может стать детальное изучение баз данных, которые вас особенно заинтересовали, или знакомство с другими системами, например, Cassandra, Drizzle или OrientDB.

Пора заняться практическим делом!



ПРИЛОЖЕНИЕ 1.

Сравнительный обзор баз данных

В этой книге приведено много информации о каждой из семи рассмотренных баз данных: PostgreSQL, Riak, HBase, MongoDB, CouchDB, Neo4j и Redis. В приложении вы найдете таблицы, в которых эти базы сравниваются по разным параметрам, суммируя изложенный в основном тексте материал. Хотя таблицы не могут заменить истинного понимания, они всё же позволяют с одного взгляда оценить возможности каждой базы, области, в которых она недотягивает, и ее место в общей картине современного мира баз данных.

	Жанр	Версия	Типы данных	Связи между данными
MongoDB	Документная	2.0	Типизированная	Нет
CouchDB	Документная	1.1	Типизированная	Нет
Riak	Ключи и значения	1.0	Blob-объекты	Произвольные (ссылки)
Redis	Ключи и значения	2.4	Полутипиризованная	Нет
PostgreSQL	Реляционная	9.1	Предопределенные и пользовательские	Предопределенные
Neo4j	Графовая	1.7	Без типов	Произвольные (ребра)
HBase	Столбцовая	0.90.3	Предопределенные и пользовательские	Нет

	Стандарт- ный объект	Написана на	Интерфейс/ протокол	HTTP/REST
MongoDB	JSON	C++	Специальный поверх TCP	Простой
CouchDB	JSON	Erlang	HTTP	Да
Riak	Текст	Erlang	HTTP, protobuf	Да
Redis	Строка	C/C++	Простой текст поверх TCP	Нет
PostgreSQL	Таблица	C	Специальный поверх TCP	Нет
Neo4j	Хеш	Java	HTTP	Да
HBase	Столбцы	Java	Thrift, HTTP	Да

	Произволь- ные запросы	Mapreduce	Масштабируе- мость	Долговеч- ность
MongoDB	Команды, mapreduce	JavaScript	ЦОД	Упреждающая запись, журналирование, безопасный режим
CouchDB	Временные представления	JavaScript	ЦОД (с помощью BigCouch)	Только защита от сбоев
Riak	Слабая поддержка, Lucene	JavaScript, Erlang	ЦОД	Кворум при записи
Redis	Команды	Нет	Кластер (с помощью репликации главный-подчиненный)	Файл дозаписи
PostgreSQL	SQL	Нет	Кластер (с помощью дополнительных расширений)	ACID
Neo4j	Обход графа, Cypher, поиск	Нет (в смысле распределенности)	Кластер (в корпоративной версии)	ACID
HBase	Слабая поддержка	Hadoop	ЦОД	Журнал с упреждающей записью

	Вторичные индексы	Версионирование	Массовая загрузка	Очень большие файлы
MongoDB	Да	Нет	mongoimport	GridFS
CouchDB	Да	Да	Bulk Doc API	Вложения
Riak	Да	Да	Нет	Lewak (не рекомендуется)
Redis	Нет	Нет	Нет	Нет
PostgreSQL	Да	Нет	Команда COPY	BLOB-объекты
Neo4j	Да (с помощью Lucene)	Нет	Нет	Нет
HBase	Нет	Да	Нет	Нет

	Обязательное сжатие	Репликация	Сегментирование	Параллелизм
MongoDB	Нет	Главный-подчиненный (с помощью наборов реплик)	Да	Блокировка записи
CouchDB	Перезапись файла	Главный-главный	Да (с фильтрами в BigCouch)	Без блокировок MVCC (многоверсионное управление параллелизмом)
Riak	Нет	Одноуровневая, главный-главный	Да	Векторные часы
Redis	Мгновенный снимок	Главный-подчиненный	Дополнительные модули (например, клиент)	Нет
PostgreSQL	Нет	Главный-подчиненный	Дополнительные модули (например, PL/Proxy)	Блокировка записи на уровне таблиц и строк
Neo4j	Нет	Главный-подчиненный (в издании Enterprise)	Нет	Блокировка записи
HBase	Нет	Главный-подчиненный	Да с помощью HDFS	Согласованность на уровне строк

	Транзакции	Триггеры	Безопасность	Множественная аренда
MongoDB	Нет	Нет	Пользователи	Да
CouchDB	Нет	Контроль обновлений или Changes API	Пользователи	Да
Riak	Нет	До и после фиксации	Нет	Нет
Redis	Очереди с несколькими запросами	Нет	Пароли	Нет
PostgreSQL	ACID	Да	Пользователи и группы	Да
Neo4j	ACID	Обработчики событий транзакции	Нет	Нет
HBase	Да (если включены)	Нет	Kerberos посредством встроенного в Hadoop механизма защиты	Нет

	Основная отличительная особенность	Слабые стороны
MongoDB	Простые запросы к «большим данным»	Встраиваемость
CouchDB	Долговечность и встраиваемые кластеры	Поддержка запросов
Riak	Высокая доступность	Поддержка запросов
Redis	Высочайшее быстродействие	Сложная структура данных
PostgreSQL	Лучшая РСУБД для поддержки операций	Доступность в распределенной системе
Neo4j	Гибкий граф	Отсутствие BLOB-объектов и поддержки терабайтных данных
HBase	Высокий уровень масштабируемости, инфраструктура Hadoop	Гибкий рост, поддержка запросов



ПРИЛОЖЕНИЕ 2.

Теорема CAP

Понимать особенности пяти жанров баз данных важно для принятия решения о выборе, но жанр – не единственный критерий. В этой книге неоднократно упоминалась теорема CAP, которая раскрывает неприглядную правду о том, как распределенные системы ведут себя в условиях нестабильной сети.

CAP утверждает, что можно создать распределенную систему, которая будет *согласованной* (consistent) (то есть операции записи атомарны, и все последующие операции чтения видят новое значение), *доступной* (available) (база данных возвращает значения, пока работает хотя бы один сервер) и *устойчивой к потере связности* (partition tolerant) (система продолжает функционировать, даже если связи между серверами временно отсутствует), но одновременно можно гарантировать только два из этих трех свойств.

Иными словами, можно создать распределенную систему, которая будет *согласованной* и *устойчивой к потере связности*, систему, которая будет *доступной* и *устойчивой к потере связности*, или систему, которая будет *согласованной* и *доступной* (но при это не будет *устойчивой к потере связности*, то есть, по существу, не будет распределенной). Но невозможно создать распределенную базу данных, которая была бы одновременно согласованной, доступной и устойчивой к потере связности.

Теорема CAP существенна при проектировании распределенной базы данных, так как вы должны решить, чем готовы пожертвовать. Какую бы базу данных вы ни выбрали, она не сможет гарантировать либо доступность, либо согласованность. Устойчивость к потере связности – чисто архитектурное решение (от него зависит, будет ли система вообще распределенной). Важно понимать смысл теоремы CAP, чтобы реалистично оценивать возможности. Компромиссы, принятые в различных описанных в книге базах данных, опираются именно на эту теорему.

A2.1. Согласованность в конечном счете

Распределенная база данных обязана быть устойчивой к потере связности, а вот выбор между доступностью и согласованностью иногда сделать трудно. Однако, хотя теорема CAP и утверждает, что, выбрав доступность, вы теряете истинную согласованность, *согласованность в конечном счете* обеспечить все-таки можно.

Приключения CAP, часть I: капитан¹

Представьте себе мир, как гигантскую распределенную базу данных. Любая обитаемая часть суши в этом мире содержит информацию по каким-то темам, и при условии, что вы находитесь рядом с людьми или техническим устройством, вы можете найти ответы на свои вопросы.

А теперь в рамках того же предположения допустим, что вы страстный поклонник Бейонсе Ноулз и что сегодня 5 сентября 2006 года. Вы находитесь в пляжном домике своего приятеля на вечеринке в честь выхода второго студийного альбома Бейонсе, как вдруг через мол прокатывается неожиданная приливная волна и уносит вас в открытое море. Вам удастся смастерить из подручных материалов плотик, на котором спустя много дней вы добираетесь до пустынного островка. Не имея никаких средств коммуникации, вы по существу *потеряли связь* с остальной системой (миром). На острове вы провели долгих пять лет... И вот в одно прекрасное утро 2011 года вас будят доносящиеся с моря крики. Вас нашел капитан старой шхуны! И вот он, перегнувшись через нос судна, громогласно обращается к вам, прошедшему пять лет в одиночестве, с вопросом: «Сколько студийных альбомов у Бейонсе?».

Вы должны принять решение. Можно назвать последнее известное вам значение (которое устарело на пять лет). Если вы ответите на вопрос капитана, то вы *доступны*. Или можно отказаться отвечать, понимая, что без связи с миром ваш ответ может оказаться не *согласованным* с реальностью. Капитан не получит ответа на свой вопрос, но состояние мира останется согласованным (вернувшись домой, он сможет получить правильный ответ). Играя роль опрашиваемого узла, вы можете отстоять либо *согласованность* мира, либо свою *доступность*, но не то и другое *вместе*.

Идея согласованности в конечном счете заключается в том, что каждый узел всегда доступен для ответа на запросы. А компромисс состоит в том, что изменения данных распространяются на другие узлы в фоновом режиме. Это означает, что в любой момент времени система может оказаться несогласованной, но по большому счету данные в ней точны.

Самым известным примером системы, согласованной в конечном счете, является служба доменных имен в Интернете (DNS). После ре-

¹ Игра слов: cap – капитан, «кэп». Прим. перев.

гистрации домена может пройти несколько дней, прежде чем изменение дойдет до каждого сервера в Интернете. Однако в любой момент любой DNS-сервер доступен (при условии, конечно, что вы можете к нему подключиться).

A2.2. CAP на практике

Некоторые устойчивые к потере связности базы данных можно настроить так, что они будут более или менее согласованы или доступны на уровне отдельного запроса. Так работает база Riak, которая позволяет клиенту в момент запроса указать, какой уровень согласованности ему нужен. Остальные базы данных занимают тот или иной уголок треугольника компромиссов CAP.

Приключения CAP, часть II: согласованность в конечном счете

Вернемся на два года назад, в 2009 год. Прожив на острове уже три года, вы случайно обнаружили в песке бутылку – бесценный контакт с внешним миром. Откупориваете ее и о, радость! Вы получили важный кусочек знания... Количество студийных альбомов Бейонсе играет для совокупного мирового знания огромную роль. Такую большую, что всякий раз, как у нее выходит новый альбом, кто-то записывает текущую дату и номер альбома на клочке бумаге. А потом запечатывает этот клочок в бутылку и бросает в море. Кто-то – как вы, например, – находящийся на необитаемом острове, оторванный от мира, может рассчитывать, что в *конечном счете* получит правильный ответ.

Перенесемся снова в настоящее. Когда капитан спрашивает «Сколько студийных альбомов у Бейонсе?», вы остаетесь *доступны* и отвечаете «три». Возможно, ваш ответ *не* согласуется с миром, но вы в нем достаточно уверены, так как не находили бутылку с другим ответом.

У этой истории счастливый конец – капитан спасает вас, вы возвращаетесь домой, обнаруживаете новый альбом певицы и живете долго и счастливо. Пока вы находитесь в обитаемой местности, нет нужды быть устойчивым к потере связности, и вы можете сохранять согласованность и доступность до конца своих дней.

Redis, PostgreSQL и Neo4J согласованы и доступны (CA); но данные в них не распределяются, поэтому об устойчивости к потере связности можно не думать (для нераспределенных систем теорема CAP не имеет особого смысла, хотя на эту тему можно поспорить). MongoDB и HBase, вообще говоря, согласованы и устойчивы к потере связности (CP). В случае потере связности сети они могут перестать отвечать на некоторые запросы (например, в наборе реплик Mongo для операций чтения флаг `slaveok` сбрасывается в `false`). На прак-

тике аппаратный сбой обрабатывается так, что функциональность снижается постепенно, – другие узлы, еще находящиеся в сети, могут подменить вышедший из строя сервер, – но, строго говоря, в смысле теоремы CAP они недоступны. Наконец, CouchDB доступна и устойчива к потере связности (AP). Хотя два и более серверов CouchDB могут реплицировать данные между собой, CouchDB не в состоянии гарантировать согласованность данных на любых двух серверах.

Стоит отметить, что большинство этих баз данных можно настроить, изменив тип CAP (Mongo может быть сделана CA, CouchDB – CP), но сейчас мы говорили о поведении, подразумеваемом по умолчанию.

A2.3. Компромиссный выбор задержки

Но учет теоремы CAP проектирование распределенной системы баз данных не исчерпывается. Например, для многих архитекторов главным вопросом является низкая задержка (то есть высокое быстроедействие). Если вы читали статью о системе Amazon Dynamo², то, наверное, обратили внимание, что в ней уделено внимание не только обеспечению доступности, но и требованиям Amazon к задержкам. Для определенного класса приложений даже небольшое изменение задержки может привести к крупным финансовым потерям. Знаменитая база данных Yahoo PNUTS жертвует и доступностью при эксплуатации в обычном режиме, и согласованностью при потере связности ради того, чтобы свести задержки к минимуму³. При рассмотрении распределенных баз данных, принимать во внимание теорему CAP важно, но не менее важно помнить о том, что на ней теория распределенных баз не заканчивается.

2 <http://allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>

3 <http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html>



СПИСОК ЛИТЕРАТУРЫ

- [TH01] David Thomas and Andrew Hunt. Programming Ruby: The Pragmatic Programmer's Guide. Addison-Wesley, Reading, MA, 2001.
- [Tat10] Bruce A. Tate. Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2010.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Символы

! (восклицательный знак), поиск по регулярному выражению 63
% (процент), поиск по LIKE 46
* (звездочка), поиск в Riak 119
* (звездочка), поиск по регулярному выражению 63
-> (оператор стрелка) в Groovy 275
/mapred, команда 96
<> (не равно) в PostgreSQL 36
? (вопросительный знак), поиск в Riak 119
@-, команда сURL 95
{...}, использование в Groovy 265
~ оператор, поиск по регулярному выражению 63

А

абстрактное синтаксическое дерево 55
абстрактным синтаксическим деревом 55
агрегатные функции 45
агрегированные запросы в MongoDB 191
алгоритм Кевина Бэйкона 286
арбитры и голосование в MongoDB 206
атрибуты 33;
в PostgreSQL (столбцы) 33

Б

базы данных;
выбор подходящей 21;
жанры 23

Безопасность;
аутентификация. См. Аутентификация
Блокировка. См. Параллелизм
Блокировка записи. См. Параллелизм
Блума фильтры 143, 334

В

векторные часы 110;
в теории 111;
на практике 112;
обрезание в Riak 115
версионирование в HBase 132
вершина;
и конвейер 269;
и связь 264
википедия, потоковая загрузка 140
виртуальные узлы (v-узлы) 102;
долговечная запись 108;
согласованность в конечном счете в Riak 105;
согласованность за счет записи 105;
согласованность за счет кворума 106;
согласованность за счет чтения 106
Внешние ключи. См. Связи
внешние ключи, индексы по 60
внешние соединения 39
возвратный интерфейс 130
временные метки;
в HBase 132;
в Riak 111

**Г**

граф, терминология в Gremlin 264
графовые базы данных 28, 360;
Neo4j 259

Д

двумерные индексы в MongoDB 187
декартово произведение 34
диаграмма сущность-связь 40
документные базы данных 27, 359;
уникальность 201
долгий опрос 245
Драйверы;
безопасный режим. См. getLastError,
команда;
гарантии записи. См. Гарантии
записи

Ж

журналирование, WAL 146

З

Запросы;
выборка подмножества полей. См.
Проецирование

И

Иерархические данные. См.
Деревья
Индексирование. См. также
Оптимизация запросов;
В-деревья. См. В-деревья;
массивов. См. Многоключевые
индексы
индексирование;
в MongoDB 187;
в Neo4j 282;
лексем 68
Индексы;
пространственные. См.
Пространственные индексы
индексы 41;
в PostgreSQL 41;
в Riak 120;

инвертированные 68;
определение 44;
по внешним ключам 60

К

кластер;
НА 295;
Redis 333;
запуск 163;
настройка 162;
опрос состояния 164;
подготовка 161;
подключение к 163
Коллекции;
ограниченные. См. Ограниченные
коллекции;
системные. См. Системные
коллекции
коллекции;
в Gremlin 267;
в MongoDB 186
конвейер;
и вершина 269;
операции Gremlin как 267;
подача строк в 323
конфликты;
в службе iCloud. См. iCloud, с
лужба
конфликты, разрешение с помощью
векторных часов 110
кортежи в PostgreSQL 33

Л

Левенштейна расстояние, поиск по 63
лексемы;
индексирование 68

М

масштабируемость 23
мгновенные снимки в Redis 327
метаданные;
сохранение 89;
ссылки как 85
метасимволы;

в LIKE и ILIKE 62;
в Riak 119;
в регулярных выражениях 63;
в ссылках Riak 86
метафоны 69
многомерные кубы 71
многостороннее хранение 28;
 пример службы 339;
 заполнение данными 341;
 поиск музыкальных групп 349;
 фаза 1, вставка в каноническую
 систему 344;
 фаза 1, трансформация данных 342
множества 313
Моделирование данных. См.
 Проектирование схемы

Н

нечеткий поиск 62

О

облачные службы, поставщики 160
Обновление;
 findAndModify. См. findAndModify,
 команда
ограничения;
 REFERENCES, ключевое слово 37;
 внешнего ключа 34;
 первичного ключа 33
оконные функции в PostgreSQL 48
операторы;
 в MongoDB 179;
 в поиске по регулярному
 выражению 63
Оптимизация запросов;
 explain(), метод. См. explain;
 профилирование. См. Профили-
 ровщик запросов
Оптимистическая блокировка. См.
 Параллелизм
ориентация. См. также повороты

П

Первичные ключи. См. _id, поле

первичный ключ 43;
 ObjectId в MongoDB 173;
 индекс по 41;
 как идентификатор в SQL 33;
 ограничения 33;
 составной ключ 36
подготовка;
 портал подготовки для iOS. См.
 портал подготовки для iOS
Поддокументы;
 и документы верхнего уровня. См.
 Вложение и ссылка
Подтипы. См. Двоичные данные
Позиционный оператор. См.
 Обновления операторы
поиск 62;
 ILIKE 62;
 LIKE 46, 62;
 TSQuery и TSVector 65;
 в Riak 117;
 комбинирование разных способов 71;
 метасимволы в Riak 119;
 метафоны 69;
 полнотекстовый 65;
 по ррегулярному выражению 63;
 расстояние Левенштейна 63;
 триграммы 64
полное сканирование таблицы 41
полнотекстовый поиск 65
правила в PostgreSQL 55
предметно-ориентированные языки 276
представления;
 URL-адрес для опроса 231;
 в PostgreSQL 54;
 использование для доступа
 к документам в CouchDB 224;
 круговое. См. WheelView, класс;
 создание в CouchDB 226;
 создание с помощью редукторов
 в CouchDB 239;
 сохранение в виде проектного
 документа 229;
 табличные. См. табличные
 представления
приложения 169;

имитации. См. имитации приложений
распространение. См. распространение
Проектирование схемы;
вложенные документы. См. Под-
документы;
связи. См. Связи
проектный документ, CouchDB 229
пространственные запросы,
MongoDB 208
пространственные индексы,
сферические 187
Протоколирование транзакций. См.
Журналирование

Р

распространение;
через App Store. См. распространение через App Store
ребра, в Gremlin 264
регионы, HBase 145, 146
регулярные выражения, поиск по 63
редукторы;
в CouchDB 239;
в MongoDB 199
реляционная алгебра 34
реляционное исчисление 34
реляционные СУБД 24, 356;
математические отношения 33;
сильные стороны 357;
слабые стороны 357;
сравнение с NoSQL 21;
сравнение со столбцовыми СУБД 26;
транзакции 49

С

сводные таблицы, PostgreSQL 58
сегментирование в MongoDB 206
сегменты, в Riak 83, 96
семейства столбцов,
HBase 130, 137, 150
сжатия алгоритмы, в HBase 143
следование по ссылкам 85;

с помощью mapreduce 100
событийно-управляемые неблоки-
рующие приложения 344
соединение 37;
внешнее 39;
внутреннее 38;
и MongoDB 183;
левое 41, 44;
определение 44;
правое 41
сократитель URL, разработка 307
сохранение данных;
Core Data. См. Core Data, подсистема;
списки свойств. См. списки свойств
список, тип данных в Redis 311
сравнение с обменом, функции. См.
атомарные операции,
сравнение с обменом,
функции
ссылки 85;
извлечение из текста 150;
тег next_to 86
столбцовые базы данных 26, 126, 358
столбцы;
в PostgreSQL 33;
определение 43
стоп-слова 66
строки;
в PostgreSQL 33;
определение 43
схемы;
в Neo4j 260;
в PostgreSQL 33;
диаграмма сущность-связь 40;
система рекомендаций фильмов 60

Т

таблицы;
изменение в HBase 133;
определение 43;
полное сканирование 41;
соединение 37;

создание в HBase 129;
создание в SQL 33
таймауты в Riak 101
типы данных;
 PostgreSQL 36;
 Redis 309
триггеры 52
триграммы 64

У

узлы;
 в Neo4j 262;
 добавление в Neo4j 262;
 долговечная запись в Riak 108;
 поиск пути в Neo4j 282;
 согласованность в Riak 106
упреждающая запись в журнал 126, 146
устройства;
 поворот. См. поворот;
 подготовка. См. подготовка
Уступка. См. Параллелизм

Ф

файл с дозаписью, в Redis 327
фильтры, в CouchDB 250
фильтры ключей в Riak 99

Х

хеш, тип данных в Redis 310
хеш-индекс, создание 42
хранилища ключей и значений 25, 357;
 Redis как пример 305
хранимые процедуры 50

Ц

центрированность 289

Ш

шаблоны в PostgreSQL 67
шаги трансформации в Gremlin 271
команды, psql 32

А

ACID, свойства 49
AirPlay. См. слайд-шоу
AirPrint. См. печать
Amazon;
 Elastic Compute Cloud (EC2) 161;
 веб-службы (AWS) 160, 164;
 статья о Dynamo и Riak 79
Apache Hadoop 126, 153, 295
Apache Solr 118
Avro, протокол, HBase 156

В

В-дерево;
 в MongoDB 187;
 определение 44;
 тип индекса в PostgreSQL 42
BigCouch и CouchDB 251
BigTable 126

С

CAP, теорема 367;
 в HBase 167;
 в Neo4j 303;
 в Riak 102, 123
Cassandra, база данных 26, 160
Changes API, интерфейс 243, 340
collect(), функция map в Groovy 274
CouchDB;
 _all_docs, представление 229;
 _changes, URL 244;
 _id, поле 217;
 _rev, поле 217;
 Changes API 243, 340;
 total_rows, поле 233;
 вложенные JSON-структуры 218;
 доступ к документам через
 представления 224;
 и BigCouch 251;
 и MongoDB 215;
 импорт данных 233;
 модификация записей 217;
 обновление документа с помощью
 PUT 222;

обработка конфликтов 253;
обращение к 219;
описание 27, 214;
опрос изменений с помощью
Node.js 245;
опрос проектных документов
с помощью cURL 231;
отслеживание изменений 243;
непрерывное 249;
поля 217;
разработка специфических
представлений 229;
репликация данных 252;
роль в примере системы с многосто-
ронним хранением 339;
сильные стороны 257;
слабые стороны 258;
создание документа с помощью
POST 221;
создание представлений 226;
создание представлений с помощью
редукторов 239;
сохранение представления в виде
проектного документа 229;
удаление документа с помощью
DELETE 223;
фильтрация изменений 250;
чтение документа с помощью
GET 220
CouchDB Futon;
документ со значением-массивом 218;
репликатор 253;
создание документов 217
count(), функция 46;
агрегирование результатов 192;
использование совместно
с HAVING 47
CREATE INDEX, команда SQL 42
CREATE TABLE, команда SQL 32, 41
crosstab(), PostgreSQL 58
CRUD, операции 36, 43;
соответствие глаголам HTTP 82
cURL;
и REST 82;

и Riak 79;
команда @- 95;
опрос проектных документов
CouchDB 231;
отправка GET-запросов 220;
параметр -X PUT 83;
флаг -v 83

D

DELETE FROM, команда SQL 35
describe, команда 150
distinct(), функция;
агрегирование результатов 192;
в агрегированных запросах 193
DISTINCT, ключевое слово SQL 47

E

EC2. См. также Amazon EC2
EC2 (Elastic Compute Cloud),
Amazon 161
Elastic Compute Cloud. См. EC2
Elastic MapReduce. См. EMR
elemMatch, оператор, MongoDB 178
Erlang 80, 108, 117;
скачивание 80;
язык CouchDB 215
eval(), функция, MongoDB 196
EXPLAIN, команда SQL 69
extract(), PostgreSQL 58

F

Facebook;
индексная таблица сообщений 132
find(), функция, MongoDB 175, 183
Freebase.com 341

G

GET (чтение);
в Neo4j 279;
в Redis 308;
выборка значений 84;
глагол, соответствующий операции
CRUD 82;
запрос к _changes 244;

отправка запросов 220;
чтение документов 220
get, команда Hbase 133
getLast(collection), MongoDB 197
GFS. См. Google File System
GIN (Generalized Inverted iNdex) 68
GIST (Generalized Index Search) 65
Google, MapReduce 167
Gremlin;
 алгоритмы на графах 286;
 и jQuery 269;
 и REST 284;
 и предметно-ориентированные
 языки 276;
 использование библиотек Java 264;
 и язык программирования
 Groovy 274;
 как универсальный язык обхода
 графов 264;
 как язык построения каналов 267;
 метод loop() 272;
 описание 264;
 ребра 264
Gremlin/Blueprint, проект 289
GridFS, MongoDB 210
Groovy, язык программирования 264;
 замыкание 265;
 функция распределения 274;
 функция редуцирования 274
group(), функция;
 агрегирование результатов 192;
 в агрегированных запросах 194
GROUP BY, фраза;
 в MySQL 47;
 в SQL 46;
 и PARTITION BY 48;
 и оконные функции 48
groupCount(), Neo4j 289

H

HA-кластер, в Neo4j 295, 299
HAVING, фраза SQL 47
HBase 125;
 Whirr, подготовка кластера 160;

 алгоритмы сжатия 143;
 временные метки 132;
 вставка данных 131;
 выборка данных 131;
 выполнение скриптов 139;
 добавление данных из программы 136;
 запуск 128;
 значения по умолчанию 134;
 изменение таблицы 135;
 импорт данных 139;
 использование места на диске 145;
 и теорема CAP 167;
 как столбцовая база данных 26;
 команды get и put 133;
 конфигурирование 128;
 обновление данных 131;
 оболочка 129, 136;
 останов 128;
 подготовка к работе в облаке 160;
 построение сканера 150;
 поточная загрузка данных из вики-
 педии 141;
 приложение, работающее
 по протоколу Thrift 158;
 протоколы для подключения
 клиентов 156;
 работа с "большими данными" 139;
 регионы 145, 146;
 режимы работы 127;
 свойства 130;
 семейства столбцов 130, 132, 137, 150;
 сетевые настройки 129;
 сильные стороны 166;
 слабые стороны 167;
 словарь 129;
 создание таблиц 129;
 типы данных 167;
 фильтры Блума 143;
 хранение данных о ссылках 150
HBaseMaster 149
HDFS (распределенная файловая
 система Hadoop) 167
Homebrew для Mac 307
HTTP, заголовки и коды ошибок

в Riak 83
HTTP, теги сущностей (Etags) 114
Hypertable, база данных 26

I

ILIKE, поиск 62
inject(), функция reduce в Groovy 274
INSERT INTO, команда SQL 36
Interface Builder. См. IB

J

Jamendo, сайт 239
Java, виртуальная машина 136
Java API, протокол HBase 156
JavaScript;
 вызов из точки подключения перед
 фиксацией в Riak 116;
 как родной язык MongoDB 173;
 функция reduce 98
jQuery;
 и Gremlin 269
JRuby;
 и Apache Hadoop 153;
 командная оболочка HBase 129
JSON;
 в Neo4j 280;
 документы 170;
 объекты в CouchDB 218, 225;
 сериализованное уведомление
 об изменениях 249;
 формат возврата результата в Riak 119;
 формат транспортировки данных 215
JUNG (Java Universal Network/
 Graph), каркас 289

L

LIKE, поиск 62;
 метасимвол % 46
listCommands(), функция
 MongoDB 195

M

map, функция;
 в Groovy 274;

в MongoDB 199;
результаты в Riak 94;
хранение в виде значения
 сегмента 97
mapreduce;
 в CouchDB 239, 241;
 в MongoDB 197;
 в Riak 95;
 введение 92;
 поиск объектов с помощью 192;
 распределение объектов с общим
 ключом 93;
 следование по ссылкам 100
max() 46
membase, база данных 25
memcached, база данных 25
MIME, типы;
 multipart/mixed 87;
 в Riak 89
min() 46
MongoDB 27, 169;
 count(), функция 192;
 distinct(), функция 192;
 elemMatch, оператор 178;
 eval(), функция 196;
 find(), функция 175, 183;
 getLast(collection) 197;
 GridFS 210;
 group(), функция 192;
 listCommands(), функция 195;
 mapreduce 197;
 ObjectId 173;
 runCommand(), функция 195;
 агрегированные запросы 191;
 вложенные массивы 177;
 голосование и арбитры 206;
 индексирование 187;
 использование JavaScript 173;
 коллекции 186;
 команды 173;
 команды на стороне сервера 194;
 конструирование произвольных
 запросов 175;
 наборы реплик 202;

обновление 181;
операторы 180;
поиск документов 175;
построение индексов по вложенным значениям 190;
проблема четного числа узлов 205;
пространственные запросы 208;
редукторы 199;
сегментирование 206;
сильные стороны 212;
слабые стороны 212;
соединение 183;
создание JavaScript-функций 174;
сокращенная нотация для простых функций принятия решения 185;
сравнение с CouchDB 215;
сравнение с mongoconfig 207;
ссылки 183;
удаление документов 184;
установка 171;
чтение данных из программы 185

N

Neo4j;
 инвертированные индексы для полнотекстового поиска 283
Neo4J, база данных 28, 259;
 Gremlin;
 и REST 284;
 groupCount(), функция 289;
 HA-кластер 295, 298;
 REST-интерфейс 279;
 алгоритмы на графах 286;
 большие наборы данных 284;
 веб-интерфейс 262;
 граф узлов 267;
 добавление узлов 262;
 индексирования 282;
 использование JUNG 289;
 каналы в Gremlin 267;
 координатор Zookeeper 295, 297;
 метод loop() 272;
 моделирование на доске 259;

 нахождение пути между двумя узлами 282;
 обновление 278;
 обход графа 261;
 останов главного сервера 300;
 построение кластера 296;
 предметно-ориентированные языки 276;
 проверка результатов репликации 299;
 проверка состояния кластера 299;
 режим высокой доступности 294, 297;
 резервное копирование 300;
 роль в примере системы с многосторонним хранением 339;
 сильные стороны 302;
 слабые стороны 303;
 случайный обход узлов 288;
 схема системы рекомендаций 260;
 транзакции ACID 293;
 удаление 278;
 язык Gremlin 264;
 язык программирования Groovy 274
netcat 323
next_to, использование в ссылках 86
Node.js;
 использование в системах с многосторонним хранением 341;
 опрос изменений 245
NoSQL, сравнение с реляционными базами данных 21
NULL, значения 36
почки подключения, до и после фиксации 115

P

PageRank, Google 289
PARTITION BY, сравнение с GROUP BY 48
PL/pgSQL 52
POST (создание);
 генерация ключа с помощью 84;
 глагол, соответствующий операции CRUD 82;
 создание документов с помощью 221
PostgreSQL, база данных 24, 30;

crosstab() 58;
extract() 58;
агрегатные функции 45;
анализаторы 67;
встроенная документация 32;
генерация вектора лексем 66;
оболочка 32;
оконные функции 48;
поиск 62;
поиск с применением индексов 41;
правила 55;
представления как правила 56;
работа с таблицами 33;
сводные таблицы 57;
сильные стороны 76;
слабые стороны 77;
соединение таблиц 37;
создание представлений 54;
типы данных 36;
транзакции 49;
триггеры 52;
установка 31;
хранимые процедуры 50;
шаблоны 67
psql, оболочка PostgreSQL 32
PUT (обновление);
 глагол, соответствующий операции
 CRUD 82;
 обновление документов
 с помощью 222;
 создание сегментов в Riak 82
put, команда Hbase 133

R

RackSpace Cloud Servers 160
redis, gem-пакет Ruby 331
Redis, база данных 25, 305;
 DEL команда 320;
 DISCARD команда 309;
 EXPIRE команда 318;
 FLUSHALL команда 321, 328;
 FLUSHDB команда 320;
 GETBIT команда 337;
 GET команда 308;

INCR команда 308;
INFO команда 325;
LASTSAVE команда 326;
LRANGE команда 316;
MGET команда 308;
MOVE команда 320;
MSET команда 308;
MULTI команда 309;
PERSIST команда 319;
RENAME команда 320;
SADD команда 330;
SAVE команда 326;
SDIFF команда 314;
SETBIT команда 337;
SETEX команда 319;
SET команда 307;
SINTER команда 314;
SMEMBERS команда 314;
TYPE команда 320;
ZRange команда 316;
безопасность 328;
блокирующие списки 313;
долговечность 326;
дополнительные параметры 329;
загрузка данных 330;
информация о сервере 325;
как основа серверной части
 системы 307;
как хранилище ключей
 и значений 305;
кластер 330;
командный интерфейс 307;
настройка 325;
отправка команд через telnet 322;
отсортированные множества 315;
параметры режима сохранения 326;
подключение к серверу 307;
получение диапазона 316;
пространства имен 319;
публикация и подписка 323;
репликация главный-подчинен-
 ный 330;
роль в примере системы с много-
 сторонним хранением 339;

сильные стороны 353;
слабые стороны 354;
типы данных 308;
транзакции 309;
фильтры Блума 334
reduce, функция;
в CouchDB 242;
в Groovy 274;
в JavaScript 98;
в MongoDB 199;
преобразование в скалярные
значения 92
REFERENCES, ключевое слово
SQL 36, 37
REST (Representational State
Transfer) 82;
в CouchDB 215, 219;
в HBase 156;
и Gremlin 284;
интерфейс с Neo4j 279
RETURNING, ключевое слово
SQL 39
Riak, база данных 25, 78;
-X PUT параметр 83;
app.config 115;
restart, аргумент 116;
timeout, параметр 101;
векторные часы 110;
временные метки 111;
встроенные функции 97;
долговечная запись 108;
заполнение сегментов 83;
и cURL 79;
и веб 79;
индексы 120;
интерфейс SOLR 118;
и теорема CAP 123;
как хранилище ключей
и значений 83;
кольцо 102;
настройка кластера 102;
поиск 117;
сильные стороны 123;
слабые стороны 123;

статья о системе Amazon
Dynamo 79;
типы MIME 87;
узлы и v-узлы 102;
установка 80;
фильтры ключей 99;
функции map и reduce 95;
хранимые функции 97
ROLLBACK;
команда SQL 49;
сравнение с командой Redis
DISCARD 309
Ruby;
загрузка данных в Riak 91;
импорт данных в CouchDB 233;
поддержка в PostgreSQL 52
runCommand(), функция
MongoDB 195

S

SELECT...FROM, команда SQL 35
SERIAL, ключевое слово 38
SQL;
агрегатные функции 45;
и PostgreSQL 24;
исполнение запроса в PostgreSQL 56;
ключевые слова;
DISTINCT 47;
ON 38;
REFERENCES 36;
RETURNING 39;
SERIAL 38;
UNIQUE 42;
команды;
CREATE INDEX 42;
CREATE TABLE 32;
DELETE FROM 35;
EXPLAIN 68;
INSERT INTO 35;
ROLLBACK 49;
SELECT...FROM 35;
справка в PostgreSQL 32;
ограничение внешнего ключа 34;
первичный ключ 33;

соединение 37;
стандартный поиск строки 62;
фразы;
GROUP BY 46;
HAVING 47;
WHERE 34

T

Thrift, протокол HBase;
генерация моделей 157;
разработка клиентского
 приложения 158;
установка 157
TSQuery, полнотекстовый поиск 65

U

UPDATE, команда SQL 37, 54

V

Voldemort, база данных 25, 160

W

WAL (журнал упреждающей
 записи) 146
Whirr;
 введение 160;
 настройка кластера 162;
 подготовка 161;
 подготовка к работе в облаке 160;
 уничтожение кластера 164
WikiMedia Foundation, выгруженные
 наборы данных 139

X

X-Riak-Meta-, префикс 89
XML, потоковая загрузка 140

Z

ZooKeeper 160, 167, 295, 297

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. **(499) 725-54-09, 725-50-27**;

Электронный адрес **books@alians-kniga.ru**.

Эрик Редмонд, Джим. Р. Уилсон

под редакцией Жаклин Картер

Семь баз данных за семь недель.
Введение в современные базы данных
и идеологию NoSQL

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru

Перевод с английского *Слинкин А. А.*

Корректор *Синяева Г. И.*

Верстка *Паранская Н. В.*

Дизайн обложки *Мовчан А. Г.*

Подписано в печать 20.10.2012. Формат 60×90 ¹/₁₆.

Гарнитура «Петербург». Печать офсетная.

Усл. печ. л. 23,52. Тираж 200 экз.

Web-сайт издательства: www.dmk-press.ru

"Никогда еще не было столько баз данных промышленного качества с открытым исходным кодом. О них необходимо знать. Каждый профессионал в области разработки ПО должен прочитать эту книгу."
Д-р Ник Уильямс, вице-президент по системной инженерии, Engine Yardians

Семь баз данных за семь недель

Совершите экскурсию по самым популярным из современных баз данных с открытым исходным кодом: Redis, Neo4J, CouchDB, MongoDB, HBase, PostgreSQL и Riak. Для каждой базы будет приведен пример работы с реальными данными, демонстрирующий основные идеи и сильные стороны.

Вы узнаете о разительных отличиях между MongoDB и CouchDB, а также о том, что в основу Riak положена система Dynamo. Заставьте свои приложения работать быстрее, воспользовавшись Redis. Храните связи в Neo4J. Используйте MapReduce для решения задач с «большими данными». Создайте серверный кластер с помощью таких масштабируемых служб, как Amazon Elastic Compute Cloud (EC2).

Эта книга прольет свет на сильные и слабые стороны каждой из семи баз данных и научит вас выбирать ту, которая лучше отвечает вашим требованиям.

Эрик Редмонд (Eric Redmond) работает в индустрии программного обеспечения свыше 15 лет. Он выступает в роли программиста, иллюстратора, докладчика на международных конференциях и активного организатор нескольких технических групп. **Джим Р. Уилсон (Jim R. Wilson)** увлекся компьютерами в 13 лет и с тех пор ни разу не пожалел об этом. Интересоваться базами данных типа NoSQL он начал в 2007 году и внес свой вклад в такие крупномасштабные проекты с открытым исходным кодом, как MediaWiki и HBase.

Internet-магазин:
www.dmk-press.ru
Книга — почтой:
e-mail: orders@aliens-kniga.ru
Оптовая продажа:
«Альянс-книга»
Тел./факс: (499) 725-5409
e-mail: books@aliens-kniga.ru



ISBN 978-5-94074-866-3



9 785940 748663 >