MongoDB

В ДЕЙСТВИИ

Кайл Бэнкер

MANNING

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Кайл Бэнкер

MongoDB в действии

Москва, 2012

MongoDB in Action

KYLE BANKER

MANNING
SHELTER ISLAND

MongoDB в действии

КАЙЛ БЭНКЕР

Москва, 2012

УДК 004.6:004.42 MongoDB
ББК 32.973.26-018.2
Б71

Б71 Кайл Бэнкер

MongoDB в действии. / Пер. с англ. Слинкина А. А. – М.: ДМК Пресс, 2012. – 394с.: ил.

ISBN 978-5-94074-831-1

MongoDB – это документо-ориентированная база данных, предназначенная для гибкой, масштабируемой и очень быстрой работы даже при больших объемах данных. При ее проектировании изначально закладывалась высокая доступность, поддержка сложных динамических схем и простое распределение данных по нескольким серверам.

Эта книга представляет собой введение в MongoDB и документо-ориентированную модель данных. Она дает не только общую картину, необходимую разработчику, но и достаточно деталей, чтобы удовлетворить системного инженера. Многочисленные примеры помогут обрести уверенность в области моделирования данных – вопросе, который необычайно важен для разработки ПО. Вам понравится углубленное изложение различных функциональных возможностей, в том числе репликации, автосегментирования и развертывания.

УДК 004.6:004.42 MongoDB
ББК 32.973.26-018.2

ISBN 978-1-93518-287-0 (англ.)
ISBN 978-5-94074-831-1 (рус.)

Эта книга посвящается миру и человеческому достоинству, а также всем, кто трудится во имя достижения этих идеалов.

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Базы данных – рабочие лошадки информационной эпохи. Как атланты, они незримо подпирают цифровой мир, в котором мы обитаем. Легко позабыть, что всё наше цифровое общение, будь то комментарии в форумах, сообщения в Твитере, поиск или сортировка, – это по существу взаимодействие с базой данных. Размышляя об этой краеугольной и в то же время скрытой функции, я испытываю чувство благоговейного трепета, сродни тому, которое возникает, когда идешь по подвесному месту, предназначенному в основном для автомобилей.

Базы данных принимают самые разные формы. Предметные указатели в книгах, каталожные карточки, когда-то стоявшие в библиотеках, – это тоже базы данных, как и специально структурированные текстовые файлы, излюбленные программистами былых времен, работавшими на языке Perl. Но, наверное, чаще всего базы данных ассоциируются с изощренными реляционными СУБД, лежащими в основе огромного количества современных программных систем, теми самыми СУБД, на которых делаются состояния. Реляционные базы, с их идеализированной третьей нормальной формой и выразительным SQL-интерфейсом, все еще вызывают уважение старой гвардии – и вполне заслуженно.

Но несколько лет назад, разрабатывая веб-приложения, я мечтал опробовать только появляющиеся на свет альтернативы правящим в мире реляционным базам. Открыв для себя MongoDB, я сразу понял – это как раз то, что мне надо. Мне понравилась идея использовать JSON-подобные структуры для представления данных. Язык JSON прост, интуитивно понятен и удобен для человека. И тот факт, что язык запросов в MongoDB также основан на JSON вселяет в пользователя этой новой СУБД ощущение комфорта и гармонии. На первом месте стоит интерфейс. А наличие таких привлекательных средств, как простая репликация и сегментирование, только подстегивает интерес к пакету. Но окончательно мое обращение состоялось, когда я написал несколько приложений на основе MongoDB и в полной мере осознал, как легко дается разработка.

По совершенно невероятному стечению обстоятельств я перешел на работу в компанию 10gen, которая была зачинателем разработки этой СУБД с открытым кодом. В течение двух лет у меня была возможность совершенствовать различные клиентские драйверы и общаться с многочисленными заказчиками, помогая им развертывать системы на основе MongoDB. Накопленный опыт и вылился в книгу, которую вы держите в руках. MongoDB активно развивается и пока еще далека от совершенства. Но она уже поддерживает тысячи приложений, использующих крупные и мелкие кластеры базы данных, и с каждым днем становится все более зрелой. Многие разработчики признавались, что при работе с ней испытывают ощущение чуда и даже счастья. Надеюсь, то же самое произойдет и с вами.

БЛАГОДАРНОСТИ

Спасибо коллективу издательства Manning за помощь в осуществлении замысла этой книги. Вместе с Майклом Стивенсом (Michael Stephens) мы разработали план книги, а редакторы, Сара Онстин (Sara Onstine) и Джефф Блейл (Jeff Bleiel) довели текст до окончательной формы, помогая на протяжении всего пути. Спасибо вам.

Написание книги – предприятие, отнимающее много времени, и, возможно, я не смог бы выделить время для ее завершения, если бы не великодушие Элиота Горовица (Eliot Horowitz) и Дуайта Мерримана (Dwight Merriman). Своим существованием проект MongoDB обязан инициативе и мастерству Элиота и Дуайта, а мне они доверили документирование. Большое им спасибо. Многие изложенные в этой книге мысли проистекают из бесед с коллегами по компании 10gen. В этой связи хочется выразить благодарность Майку Дирольфу (Mike Dirolf), Скотту Эрнандесу (Scott Hernandez), Элвину Ричардсу (Alvin Richards) и Матиасу Стирну (Mathias Stearn). Особенно я в долгу перед Кристиной Ходороу (Kristina Chowdorow), Ричардом Кройтером (Richard Kreuter) и Аароном Стейплом (Aaron Staple) за квалифицированные рецензии на целые главы.

На различных стадиях работы над рукописью ее читали следующие рецензенты, которым я благодарен за ценные замечания: Кэвин Джексон (Kevin Jackson), Харди Ферентшик (Hardy Ferentschik), Дэвид Синклер (David Sinclair), Крис Чэндлер (Chris Chandler), Джон Ньюнмейкер (John Nunemaker), Роберт Хансон (Robert Hanson), Альберто Лернер (Alberto Lerner), Рик Вагнер (Rick Wagner), Райан Кокс (Ryan Cox), Энди Бруткуль (Andy Brudtkuhl), Дэниэл Бретуа (Daniel Bretoi), Грег Дональд (Greg Donald), Шон Рейли (Sean Reilly), Кэртис Миллер (Curtis Miller), Санше Диге (Sanchet Dighe), Филипп Хэллстрем (Philip Hallstrom) и Энди Дингли (Andy Dingley). Спасибо Элвину Ричардсу за обстоятельное научное редактирование окончательного варианта рукописи непосредственно перед сдачей в печать.

И почетное место отвожу благодарности моей изумительной супруге, Доминике, за терпение и поддержку и моему чудесному сыну, Оливеру, просто за то, что он такой замечательный.

ОБ ЭТОЙ КНИГЕ

Эта книга предназначена разработчикам приложений и администраторам баз данных, которые хотели бы узнать всё о MongoDB, начиная с самых основ. Если вы раньше не работали с MongoDB, то найдете здесь учебное пособие, не слишком стремительное. Если вы уже пользуетесь этой СУБД, то вам пригодятся более детальные справочные разделы, которые помогут восполнить пробелы в знаниях. Что касается глубины изложения, то материал должен удовлетворить всех, кроме самых продвинутых пользователей.

Примеры написаны на JavaScript, языке оболочки MongoDB, и на Ruby, популярном скриптовом языке. Я стремился, чтобы примеры были простыми и в то же время полезными, и не использовал сколько-нибудь сложных языковых средств JavaScript и Ruby. Основная цель – представить MongoDB API в максимально доступном виде. Если вы знакомы с другими языками программирования, то легко разберетесь в примерах.

И еще одно замечание о языках. Если у вас возникает вопрос «Почему бы не использовать в этой книге язык X?», можете расслабиться. Все официально поддерживаемые драйверы MongoDB предоставляют единообразный API. Это означает, что изучив основы API для какого-нибудь одного драйвера, вы легко освоите и все остальные. А чтобы помочь вам в этом, в приложении D приведен обзор драйверов для языков PHP, Java и C++.

Как пользоваться этой книгой

Эта книга представляет собой одновременно учебное пособие и справочное руководство. Если вы ничего не знаете о MongoDB, то имеет смысл читать ее по порядку. В тексте много примеров, которые вы можете выполнять для закрепления пройденного. Как минимум, необходимо установить саму MongoDB; драйвер Ruby желателен, но не обязателен. Инструкции по установке приведены в приложении A.

Если вы уже работали с MongoDB, то, возможно, вас будут интересовать какие-то конкретные вопросы. Главы 7 – 10 и все приложения не зависят друг от друга, читать их можно в любом порядке.

Кроме того, главы 4 – 6 содержат «основные моменты» – фундаментальные принципы системы. К ним также можно обращаться из любого места.

Организация материала

Книга состоит из трех частей.

Часть 1 представляет собой сквозное введение в MongoDB. В главе 1 приводится обзор истории MongoDB, основных возможностей и сценариев использования. В главе 2 основные идеи СУБД излагаются в виде учебного пособия с использованием командной оболочки MongoDB. В главе 3 приводится пример проектирования простого приложения на основе MongoDB.

Часть 2 содержит более подробное описание MongoDB API, представленного в части 1. В трех входящих в нее главах рассматривается разработка приложения для электронной торговли, начиная со схемы базы данных и заканчивая различными операциями. В главе 4 речь пойдет о документах, наименьшей единице хранения данных в MongoDB, здесь же описывается проектирование простой схемы. В главах 5 и 6 мы будем работать с этой схемой и познакомимся с запросами для выборки и обновления данных. Дополнительно в каждой из глав второй части содержатся углубленные комментарии к излагаемому материалу.

Часть 3 посвящена вопросам производительности и промышленной эксплуатации. В главе 7 подробно рассматривается индексирование и оптимизация запросов. Тема главы 8 – репликация, с упором на стратегию развертывания MongoDB для обеспечения высокой доступности и масштабируемости операций чтения. В главе 9 описывается сегментирование – ответ MongoDB на проблему горизонтальной масштабируемости. В главе 10 приведены рекомендации по развертыванию, администрированию и отладке систем на основе MongoDB.

В книге имеется пять приложений. В приложении A рассматривается установка MongoDB и Ruby (как пример драйвера) в Linux, Mac OS X и Windows. В приложении B приведен ряд паттернов проектирования схемы и приложения, а также несколько антипаттернов. В приложение C рассказано о работе с двоичными данными в MongoDB и об использовании GridFS – реализованной во всех драйверах спецификации хранения в базе очень больших файлов. Приложение D содержит сравнительный анализ драйверов для языков PHP, Java и C++. В приложение E показано, как использовать пространственные индексы для запроса по географическим координатам.

Графические выделения и загрузка кода

Исходный код в листингах и в тексте набран моноширинным шрифтом, выделяющим его на фоне обычного текста.

Некоторые листинги сопровождаются аннотациями, в которых обсуждаются важные идеи. Иногда фрагменты кода помечаются числами, на которые потом даются ссылки в пояснениях. Поскольку MongoDB – проект с открытым кодом, компания 10gen поддерживает систему сбора информации об ошибках, открытую для всего сообщества. В нескольких местах, по преимуществу в сносках, вы встретите ссылки на извещения об ошибках и о планируемых усовершенствованиях. Например, заявка на добавление средств полнотекстового поиска имеет номер SERVER-380. Чтобы узнать о состоянии любой заявки, перейдите в браузере по адресу http://jira.mongodb.org и введите ее идентификатор в поисковое поле.

Исходный код приведенных в книге примеров, а также тестовые данные, можно скачать с сайта книги по адресу http://mongodb-book.com, а также с сайта издательства по адресу http://manning.com/MongoDBinAction.

Требования к программному обеспечению

Чтобы получить от книги максимальную пользу, необходимо установить MongoDB на свой компьютер. Инструкции по установке приведены в приложении А и на официальном сайте MongoDB (http://mongodb.org).

Если вы собираетесь выполнять примеры, написанные на Ruby, то надо будет установить Ruby. Инструкции также имеются в приложении А.

Автор в Сети

Приобретение книги «MongoDB в действии» открывает бесплатный доступ к закрытому форуму, организованному издательством Manning Publications, где вы можете оставить свои комментарии к книге, задать технические вопросы и получить помощь от автора и других пользователей. Получить доступ к форуму и подписаться на список рассылки можно на странице www.manning.com/MongoDBinAction. Там же написано, как зайти на форум после регистрации, на какую помощь можно рассчитывать, и изложены правила поведения в форуме.

Издательство Manning обязуется предоставлять читателям площадку для общения с другими читателями и автором. Однако это

не означает, что автор обязан как-то участвовать в обсуждениях; его присутствие на форуме остается чисто добровольным (и не оплачивается). Мы советуем задавать автору хитроумные вопросы, чтобы его интерес к форуму не угасал!

Форум автора в сети и архивы будут доступны на сайте издательства до тех пор, пока книга не перестанет печататься.

ОБ ИЛЛЮСТРАЦИИ НА ОБЛОЖКЕ

Рисунок на обложке книги «MongoDB в действии» называется «Бургундец», то есть житель исторической области Бургундия на северо-востоке Франции. Иллюстрация взята из опубликованного в девятнадцатом веке во Франции четырехтомного справочника местных традиционных костюмов. Все рисунки исполнены и раскрашены вручную. Богатое собрание Марешаля живо напоминает нам о тех огромных культурных различиях, которые существовали между городами и регионами каких-то 200 лет назад. Изолированные друг от друга, люди говорили на разных языках и диалектах. Встретив человека на улице или в деревне, можно было по одежде легко определить, откуда он родом и чем занимается.

Манера одеваться и общий уклад жизни с тех пор сильно изменились, и различия между областями, когда-то столь разительные, сгладились. Теперь трудно отличить друг от друга даже выходцев с разных континентов, что уж говорить о деревеньках и городках. Мы обменяли культурное разнообразие на иное устройство личной жизни – основанное на многостороннем и стремительном технологическом развитии.

В эпоху, когда одну книгу по программированию трудно отличить от другой, издательство Manning откликается на новации и инициативы в компьютерной отрасли обложками своих книг, на которых представлено широкое разнообразие местных укладов быта в позапрошлом веке. Мы возвращаем его в том виде, в каком оно запечатлено на рисунках Марешаля.

Часть 1

ПРИСТУПАЯ К РАБОТЕ

Эта часть книги представляет собой общее практическое введение в MongoDB. Здесь же мы познакомимся с оболочкой, в которой применяется язык JavaScript, и с драйвером для Ruby. Примеры в книге написаны именно на этих языках.

В главе 1 мы поговорим об истории СУБД MongoDB, целях, поставленных при ее проектировании, и о сценариях использования. Мы также сравним MongoDB с другими СУБД класса «NoSQL» и отметим, в чем ее уникальность.

В главе 2 вы заговорите на языке оболочки MongoDB — познакомитесь с основами языка запросов MongoDB и потренируетесь в создании, выборке, обновлении и удалении документов. Заканчивается глава рассмотрением нескольких продвинутых приемов работы с оболочкой и дополнительных команд MongoDB.

Глава 3 содержит введение в драйверы MongoDB и внутренний формат данных — BSON. Здесь мы узнаем, как обращаться к базе данных из программ на языке Ruby, и напишем на Ruby простенькое приложение, демонстрирующее гибкость и мощь языка запросов MongoDB.

ГЛАВА 1.
База данных для современной веб

В этой главе:

- История, проектные цели и основные особенности MongoDB.
- Краткое введение в оболочку и драйверы.
- Сценарии использования и ограничения.

Если за последние несколько лет вам доводилось писать веб-приложения, то, вероятно, в качестве основного хранилища данных вы пользовались какой-нибудь реляционной базой и, скорее всего, она демонстрировала прекрасную производительность. Большинство разработчиков знакомы с языком SQL и могут оценить красоту хорошо нормализованной модели данных, необходимость транзакций и те гарантии долговечности, которые дает СУБД. И даже если вам не нравится работать с реляционной базой данных напрямую, существует масса инструментов – от административных консолей до систем объектно-реляционного отображения, которые помогают справиться со сложностью и громоздкостью. Проще говоря, реляционные СУБД хорошо известны и достигли зрелости. Поэтому, когда небольшая, но громогласная группа разработчиков начинает агитировать за альтернативные хранилища данных, возникает вопрос: насколько полезны и жизнеспособны эти новые технологии? Смогут ли они стать заменой реляционным СУБД? Кто использует их в производственных системах и почему? На какие компромиссы придется пойти, если переходить к нереляционной базе данных? Все эти вопросы сводятся к одному: почему разработчики проявляют интерес к MongoDB?

MongoDB – это система управления базами данных, «заточенная» под веб-приложения и инфраструктуру Интернета. Модель данных и стратегия их постоянного хранения спроектированы для достижения высокой пропускной способности чтения и записи и обеспечивает простую масштабируемость с автоматическим переходом на резервный ресурс в случае отказа. Сколько бы узлов ни требовалось приложению – один или десятки, – MongoDB сумеет обеспечить поразительно высокую производительность. Того, кто раньше мучился с масштабированием реляционных баз, эта новость обрадует. Но не всем приложениям необходим крупный масштаб. Быть может, вам всегда хватало одного сервера базы данных. Зачем тогда использовать MongoDB?

Как выясняется, привлекательность MongoDB объясняется в первую очередь не стратегией масштабирования, а интуитивно понятной моделью данных. Учитывая, что с помощью документо-ориентированной модели можно представить развитые иерархически организованные структуры данных, часто оказывается возможно обойтись без присущих реляционным СУБД сложностей, связанных с соединением нескольких таблиц. Пусть, например, вы моделируете товары для сайта интернет-магазина. В полностью нормализованной базе данных информация об одном товаре может быть разбросана по десяткам таблиц. Чтобы получить его представление в интерактивной оболочке СУБД, придется написать сложный SQL-запрос с кучей соединений. Поэтому разработчики, как правило, обращаются к дополнительным программам, когда хотят собрать разрозненные данные в нечто осмысленное.

С другой стороны, в документной модели большая часть информации о товаре может быть представлена в виде одного документа. В оболочке MongoDB, построенной на основе языка JavaScript, нетрудно получить полное представление о товаре в виде иерархически организованной JSON-подобной структуры[1]. К ней можно предъявлять запросы, ей можно манипулировать. Средства составления запросов в MongoDB специально спроектированы для работы со структурированными документами, но так, чтобы пользователь, имеющий опыт работы с реляционными базами, располагал сравнимой выразительной мощью. К тому же, сегодня большинство разработчиков пишут на объектно-ориентированных языках, поэтому им нужно такое хра-

[1] JSON – это сокращение от *JavaScript Object Notation*. Как мы вскоре увидим, JSON-структуры состоят из ключей и значений и допускают произвольную глубину вложенности. Они аналогичны словарям и хешам в других языках программирования.

нилище, которое было бы проще отобразить на объекты. В случае MongoDB объект, определенный на языке программирования, сохраняется «как есть» – без дополнительной сложности, привносимой системами объектно-реляционного отображения.

Если различие между табличным и объектным представлением данных для вас новость, то, наверное, у вас возникла куча вопросов. Но потерпите немного – после того как в этой главе я расскажу о проектных целях и возможностях MongoDB, вам станет ясно, почему такие компании, как Geek.net (SourceForge.net) и газета The New York Times стали использовать в своих проектах MongoDB. Мы познакомимся с историей MongoDB и совершим экскурсию по ее основным средствам. Затем мы поговорим о некоторых альтернативах и о так называемом движении NoSQL[2], показав место MongoDB в общей картине. Наконец, я расскажу, в каких ситуациях MongoDB работает прекрасно, а когда лучше предпочесть другое хранилище данных.

1.1. Рожденная в облаке

История MongoDB, хоть и коротка, но заслуживает упоминания, потому что родилась эта СУБД в ходе работы над гораздо более амбициозным проектом. В середине 2007 года только что образованная компания 10gen приступила к разработке проекта «программная платформа как услуга». Идея была в том, чтобы создать сервер приложений и базу данных, которые могли бы служить хостингом для веб-приложений, обеспечивая масштабирование по мере необходимости. Как и система AppEngine, созданная Google, платформа компании 10gen проектировалась с расчетом на автоматическое масштабирование и управление программной и аппаратной инфраструктурой. В итоге 10gen обнаружила, что большинство разработчиков не готовы отдать в чужие руки управление своим технологическим хозяйством, но зато новая технология баз данных оказалась востребованной. Поэтому компания решила сосредоточить усилия исключительно на этой СУБД, которая получила название MongoDB.

Теперь, когда признание MongoDB ширится, а количество промышленных установок – крупных и не очень – растет, компания 10gen продолжает спонсировать разработку СУБД в качестве проекта с открытым кодом. Код общедоступен, открыт для модификации

---

[2] Собирательный термин *NoSQL* был введен в обращение в 2009 году для обозначения многочисленных нереляционных СУБД, которые в то время набирали популярность.

и использования на условиях лицензии. Всё сообщество призывают сообщать об ошибках и предлагать исправления. Тем не менее, разработкой ядра занимаются исключительно основатели или работники компании 10gen, а планы развития проекта по-прежнему определяются нуждами пользователей и главной целью – создать такую СУБД, которая объединяла бы в себе лучшие черты реляционных баз данных и распределенных хранилищ ключей и значений. Таким образом, бизнес-модель 10gen в основном такая же, как в других хорошо известных компаниях, занимающихся ПО с открытым кодом: поддерживать разработку продукта и предлагать абонентское обслуживание конечным пользователям.

В этой истории заключено два важных урока. Во-первых, MongoDB изначально разрабатывалась для платформы, которая по определению нуждалась в базе данных, допускающей автоматическое масштабирование на несколько машин. Во-вторых, MongoDB проектировалась как хранилище данных для веб-приложений. Как мы увидим, именно тот факт, что MongoDB задумана как горизонтально масштабируемое первичное хранилище данных, и выделяет ее на фоне других современных СУБД.

1.2. Основные особенности MongoDB

База данных в значительной мере определяется своей моделью данных. В этом разделе мы рассмотрим документную модель данных, а затем покажем, какие особенности MongoDB позволяют эффективно работать с этой моделью. Мы также обсудим вопросы промышленной эксплуатации MongoDB, уделив особое внимание средствам репликации и стратегии горизонтального масштабирования.

1.2.1. Документная модель данных

Модель данных MongoDB является документо-ориентированной. Для тех, кто не знаком с идеей документа в контексте баз данных, продемонстрировать ее проще всего на примере.

Листинг 1.1. Документ, представляющий одну статью на социальном новостном сайте

```
{  _id: ObjectID('4bd9e8e17cefd644108961bb'),
   title: 'Adventures in Databases',
```

Поле _id – первичный ключ

```
   url: 'http://example.com/databases.txt',
   author: 'msmith',
   vote_count: 20,

   tags: ['databases', 'mongodb', 'indexing'],

   image: {
      url: 'http://example.com/db.jpg',
      caption: '',
      type: 'jpg',
      size: 75381,
      data: "Binary"
   },

   comments: [
      {  user: 'bjones',
         text: 'Interesting article!'
      },
      {  user: 'blogger',
         text: 'Another related article is at http://example.com/db/db.txt'
      }
   ]
}
```

❶ **Теги хранятся в виде массива строк**

❷ **Атрибут указывает на другой документ**

❸ **Комментарии хранятся в виде массива объектов, представляющих один комментарий**

В листинге 1.1 приведен пример документа, представляющего статью на социальном новостном сайте (вспомните о Digg). Как видите, документ – это по существу набор, состоящий из имен и значений свойств. Значение может быть представлено простым типом, например: строки, числа и даты. Но может быть также массивом и даже другим документом ❷. С помощью таких конструкций можно представлять весьма сложные структуры данных. Так, в нашем примере имеется свойство `tags` ❶ – массив, в котором хранятся ассоциированные со статьей теги. Но еще интереснее свойство `comments` ❸, которое ссылается на массив документов, содержащих комментарии.

Сделаем небольшую паузу и сравним это с представлением тех же данных в стандартной реляционной базе. На рис. 1.1 изображен типичный реляционный аналог. Поскольку таблицы по сути своей плоские, для представления связей типа один-ко-многим необходимо несколько таблиц. Мы начинаем с таблицы posts, в которой хранится основная информация о каждой статье. Затем создаем еще три таблицы, каждая из которых содержит поле `post_id`, ссылающееся на исходную статью. Эта техника разнесения данных объекта по нескольким таблицам называется *нормализацией*. Среди прочего, нормализованный набор данных характеризуется тем, что каждый элемент данных хранится только в одном месте.

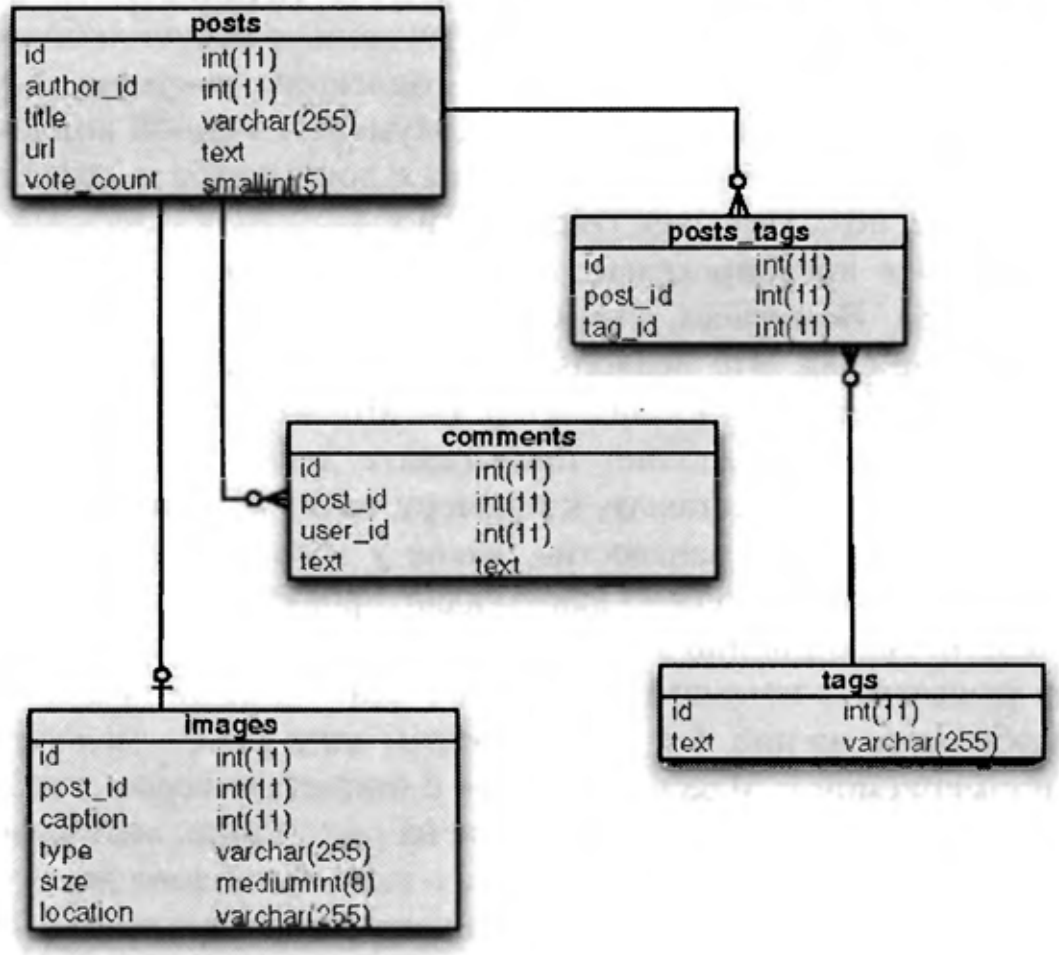


Рис. 1.1. Простая реляционная модель данных, описывающая статьи на социальном новостном сайте

Но за строгую нормализацию приходится платить. И самое главное то, что впоследствии данные нужно как-то собрать. Чтобы показать статью, необходимо выполнить соединение таблиц post и tags. А комментарии либо запрашивать по отдельности, либо тоже включить их в соединение. В конечном итоге ответ на вопрос, нужна ли строгая нормализация, зависит от вида моделируемых данных. Я еще вернусь к этой теме в главе 4. Здесь же важно отметить, что документоориентированная модель позволяет естественно представить данные в агрегированной форме, то есть работать в объектом, как с единым целым: все данные, относящиеся к статье, от комментариев до тегов, хранятся в одном объекте базы данных.

Вероятно, вы обратили внимание, что документы не только позволяют представлять данные со сложной структурой, но и не нуждаются в заранее определенной схеме. В реляционной базе данных строки хранятся в таблице. У каждой таблицы имеется строго определенная схема, описывающая, какие столбцы и типы данных допустимы. Если окажется, что необходимо добавить еще одно поле, то таблицу надо

будет явно изменить. В MongoDB документы группируются в коллекции – контейнеры, не налагающие на данные какую-либо схему. Теоретически у каждого входящего в коллекцию документа может быть своя структура, но на практике документы в одной коллекции похожи друг на друга. Например, у всех документов в коллекции posts имеются поля title, tags, comments и т. д.

Отсутствие предопределенной схемы несет с собой некоторые преимущества. Во-первых, структуру данных определяет код приложения, а не база. Это позволяет ускорить разработку на ранних этапах, когда схема часто изменяется. Во-вторых, и это куда важнее, безсхемная модель позволяет представить данные с переменным набором свойств. Представьте, к примеру, каталог товаров к интернет-магазине. Заранее неизвестно, какие у товара будут атрибуты, поэтому приложение должно как-то адаптироваться к подобной изменчивости. Традиционно в базе данных с фиксированной схемой эта задача решается с помощью паттерна сущность-атрибут-значение[3], как изображено на рис. 1.2. Здесь показан один раздел модели данных, применяемой в Magento, каркасе с открытым кодом для электронной торговли. Обратите внимание на ряд таблиц, отличающихся только в одном поле `value`, которое в каждой таблице имеет свой тип. Такая структура позволяет администратору определять новые типы товаров вместе с их атрибутами, но ценой возрастания сложности. Представьте себе, какой запрос нужно было бы написать в оболочке MySQL, если бы потребовалось выбрать или обновить товар, смоделированный подобным образом; операции соединения таблиц оказались бы необычайно сложными. А при моделировании в виде документа никакого соединения не требуется, и добавлять новые атрибуты можно динамически.

1.2.2. Произвольные запросы

Говорят, что система поддерживает произвольные запросы, если не требуется заранее определять, какие запросы разрешены. Реляционная база данных таким свойством обладает; она честно выполнит любой правильно записанный SQL-запрос, сколько бы условий он ни содержал. Если вы работали только с реляционными СУБД, то, возможно, считаете произвольные запросы чем-то само собой разумеющимся. Однако не все базы данных поддерживают динамические запросы. Например, хранилища ключей и значений можно опраши-

[3] http://en.wikipedia.org/wiki/Entity-attribute-value_model

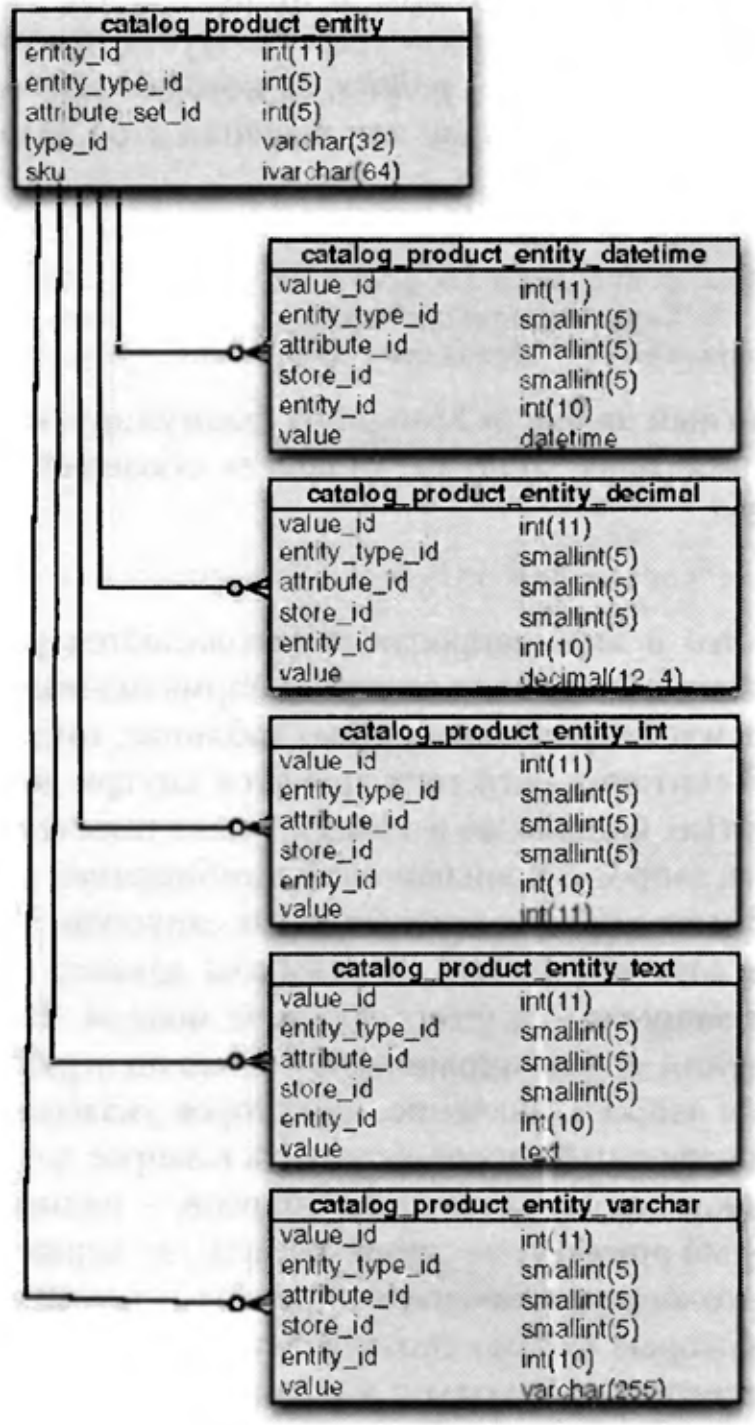


Рис. 1.2. Часть схемы проекта Magento, написанного на PHP и предназначенного для создания сайтов электронной торговли. Эти таблицы позволяют динамически создавать атрибуты товаров.

вать только по одной оси: ключу. Как и многие другие системы, хранилища ключей и значений жертвуют гибкостью запросов в обмен на простоту модели масштабируемости. При проектировании MongoDB в частности ставилась задача по возможности сохранить выразительную мощь запросов, которая считается принципиально важной особенностью реляционных СУБД.

Принцип построения запросов в MongoDB мы рассмотрим на простом примере статей и комментариев. Пусть необходимо найти все статьи, помеченные тегом *politics*, за которые проголосовало более 10 посетителей. SQL-запрос для решения этой задачи выглядел бы так:

```
SELECT * FROM posts
      INNER JOIN posts_tags ON posts.id = posts_tags.post_id
      INNER JOIN tags ON posts_tags.tag_id == tags.id
      WHERE tags.text = 'politics' AND posts.vote_count > 10;
```

Эквивалентный запрос в MongoDB формулируется путем задания документа-образца. Условие «больше» обозначается специальным ключом `$gt`.

```
db.posts.find({'tags': 'politics', 'vote_count': {'$gt': 10}});
```

Отметим, что в этих запросах предполагаются разные модели данных. SQL-запрос опирается на строго нормализованную модель, в которой статьи и теги хранятся в разных таблицах, тогда как в запросе для MongoDB считается, что теги хранятся внутри документа, описывающего статью. Однако же в обоих случаях имеется возможность формулировать запрос с произвольной комбинацией атрибутов, что и составляет смысл понятия произвольных запросов.

Выше уже отмечалось, что не все базы данных поддерживают произвольные запросы – в угоду простоте модели. Так, хранилище ключей и значений можно опрашивать только по первичному ключу. С точки зрения запроса, значение, на которое указывает ключ, непрозрачно. Единственный способ включить в запрос дополнительный атрибут, в данном случае количество голосов, – написать специальный код, который вручную построит записи, где первичным ключом будет счетчик голосов, а значением – список первичных ключей документов, за которые подано соответствующее количество голосов. Если вы примените такой подход к хранилищу ключей и значений, то вас обвинят в трюкачестве. Хотя для небольших наборов данных это, возможно, и будет работать, но, вообще говоря, упаковывание нескольких индексов в физически единственный индекс – мысль не слишком удачная. К тому же, хешированные индексы, применяемые в хранилищах ключей и значений, не поддерживают запросы по диапазону, которые, скорее всего, необходимы для поиска с участием количества голосов.

Если вы раньше работали с реляционными базами данных, в которых произвольные запросы считаются нормой, то возьмите на за-

метку, что MongoDB обеспечивает аналогичную гибкость. Оценивая различные технологии баз данных, имейте в виду, что не все они поддерживают произвольные запросы, и если такая возможность необходима, то MongoDB может оказаться подходящим вариантом. Но одних лишь произвольных запросов недостаточно. Когда объем данных достигает определенного уровня, для обеспечения приемлемой эффективности становятся необходимы индексы. Наличие подходящих индексов увеличивает скорость выполнения запросов и сортировки на несколько порядков; следовательно, любая система, поддерживающая произвольные запросы, должна также поддерживать вторичные индексы.

1.2.3. Вторичные индексы

Понять, что такое индексы в базе данных, проще всего воспользовавшись аналогией: во многих книгах имеются указатели, отображающие ключевые слова на номера страниц. Предположим, что имеется кулинарная книга и требуется найти все рецепты, в которых участвуют груши (быть может, вы собрали хороший урожай груш и не хотите, чтобы они сгнили). Есть долгий способ – просмотреть все рецепты без исключения, отыскивая те, для которых в состав ингредиентов входят груши. Но большинство хозяек предпочтет найти в указателе слово *груши* и получить перечень всех включающих груши рецептов. Индексы базы данных – это структуры, решающие ту же задачу.

В MongoDB вторичные индексы реализованы в виде *B-деревьев*. Такие индексы, применяемые по умолчанию и в большинстве реляционных СУБД, оптимизированы для выполнения самых разнообразных запросов, в том числе сканирования диапазона и запросов с сортировкой результатов. MongoDB допускает несколько вторичных индексов и за счет этого оптимизирует выполнение широкого спектра запросов.

В MongoDB можно создать до 64 индексов над одной коллекцией. При этом поддерживаются все вариации индексов, встречающиеся в РСУБД: по возрастанию, по убыванию, уникальные, с составным ключом и даже геопространственные. Поскольку в MongoDB для индексов применяются те же структуры данных, что и в большинстве РСУБД, рекомендации по управлению индексами схожи. К изучению индексов мы приступим в следующей главе, а, кроме того, поскольку хорошее понимание индексов критически важно для эффективной работы базы данных, я посвящу этой теме всю главу 7.

1.2.4. Репликация

В MongoDB репликация базы данных обеспечивается с помощью так называемой топологии *набора реплик*. Набор реплик распределяется между несколькими машинами для обеспечения избыточности и отработки отказа – автоматического перехода на другой ресурс в случае отказа сервера или сети. Кроме того, репликация применяется для масштабирования операций чтения. Если приложение в основном читает данные из базы, что характерно для веб, то операции чтения можно распределить по нескольким машинам, входящим в кластер наборов реплик.

Рис. 1.3. Автоматическая отработка отказа с помощью набора реплик

Набор реплик состоит из единственного первичного узла и одного или более вторичных. Как в случае репликации главный-подчиненный, с которой вы, возможно, знакомы по другим СУБД, первичный узел набора реплик может выполнять операции чтения и записи, а вторичные узлы – только операции чтения. Уникальная особенность наборов реплик заключается в поддержке автоматического перехода на резервный ресурс в случае отказа: если основной узел выходит из строя, то кластер выбирает какой-то вторичный узел и назначает его первичным. Когда прежний первичный узел восстановится, он станет вторичным. На рис. 1.3 приведена иллюстрация этой процедуры.

Подробно репликация обсуждается в главе 8.

1.2.5. Быстродействие и долговечность

Чтобы разобраться в подходе MongoDB к обеспечению долговечности данных, полезно сначала поговорить о некоторых концепциях.

В мире баз данных существует обратная зависимость между скоростью записи и долговечностью. Под *скоростью записи* понимается совокупный объем операций вставки, обновления и удаления, которые база данных способна обработать в единицу времени. Под *долговечностью* понимается мера уверенности в том, что результат операций записи сохранится.

Пусть, например, вы записали в базу 100 записей по 50 КБ каждая и сразу после этого выключили питание сервера. Можно ли будет восстановить эти записи после перезагрузки машины? Ответ – «может быть», и зависит это как от СУБД, так и от оборудования, на котором она работает. Проблема в том, что запись на магнитный жесткий диск на несколько порядков медленнее, чем запись в оперативную память (ОЗУ). Некоторые базы данных, например memcached, записывают только в ОЗУ, что делает их исключительно быстрыми, но не сохраняющими информацию при выключении питания. С другой стороны, лишь немногие базы данных пишут только на диск, поскольку столь низкая производительность неприемлема. Поэтому проектировщики баз данных обычно идут на компромиссы, чтобы добиться оптимального соотношения быстродействия и долговечности.

В случае MongoDB выбором компромисса управляет пользователь, который задает семантику записи и решает, включать ли запись в журнал. По умолчанию все операции записи следуют семантике *«выстрелил и забыл»*, то есть информация посылается через TCP-сокет, а ответ от СУБД не требуется. Если пользователь хочет получать подтверждение, то может выполнить операцию записи в специальном *безопасном режиме*, поддерживаемом всеми драйверами. Тогда клиент будет ждать от сервера подтверждения, что сведения об операции записи получены без ошибок. Безопасный режим допускает настройку; он позволяет блокировать выполнение программы до тех пор, пока операция записи не будет реплицирована на определенное количество серверов. Для работы в случае, когда количество операций записи велико, а размер каждой операции мал (например, при регистрации переходов по ссылкам и записи в журналы) семантика «выстрелил и забыл» может оказаться идеальной. Для записи более важных данных предпочтителен безопасный режим.

В версии MongoDB 2.0 по умолчанию включено журналирование. В этом режиме каждая операция записи фиксируется в журнале, допускающем только дозапись в конец. Если сервер неожиданно остановится (например, из-за выключения питания), то после его перезапуска журнал позволит MongoDB восстановить согласованное

состояние файлов данных. Это наиболее безопасный способ эксплуатации MongoDB.

Журнал транзакций

Примером компромисса между быстродействием и долговечностью может служить подсистема хранения InnoDB в MySQL. InnoDB – транзакционная подсистема, то есть по определению должна гарантировать долговечность. Для этого любое обновление записывается в два места: в журнал транзакций и в находящийся в памяти пул буферов. Журнал транзакций сбрасывается на диск немедленно, тогда как пул буферов синхронизируется с диском отдельным фоновым потоком с некоторой задержкой. Причина такой двойной записи в том, что, вообще говоря, ввод/вывод с произвольной выборкой гораздо медленнее последовательного. Поскольку запись в основные файлы данных произвольна, то быстрее сначала сохранить изменения в ОЗУ, а синхронизировать их с диском позже. Но для гарантии долговечности записать данные на диск все равно необходимо, и важно, чтобы такая операция записи была последовательной; именно это и обеспечивает журнал транзакций. После аварийного останова InnoDB воспроизводит свой журнал транзакций, внося соответствующие обновления в основные файлы данных. Тем самым достигается приемлемая производительность и одновременно гарантируется высокий уровень долговечности.

Для определенных видов рабочей нагрузки можно отключать режим журналирования на сервере, чтобы повысить производительность. Но при этом не надо забывать, что в случае аварийного останова файлы данных могут оказаться поврежденными. Поэтому, отключая журналирование, следует обеспечить репликацию, предпочтительно в другой центр обработки данных, чтобы при отказе все же сохранилась неиспорченная копия данных.

Репликация и долговечность – обширная тема, ее детальное обсуждение мы отложим до главы 8.

1.2.6. Масштабирование

Простейший способ масштабирования большинства баз данных – модернизация оборудования. Если приложение работает на одном узле, то обычно можно увеличить пропускную способность дисковой подсистемы, добавить память или установить дополнительные процессоры и тем самым расшить узкие места. Методика наращивания оборудования в одном узле называется *вертикальным масштабированием*. Этот способ прост, надежен и рентабелен, но только до определенного момента. Если вы запускаете приложение на виртуализированном оборудовании (например, в кластере Amazon EC2), то рано или поздно обнаруживается, что экземпляра достаточного объема

не существует. А если используется физическое оборудование, то в конце концов наступает момент, когда затраты на приобретение более мощного сервера запредельно высоки.

Тогда имеет смысл задуматься о *горизонтальном масштабировании*. В этом случае вместо наращивания мощности одного узла база данных распределяется по нескольким компьютерам. Поскольку в горизонтально масштабируемой системе можно использовать стандартное оборудование, стоимость хранения и обслуживания всего набора данных можно существенно снизить. К тому же, распределение данных по нескольким машинам ослабляет негативные последствия отказа. Все компьютеры время от времени выходят из строя – это неизбежно. В случае вертикального масштабирования необходимо как-то решать проблему выхода из строя машины, от которой зависит большая часть системы. Всё не так страшно, если существует копия данных на реплицированном сервере, но иногда, тем не менее, имеется какой-то один сервер, отказ которого делает неработоспособной всю систему. А теперь сравните это с отказом одной из машин горизонтально масштабированной системы. Его последствия не столь катастрофичны, так как одна машина представляет лишь малую часть всей системы.

MongoDB изначально спроектирована в расчете на горизонтальное масштабирование. Для этого используется механизм сегментиро-

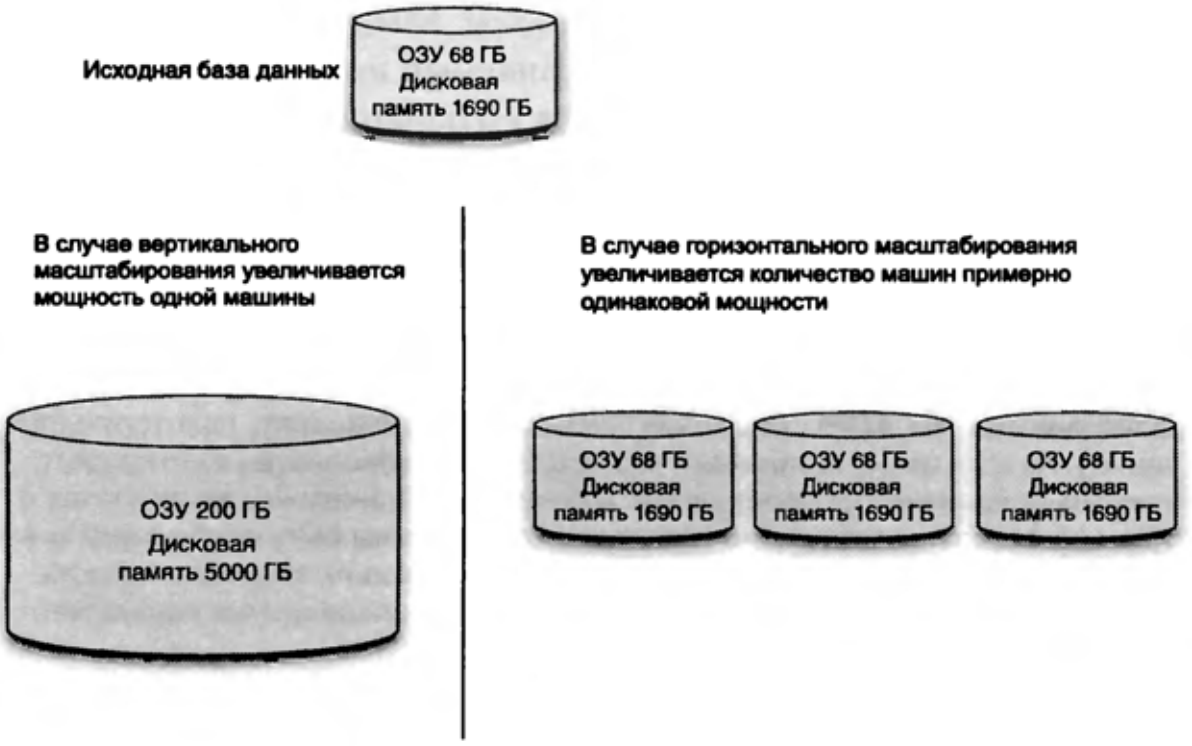


Рис. 1.4. Горизонтальное и вертикальное масштабирование

вания по диапазонам, *автосегментирование* (auto-sharding), который автоматически управляет распределением данных между узлами. Система сама обрабатывает добавление новых сегментных узлов и автоматически переходит на резервный ресурс в случае отказа. Каждый сегмент состоит из набора реплик, занимающего по меньшей мере два узла[4], что гарантирует автоматическое восстановление без точки общего отказа. Это также означает, что прикладной код ничего не должен знать об этой инфраструктуре; приложение взаимодействует с сегментированным кластером так, будто это один узел.

Мы рассмотрели наиболее интересные особенности MongoDB и в главе 2 начнем изучать, как они работают на практике. А пока давайте посмотрим на базу данных с более прагматичной точки зрения. В следующем разделе я расскажу об окружении MongoDB, об инструментальных средствах, поставляемых вместе с сервером, и о нескольких способах загрузки и выгрузки данных.

1.3. Сервер и инструментальные средства MongoDB

СУБД MongoDB написана на языке C++ и активно разрабатывается компанией 10gen. Проект компилируется во всех основных операционных системах, включая Mac OS X, Windows и почти все дистрибутивы Linux. На сайте mongodb.org выложены откомпилированные двоичные файлы для каждой платформы. MongoDB – проект с открытым исходным кодом, распространяемый на условиях лицензии GNU-AGPL. Исходный код находится в открытом бесплатном доступе на сайте GitHub, в него часто включаются дополнения, предлагаемые сообществом. Но руководство проектом в целом осуществляет группа разработки сервера, состоящая из сотрудников компании 10gen, и подавляющее большинство обновлений в системе управления версиями исходит от нее.

О лицензии GNU-AGPL. Лицензия GNU-AGPL может вызвать некоторую путаницу. На практике она означает, что исходный код бесплатен и что приветствуются дополнения от сообщества. А основное ограничение заключается в том, что любые модификации исходного кода должны быть опубликованы в интересах всего сообщества. Для компаний, желающих защитить произведенные модификации сервера, 10gen предлагает специальные коммерческие условия лицензирования.

---

[4] Строго говоря, в каждом наборе реплик должно быть не менее трех узлов, но только на двух хранится копия данных.

Версия MongoDB 1.0 была выпущена в ноябре 2009 года. Новые основные версии выходят примерно раз в три месяца, причем четными числами обозначаются стабильные ветви, а нечетными – разрабатываемые. На момент написания этой книги последней стабильной версией была 2.0[5].

Далее следует обзор компонентов, входящих в комплект поставки MongoDB, сопровождаемый общим описанием инструментальных средств и языковых драйверов для разработки приложений, пользующихся этой СУБД.

1.3.1. Сервер

Исполняемый файл сервера базы данных называется `mongod` (`mongodb.exe` в Windows). Процесс `mongod` взаимодействует с клиентами по специальному протоколу, получая команды через сетевой сокет. Все используемые им файлы по умолчанию хранятся в каталоге /data/db[6].

Сервер `mongod` может работать в нескольких режимах, самый употребительный – член набора реплик. Поскольку репликация рекомендуется, то обычно конфигурируется набор, состоящий из двух реплик и процесса-арбитра, расположенного на третьем сервере[7]. Если MongoDB работает в режиме автосегментирования, то процессы `mongod` в каждом сегменте конфигурируются как наборы реплик, а «сбоку» находятся специальные серверы метаданных, называемые *конфигурационными серверами*. Для отправки запросов конкретным сегментам используется также отдельный маршрутный сервер `mongos`.

Настройка процесса `mongod` проста по сравнению с другими СУБД, например MySQL. Конечно, можно определить номера портов и пути к каталогам данных, но для настройки собственно базы данных параметров не так много. Оптимизация базы данных, которая в большинстве реляционных СУБД подразумевает манипуляции с кучей параметров, управляющих распределением памяти и другими аспектами работы сервера, превратилась в нечто сродни черной магии. Но при проектировании MongoDB было решено, что с управлением памятью операционная система справится лучше, чем администратор базы данных или разработчик приложения. Поэтому файлы

[5] Всегда следует работать с последней стабильной выпускной версией, например, 2.0.1.

[6] c:\data\db в Windows.

[7] Процессы-арбитры потребляют немного ресурсов и могут запускаться, например, на сервере приложений.

данных проецируются на виртуальную память системы с помощью системного вызова mmap(). Тем самым забота об управлении памятью передается ядру ОС. Ниже я еще вернусь к вызову mmap(), а пока отмечу лишь, что почти полное отсутствие конфигурационных параметров – продуманная стратегия, а не ошибка.

1.3.2. JavaScript-оболочка

Командная оболочка MongoDB – это инструмент для администрирования базы и манипулирования данными с помощью команд на языке JavaScript. Исполняемый файл mongo загружает оболочку и устанавливает соединение с указанным процессом mongod. Оболочка по своим возможностям сравнима с оболочкой MySQL с тем отличием, что язык SQL не используется. Вместо этого команды по большей части записываются в виде выражений языка JavaScript. Вот, например, как выбрать базу данных и вставить простой документ в коллекцию users:

```
> use mongodb-in-action
> db.users.insert({name: "Kyle"})
```

Первая команда, определяющая, какая база данных будет использоваться, знакома любому пользователю MySQL. Вторая команда – выражение JavaScript для вставки простого документа. Чтобы увидеть результат вставки, можно выполнить простой запрос:

```
> db.users.find()
{ _id: ObjectId("4ba667b0a90578631c9caea0"), name: "Kyle" }
```

Метод find возвращает вставленный документ вместе с добавленным идентификатором объекта. В любом документе должен быть первичный ключ, хранящийся в поле _id. Разрешается задавать _id самостоятельно при условии, что вы гарантируете его уникальность. Если поле _id вообще опустить, то MongoDB присвоит объекту идентификатор автоматически.

Помимо вставки и выборки данных, оболочка позволяет выполнять административные команды, например, показывать текущую выполняемую сервером операцию, проверять состояние репликации на вторичный узел и настраивать сегментирование коллекции. Как вы убедитесь, оболочка MongoDB – действительно мощный инструмент, который полезно освоить.

Но основная работа с MongoDB осуществляется в приложении, написанном на том или ином языке программирования. Чтобы по-

нять, как это делается, следует сказать несколько слов о языковых драйверах MongoDB.

1.3.3. Языковые драйверы

Если сама идея языкового драйвера вызывает у вас кошмарные ассоциации с низкоуровневыми драйверами устройств, расслабьтесь. Пользоваться драйверами MongoDB просто. Разработчики постарались предоставить API, согласующийся с идиомами конкретного языка, обеспечив в то же время относительное единообразие интерфейса. Так, все драйверы реализуют общий набор методов для вставки документа в коллекцию, однако представление самого документа обычно выглядит естественно для данного языка. Например, в Ruby это будет хеш, в Python – словарь, а в Java, где нет аналогичного примитива на уровне языка, документ представляется с помощью специального класса конструктора документов, который реализует интерфейс `LinkedHashMap`.

Поскольку драйверы предоставляют развитый адаптированный к языку интерфейс с базой данных, то для написания приложения не нужен слой абстрагирования поверх самого драйвера. Этим разработка принципиально отличается от РСУБД, для которых почти всегда присутствует библиотека, являющаяся посредником между реляционной моделью данных и объектно-ориентированной моделью, характерной для большинства современных языков программирования. Но даже в отсутствие тяжеловесной системы объектно-реляционного отображения многие разработчики предпочитают использовать тонкую обертку вокруг драйверов для обработки ассоциаций, проверки данных и контроля типов[8].

На момент написания этой книги компания 10gen официально поддерживала драйверы для языков C, C++, C#, Erlang, Haskell, Java, Perl, PHP, Python, Scala и Ruby – и этот перечень постоянно расширяется. Если вам нужна поддержка для какого-то другого языка, то очень может быть, что уже имеется драйвер, написанный сообществом. А если такового не окажется, то на сайте mongodb.org опубликованы спецификации для разработки драйверов. Поскольку все официально поддерживаемые драйверы активно применяются в производственных системах и распространяются на условиях лицензии Apache, то авторы новых драйверов не испытывают недостатка в хо-

[8] Из популярных на данный момент оберток стоит упомянуть Morphia для Java, Doctrine для PHP и Mongo-Mapper для Ruby.

роших примерах. Начиная с главы 3, я буду описывать, как работают драйверы и как ими пользоваться при написании программ.

1.3.4. Командные утилиты

В комплект поставки MongoDB входят несколько командных утилит:

- `mongodump` *и* `mongorestore` – стандартные утилиты резервного копирования и восстановления базы данных. `mongodump` сохраняет данные во внутреннем формате BSON и потому применяется главным образом для создания резервных копий. Достоинством этой программы является возможность использования для снятия горячих резервных копий, из которых впоследствии можно восстановить базу данных с помощью утилиты `mongorestore`.
- `mongoexport` *и* `mongoimport` – эти утилиты экспортируют и импортируют файлы в форматах JSON, CSV и TSV, поэтому они полезны, когда требуется представить данные в одном из широко распространенных форматов. `mongoimport` также удобна для начальной загрузки больших наборов данных, хотя стоит отметить, что перед импортом часто бывает желательно подправить модель данных, так чтобы можно было в полной мере задействовать все преимущества MongoDB. В таких случаях проще импортировать данные с помощью специально написанной программы на одном из поддерживаемых языков.
- `mongosniff` – анализатор протокола для просмотра команд, посылаемых серверу базы данных. По сути дела транслирует передаваемые команды в формате BSON в понятные человеку команды оболочки.
- `mongostat` – аналог `iostat`; постоянно опрашивает MongoDB и операционную систему для выдачи полезной статистики, в том числе количества операций (вставки, выборки, обновления, удаления и т. п.) в секунду, объема выделенной виртуальной памяти и числа подключений к серверу.

Оставшиеся утилиты, `bsondump` и `monfiles`, будут рассмотрены ниже.

1.4. Почему именно MongoDB?

Я уже приводил несколько причин, почему MongoDB может оказаться удачным выбором для вашего проекта. Сейчас я сформулирую их более отчетливо, но сначала остановлюсь на общих целях, поставленных при проектировании MongoDB. По словам создателей, MongoDB проектировалась так, чтобы объединить лучшие черты хранилищ ключей и значений и реляционных баз данных. Хранилища ключей и значений благодаря своей простоте работают чрезвычайно быстро и относительно легко масштабируются. Масштабировать реляционные базы труднее, по крайней мере по горизонтали, зато они предлагают развитую модель данных и мощный язык запросов. Если считать, что MongoDB находится посредине между этими двумя крайностями, то получается СУБД, которая легко масштабируется, позволяет хранить сложные структуры данных и предоставляет развитые механизмы формулирования запросов.

Если говорить о сценариях использования, то MongoDB хорошо подходит в качестве основного хранилища данных для веб-приложений, аналитических приложений, протоколирования и для любых приложений, которым необходим кэш среднего класса. Кроме того, поскольку MongoDB позволяет хранить безсхемные данные, то она удобна, когда структура данных заранее неизвестна.

Всё это довольно смелые заявления. Чтобы подкрепить их, мы дадим широкий обзор типов используемых в настоящее время СУБД и сравним их с MongoDB. Затем я расскажу о некоторых специфичных для MongoDB сценариях и приведу примеры реального использования. Наконец, мы обсудим некоторые важные практические соображения касательно применения MongoDB.

1.4.1. Сравнение MongoDB с другими СУБД

Количество имеющихся на рынке СУБД растет лавинообразно, и сравнивать одну с другой становится все труднее. К счастью, большинство из них попадает в одну из нескольких категорий. В последующих разделах я опишу простые и более изощренные хранилища ключей и значений, реляционные и документные базы данных и сопоставлю их с MongoDB.

Таблица 1.1. Семейства СУБД

	Примеры	Модель данных	Модель масштабируемости	Применение
Простые хранилища ключей и значений	Memcached	Ключ-значение, где значением может быть произвольный двоичный объект.	Разные. Memcached может масштабироваться на несколько узлов, представляя всю доступную память в виде единого монолитного хранилища данных.	Кэширование, веб-приложения.
Развитые хранилища ключей и значений	Cassandra, проект Voldemort, Riak	Разные. В Cassandra используется структура ключ-значение, называемая *столбцом*. В Voldemort хранятся двоичные объекты.	Согласованное в конечном счете распределение по узлам, обеспечивающее высокую доступность и простую отработку отказа.	Вертикальные приложения с высокой пропускной способностью (веб-каналы активности, очереди сообщений). Кэширование. Веб-приложения.
Реляционные СУБД	Oracle, MySQL, PostgreSQL	Таблицы.	Вертикальное масштабирование. Ограниченная поддержка кластеризации и ручного сегментирования.	Системы, нуждающиеся в транзакциях (банковские и финансовые приложения) или в языке SQL. Нормализованная модель данных.

Простые хранилища ключей и значений

Назначение следует из названия – индексирование значение по ключу. Типичное применение – кэширование. Пусть, например, требуется кэшировать HTML-страницы, генерируемые приложением. В этом случае ключом может быть URL-адрес, а значением – сама HTML-страница. Отметим, что с точки зрения хранилища значение представляет собой непрозрачный массив байтов. Нет ни предопределенной схемы, как в реляционных базах данных, ни какого-либо

понятия о типе данных. Это налагает естественное ограничение на операции, выполняемые хранилищами ключей и значений: можно только поместить новое значение, а затем получить или удалить его по ключу. Столь простые системы обычно работают очень быстро и хорошо масштабируются.

Из всех простых хранилищ ключей и значений наиболее широкое распространение получило memcached (произносится *мэм-кэш-ди*). Memcached хранит все данные в оперативной памяти, то есть приносит долговечность в жертву быстродействию. Хранилище распределенное – узлы memcached, размещенные на нескольких серверах, могут работать как единое хранилище данных, устраняя тем самым сложности, связанные с отслеживанием состояния кэша на разных машинах.

По сравнению с MongoDB простые хранилища ключей и значений типа memcached часто демонстрируют более высокую скорость операций чтения и записи. Но, в отличие от MongoDB, они редко используются в качестве основного хранилища данных. Они больше подходят на роль придатков, например, слоя кэширования поверх традиционной СУБД или простого слоя хранения для таких недолговечных структур, как очереди задач.

Развитые хранилища ключей и значений

Простую модель ключей и значений можно развить, сделав ее пригодной для обработки более сложных схем чтения-записи или для предоставления улучшенной модели данных. Примером такого *развитого хранилища ключей и значений* может служить система Dynamo, разработанная компанией Amazon и описанная в широко известном документе «Dynamo: Amazon's Highly Available Key-value Store». Задача Dynamo – предложить базу данных, которая могла бы продолжать работу в условиях сетевых сбоев, отключения питания в центре обработки данных и других подобных отказов. Таким образом, требуется, чтобы система при любых обстоятельствах обеспечивала чтение и запись, а это значит, что данные должны автоматически реплицироваться на несколько узлов. Если какой-то узел выходит из строя, то обслуживание пользователя системы, быть может, клиента Amazon, положившего что-то в корзину, не будет прервано. Dynamo располагает средствами разрешения неизбежных конфликтов, возникающих, когда системе разрешено записывать одни и те же данные на несколько узлов. В то же время Dynamo легко масштабируется. Поскольку в системе нет главного узла – все узлы

равноправны, – то разобраться в ее топологии несложно, и добавлять новые узлы не составляет труда. Хотя Dynamo – закрытая система, идеи, положенные в ее основу, вдохновили разработчиков на создание многих других систем, попадающих в категорию NoSQL, в том числе Cassandra, проект Voldemort и Riak.

Если посмотреть, кто разрабатывал развитые хранилища ключей и значений и как они применяются на практике, то станет ясно, что это направление процветает. Взять, к примеру, систему Cassandra, в которой реализованы многие свойства масштабируемости Dynamo и одновременно предлагается колоночная модель данных по образцу Google BigTable. Cassandra – проект с открытым кодом, созданный компанией Facebook для поиска по «входящим». Система горизонтально масштабируется, обеспечивая индексирование свыше 50 ТБ данных в ящиках «входящие» и поиск по ключевым словам и получателям. Данные индексируются по идентификатору пользователя, а каждая запись состоит из массива поисковых терминов для поиска по ключевым словам и массива идентификаторов получателей для поиска по получателям[9].

Подобные развитые хранилища ключей и значений разрабатываются такими крупными интернет-компаниями, как Amazon, Google и Facebook, для управления отдельными частями систем, хранящих невероятно большие объемы данных. Иными словами, хранилище предназначено для работы с относительно автономной подсистемой, для которой требуется очень много внешней памяти и высокая доступность. Благодаря отсутствию главного узла такие системы легко масштабируются путем добавления новых узлов. Они поддерживают модель согласованности в конечном счете (eventual consistency); это означает, что результат операции чтения не обязательно отражает последнее обновление в результате операции записи. Но в обмен на более слабые гарантии согласованности пользователь получает возможность читать данные в случае отказа любого узла.

По сравнению с такими системами MongoDB характеризуется строгой согласованностью, наличием одного главного узла (в каждом сегменте), более развитой моделью данных и поддержкой вторичных индексов. Последние два свойства неразрывно связаны – если система допускает моделирование нескольких предметных областей, что необходимо в любом сколько-нибудь полном веб-приложении, то она должна поддерживать запросы ко всей модели данных, а, значит, без вторичных индексов не обойтись.

---

[9] http://mng.bz/5321.

Благодаря развитой модели данных MongoDB можно считать более общим решением задачи построения больших масштабируемых веб-приложений. Архитектуру масштабирования MongoDB иногда критикуют за то, что она не повторяет идеи Dynamo. Но ведь это разные решения для разных задач масштабирования. В основу механизма автосегментирования в MongoDB положены идеи, воплощенные в хранилищах данных PNUTS от Yahoo!'s PNUTS и BigTable от Google. Любой, кто читал официальные документы, описывающие эти хранилища, увидит, что подход MongoDB к масштабированию уже был реализован и притом успешно.

Реляционные СУБД

О реляционных СУБД уже много было сказано во введении к этой книге, поэтому краткости ради я лишь отмечу, что у них общего с MongoDB и в чем они различаются. И MongoDB, и MySQL[10] способны представить развитую модель данных, хотя в MySQL используются таблицы с фиксированной схемой, а в MongoDB – документы без схемы. И MySQL, и MongoDB поддерживают индексы на основе B-деревьев, поэтому пользователи, привыкшие работать с индексами в MySQL, могут ожидать аналогичного поведения от MongoDB. MySQL поддерживает операцию соединения и транзакции, поэтому если необходимо использовать SQL или требуются транзакции, то следует обратиться к MySQL или другой РСУБД. Но при этом документной модели MongoDB зачастую достаточно для представления объектов без соединения. А любое обновление применяется к отдельным документам атомарно, то есть мы имеем подмножество функциональности, традиционно обеспечиваемой транзакциями. И MongoDB, и MySQL поддерживают репликацию. Что касается масштабируемости, то MongoDB изначально проектировалась с учетом горизонтального масштабирования с автоматическим сегментированием и отработкой отказов. В MySQL сегментирование необходимо организовывать вручную, поэтому, принимая во внимание сложность этой задачи, системы на основе MySQL чаще масштабируются по вертикали.

Документные базы данных

Немногие СУБД позиционируют себя как документные. На момент написания этой книги единственной хорошо известной документной базой данных, помимо MongoDB, была CouchDB от Apache. Модель документа в CouchDB похожа, но данные хранятся в виде

[10] Здесь название «MySQL» используется в общем смысле, так как описываемые возможности имеются в большинстве реляционных СУБД.

обычного текста в формате JSON, тогда как в MongoDB применяется двоичный формат BSON. Как и MongoDB, CouchDB поддерживает вторичные индексы; различие в том, что для определения индекса в CouchDB необходимо написать функции распределения-редукции (map-reduce), а это сложнее, чем декларативный синтаксис, используемый в MySQL и в MongoDB. И масштабирование в этих системах организовано по-разному. В CouchDB нет сегментирования по машинам, каждый узел CouchDB является полной репликой любого другого узла.

1.4.2. Сценарии использования и примеры реального развертывания

Будем честны. СУБД выбирается не только за ее возможности. Необходимо знать, какие реальные компании успешно использовали ее в своем бизнесе. Ниже я сделаю обзор типичных сценариев использования MongoDB и приведу примеры развертывания в производственной среде[11].

Веб-приложения

MongoDB подходит на роль основного хранилища данных для веб-приложений. Даже в простом веб-приложении имеется много моделей данных для представления пользователей, сеансов, специфических данных, закачек и разрешений, не говоря уже о собственно предметной области. Все это прекрасно ложится как на табличную структуру, предлагаемую реляционными СУБД, так и на модель коллекций и документов в MongoDB. А поскольку с помощью документа можно представить сложные структуры данных, то количество коллекций обычно оказывается меньше, чем количество таблиц, необходимых для моделирования тех же данных в полностью нормализованной реляционной модели. Кроме того, с помощью динамических запросов и вторичных индексов можно без труда реализовать большинство запросов, знакомых пользователям SQL. Наконец, по мере роста приложения MongoDB предлагает простой механизм масштабирования.

На практике MongoDB подтвердила способность справляться со всеми аспектами приложения – от основных задач предметной области до таких вспомогательных аспектов, как протоколирование и аналитика в реальном времени. Примером может служить новостной сайт *The*

[11] Актуальный перечень промышленных внедрений MongoDB приведен по адресу http://mng.bz/z2CH.

Business Insider (TBE), который применяет MongoDB в качестве основного хранилища данных с января 2008 года. TBE обслуживает более миллиона уникальных просмотров страниц в день. Интересно, что помимо обработки основного содержимого сайта (статьи, комментарии, пользователи и т. д.), MongoDB также обрабатывает и хранит аналитические данные в режиме реального времени. Эти данные используются TBE для динамической генерации карт распределения интереса, на которых отражается кликабельность (CTR) различных новостей. Объем данных на сайте пока не настолько велик, чтобы требовалось сегментирование, но наборы реплик тем не менее используются для гарантированного перехода на резервный ресурс в случае отказа.

Гибкая разработка

Как бы вы относиться к движению за гибкую разработку, трудно отрицать, что приложение желательно создавать быстро. Немало коллективов, в частности Shutterfly и The New York Times, выбрали MongoDB отчасти за то, что она позволяет разрабатывать приложения быстрее, чем при использовании реляционных СУБД. Одна из причин, лежащая на поверхности, заключается в отсутствии фиксированной схемы, что позволяет не тратить время на разработку схемы, ее распространение и внесение в нее изменений.

Кроме того, меньше времени уходит на втискивание реляционного представления данных в объектно-ориентированную модель, на борьбу с капризами системы ORM и на оптимизацию сгенерированного ей кода. Поэтому MongoDB часто используется в проектах с коротким циклом разработки, над которыми трудятся гибкие команды среднего размера.

Аналитика и протоколирование

Я уже упоминал, что MongoDB хорошо приспособлена для аналитики и протоколирования, и число приложений, в которых эта СУБД используется для таких целей, быстро растет. Нередко солидные компании с прочной репутацией начинают вторжение в мир MongoDB со специальных приложений, относящихся в аналитике. В качестве примеров приведу GitHub, Disqus, Justin.tv и Gilt Groupe.

Для аналитических приложений MongoDB привлекательна своим быстродействием и двумя важными особенностями: атомарным обновлением и ограниченными коллекциями (capped collections). Атомарное обновление позволяет клиенту эффективно увеличивать счетчики и помещать новые значения в массив. Ограниченные кол-

лекции, часто применяемые для протоколирования, имеют фиксированный размер, то есть старые элементы автоматически удаляются в порядке FIFO. Хранение протоколируемых данных в базе, а не в файловой системе упрощает администрирование и существенно расширяет возможности запросов. Теперь вместо использования `grep` или специализированной утилиты поиска в журнале пользователь может написать запрос на знакомом языке запросов MongoDB.

Кэширование

Благодаря модели данных, позволяющей представлять объекты целиком, и повышенной скорости выполнения запросов MongoDB можно использовать вместо более традиционной связки MySQL и memcached. Например, вышеупомянутый сайт TBE смог отказаться от memcached в пользу непосредственной генерации страниц из данных, хранящихся в MongoDB.

Переменные схемы

Взгляните на следующий пример[12]:

```
curl https://stream.twitter.com/1/statuses/sample.json -umongodb:secret
| mongoimport -c tweets
```

Здесь мы скачиваем небольшой фрагмент потока Twitter и вставляем его непосредственно в коллекцию MongoDB. Так как поток состоит из документов в формате JSON, нет необходимости преобразовывать данные перед записью в базу. Утилита `mongoimport` сама преобразует данные в формат BSON. Это означает, что каждый «твит» сохраняется в исходном виде как отдельный документ в коллекции. Поэтому можно сразу же начать работу с данными: опрашивать, индексировать или агрегировать с помощью функций распределения и редукции. И объявлять структуру данных заранее не нужно.

Если приложение зависит от JSON API, то система, позволяющая так просто работать с JSON, бесценна. Когда структура данных заранее неизвестна, отсутствие в MongoDB фиксированной схемы может существенно упростить модель данных.

1.5. Советы и ограничения

Но какими бы привлекательными ни были все эти возможности, не стоит забывать о принятых в системе компромиссах и ограничениях.

[12] Идея заимствована из статьи по адресу http://mng.bz/52XI. Чтобы выполнить этот код, подставьте вместо строки `-umongodb:secret` свои имя пользователя и пароль в Twitter.

О некоторых ограничениях нужно знать перед тем, как принимать решение о разработке реального приложения на основе MongoDB. По большей части они связаны с тем, что в MongoDB используются проецируемые на память файлы.

Прежде всего, MongoDB обычно следует устанавливать на 64-разрядные компьютеры. 32-разрядные системы могут адресовать не более 4 ГБ памяти. Учитывая, что, как правило, половину этого объема занимают операционная система и исполняемые процессы, для проецирования файлов остается только 2 ГБ. Поэтому даже при небольшом количестве индексов размер данных ограничен примерно 1,5 ГБ. Но для большинства производственных систем требуется больше памяти, отсюда и необходимость 64-разрядной архитектуры[13].

Второе следствие проецирования на виртуальную память заключается в том, что память для данных выделяется автоматически, по мере необходимости. Это усложняет использование базы данных в условиях разделяемого хостинга. Как и любую СУБД, MongoDB, вообще говоря, лучше запускать на выделенном сервере.

Наконец, при работе с MongoDB важно включать репликацию, особенно если не включен режим журналирования. Из-за использования проецируемых на память файлов любой нештатный останов `mongod` при отключенном журналировании может привести к повреждению данных. Поэтому так важно иметь резервную копию в виде реплики, пригодной для отработки отказа. Это рекомендация в равной мере относится к любой базе данных (было бы неразумно пренебречь ей при развертывании любого сколько-нибудь серьезного приложения на основе MySQL), но особенно она важна для MongoDB, работающей без журналирования.

1.6. Резюме

Мы рассмотрели много вопросов. Подведем итоги. MongoDB – документо-ориентированная СУБД с открытым исходным кодом. Будучи спроектирована с учетом требований к данным и масштабируемости, предъявляемым современными веб-приложениями, MongoDB поддерживает динамические запросы и вторичные индексы, быстрое атомарное обновление и сложные способы агрегирования, репликацию с автоматической отработкой отказов и сегментирование, обеспечивающее масштабирование по горизонтали.

[13] 64-разрядная архитектура теоретически позволяет адресовать до 16 экзабайтов, что для всех практических целей можно считать неограниченной памятью.

Фраза получилась длинной. Но если вы дочитали до этого места, то уже понимаете, что стоит за этими словами. Наверное, вам не терпится приступить к кодированию. В конце концов, рассуждать о возможностях базы данных – одно, а использовать их на практике – совсем другое. Именно последним мы и будем заниматься в двух следующих главах. Сначала мы познакомимся с JavaScript-оболочкой MongoDB, которая весьма полезна для интерактивной работы с базой. А затем, в главе 3, мы начнем экспериментировать с драйвером и напишем простое приложение на языке Ruby с использованием MongoDB.

ГЛАВА 2. MongoDB сквозь призму JavaScript-оболочки

В этой главе:

- Операции CRUD в оболочке MongoDB.
- Построение индексов и команда `explain()`.
- Получение справки.

Мы закончили предыдущую главу словами о практической работе с MongoDB. Если вы готовы к экспериментам, то приступим. В этой главе мы с помощью оболочки MongoDB изучим основные концепции этой СУБД на ряде примеров. Вы научитесь создавать, читать, обновлять и удалять (create, read, update, delete – CRUD) документы и по ходу дела познакомитесь с языком запросов MongoDB. Кроме того, мы немного поговорим об индексах базы данных и об их применении для оптимизации запросов. И закончим главу рассмотрением некоторых простых административных команд, после чего я покажу, как можно получать справку по ходу работы. Можете считать эту главу одновременно более подробным изложением идей, упомянутых во введении, и практическим занятием по наиболее употребительным способам использования оболочки MongoDB.

Для тех, кто раньше никогда не работал с оболочкой MongoDB, скажу, что она предоставляет всё, что принято ожидать от такого рода программ: позволяет просматривать и изменять данные и администрировать сам сервер. А отличается от других подобных инструментов языком запросов. Вы взаимодействуете с сервером не на стандартизованном языке запросов типа SQL, а на языке программирования JavaScript, дополненным простым API. Если вы не знаете JavaScript, не волнуйтесь – для работы с оболочкой необходимо лишь поверх-

ностное знакомство с ним, а все примеры сопровождаются подробными пояснениями.

Чтобы извлечь из этой главы максимальную пользу, нужно проработать примеры, а для этого необходимо установить на свою машину MongoDB. Инструкции по установке приведены в приложении А.

2.1. Первое знакомство с оболочкой MongoDB

JavaScript-оболочка MongoDB позволяет манипулировать данными и почувствовать, что представляют собой документы, коллекции и язык запросов. Считайте приведенные ниже примеры практическим введением в MongoDB.

Мы начнем с запуска оболочки. Затем посмотрим, как в JavaScript представляются документы, и научимся вставлять документы в коллекцию. Для проверки результата вставки мы попрактикуемся в опросе коллекций. Потом перейдем к обновлению. И напоследок покажем, как очищать и удалять коллекции.

2.1.1. Запуск оболочки

Если вы выполнили инструкции, приведенные в приложении А, то сейчас на ваш компьютер установлен работоспособный экземпляр MongoDB. Убедитесь, что процесс `mongod` работает, а затем запустите оболочку MongoDB:

```
./mongo
```

Если все нормально, то на экране появится приглашение, как на рис. 2.1. В заголовке отображается версия MongoDB, с которой вы работаете, и сведения о текущей базе данных.

Если вы хоть немного знакомы с языком JavaScript, то можете сразу же приступить к вводу кода и исследованию возможностей оболочки. В противном случае читайте дальше, и вы узнаете, как вставить свой первый документ.

2.1.2. Вставка и выборка

Если при запуске не указана база данных, то по умолчанию оболочка выбирает базу `test`. Но чтобы во всех последующих упражнениях оставаться в одном и том же пространстве имен, переключимся на базу данных `tutorial`:

Рис. 2.1. JavaScript-оболочка MongoDB сразу после запуска

```
> use tutorial
switched to db tutorial
```

Появляется сообщение, подтверждающее переключение на другую базу.

О создании баз данных и коллекций. Возможно, у вас возник вопрос, как можно переключиться на базу данных `tutorial`, не создав ее явно. На самом деле, создавать базу данных необязательно. Базы данных и коллекции создаются при вставке первого документа. Такое поведение согласуется с принятым в MongoDB динамическим подходом к данным – точно так же, как структуру документов не нужно определять заранее, так и отдельные коллекции и базы можно создавать прямо во время выполнения. Это упрощает и ускоряет процесс разработки и позволяет динамически назначать пространства имен, что очень часто бывает полезно. Тем не менее, если вас пугает возможность непреднамеренного создания базы данных или коллекции, то в большинстве драйверов имеется *строгий режим*, предотвращающий такого рода случайные ошибки.

Но пора уже создать первый документ. Так как мы работаем с JavaScript-оболочкой, то документы представляются в формате JSON (JavaScript Object Notation). Простейший документ, описывающий одного пользователя, мог бы выглядеть так:

```
{username: "jones"}
```

Этот документ содержит одну пару ключ-значение для хранения имени пользователя Jones. Чтобы сохранить этот документ, нужно указать коллекцию. Коллекция `users` вполне подойдет:

```
> db.users.insert({username: "smith"})
```

После ввода этого предложения может наблюдаться небольшая задержка. В это время на диске создается база данных `tutorial` и коллекция `users`. Задержка обусловлена созданием начальных файлов для того и другого.

Если вставка завершилась успешно, то ваш первый документ сохранен. Чтобы убедиться в этом, можно выполнить простой запрос:

```
> db.users.find()
```

Ответ будет выглядеть примерно так:

```
{ _id : ObjectId("4bf9bec50e32f82523389314"), username : "smith" }
```

Обратите внимание, что в документ добавлено поле _id. Можете считать его значение первичным ключом документа. В любом документе MongoDB обязательно присутствует поле _id, и если в момент создания его значение не задано, то MongoDB генерирует специальный идентификатор объекта. На вашей консоли отобразится не тот же идентификатор, что в примере выше, но он гарантированно будет уникален среди всех значений _id в данной коллекции – единственное непререкаемое требование к этому полю.

Я еще вернусь к вопросу об идентификаторах объектов в следующей главе. А пока продолжим – добавим в коллекцию еще одного пользователя:

```
> db.users.save({username: "jones"})
```

Теперь в коллекции должно быть два документа. Проверим это, выполнив команду count:

```
> db.users.count()
2
```

Имея в коллекции более одного документа, мы можем попробовать более сложные запросы. Как и раньше, можно запросить все документы в коллекции:

```
> db.users.find()
{ _id : ObjectId("4bf9bec50e32f82523389314"), username : "smith" }
{ _id : ObjectId("4bf9bec90e32f82523389315"), username : "jones" }
```

Но можно также передать методу find простой селектор запроса. *Селектором запроса* называется документ, с которым сравниваются все документы в коллекции. Чтобы найти все документы, в которых поле username равно jones, нужно задать такой селектор:

```
> db.users.find({username: "jones"})
{ _id : ObjectId("4bf9bec90e32f82523389315"), username : "jones" }
```

Селектор запроса {username: "jones"} возвращает все документы, в которых имя пользователя содержит строку jones – документ-образец буквально сравнивается со всеми хранящимися в коллекции документами.

Пока что мы познакомились с простейшими операциями создания и чтения данных. Теперь посмотрим, как данные обновляются.

2.1.3. Обновление документов

Для обновления нужно задать по меньшей мере два аргумента. Первый определяет, какие документы обновлять, второй – как следует модифицировать отобранные документы. Существует два способа модификации; в этом разделе мы рассмотрим *направленную модификацию* (targeted modification) – одну из наиболее интересных и уникальных особенностей MongoDB.

Предположим, что пользователь `smith` решил указать свою страну проживания. Сделать это можно следующим образом:

```
> db.users.update({username: "smith"}, {$set: {country: "Canada"}})
```

Здесь мы просим MongoDB найти документ, в котором поле `username` равно `smith`, и записать в свойство `country` значение `Canada`. Если теперь запросить этот документ, то мы увидим, что он обновился:

```
> db.users.find({username: "smith"})
{ "_id" : ObjectId("4bf9ec440e32f82523389316"),
"country" : "Canada", username : "smith" }
```

Если впоследствии пользователь решит убрать страну из своего профиля, то столь же просто сможет удалить значение с помощью оператора `$unset`:

```
> db.users.update({username: "smith"}, {$unset: {country: 1}})
```

Разовьем этот пример. Данные представляются в виде документов, которые, как мы видели в главе 1, могут содержать сложные структуры данных. Предположим, что, помимо профиля, пользователь желает хранить списки своих любимых вещей. Представление такого документа могло бы выглядеть так:

```
{  username: "smith",
   favorites: {
      cities: ["Chicago", "Cheyenne"],
      movies: ["Casablanca", "For a Few Dollars More", "The Sting"]
   }
}
```

Ключ `favorites` указывает на объект, содержащий два других ключа, указывающих на списки любимых городов и фильмов. Можете ли вы придумать, как с помощью того, что вам уже известно, моди-

фицировать исходный документ о пользователе `smith` так, чтобы он принял такой вид? На ум должен сразу прийти оператор `$set`. Отметим, что в этом случае мы практически полностью переписываем документ, и `$set` ничего не имеет против:

```
> db.users.update( {username: "smith"},
{  $set: {favorites:
   {
      cities: ["Chicago", "Cheyenne"],
      movies: ["Casablanca", "The Sting"]
   }
}
})
```

Произведем аналогичную модификацию для пользователя `jones`, но на этот раз добавим два любимых фильма:

```
db.users.update( {username: "jones"},
   {"$set": {favorites:
      {
         movies: ["Casablanca", "Rocky"]
      }
   }
})
```

Теперь запросите все документы из коллекции `users`, чтобы убедиться, что оба обновления выполнены:

```
> db.users.find()
```

Имея под рукой два примера документов, вы теперь можете раскрыть всю мощь языка запросов MongoDB. В частности, в такой ситуации особенно полезна возможность заглядывать во вложенные объекты и сравнивать документ-образец с элементами массива. Для такого запроса понадобится использовать специальную точечную нотацию. Пусть требуется найти всех пользователей, которым нравится фильм *Casablanca*. Вот как выглядит такой запрос:

```
> db.users.find({"favorites.movies": "Casablanca"})
```

Точка между `favorites` и `movies` означает, что нужно найти ключ `favorites`, который указывает на вложенный объект с ключом `movies`, а затем сравнить значение этого вложенного ключа с указанным в запросе. Этот запрос вернет оба документа. Чтобы усложнить задачу, предположим, что вам заранее известно, что всякий пользователь, которому нравится фильм *Casablanca*, также любит фильм *The Maltese Falcon* (Мальтийский сокол), и вы хотели бы обновить базу данных, отразив этот факт. Как записать такой запрос на обновление в MongoDB?

Можно было бы снова воспользоваться оператором `$set`, но это означало бы, что надо переписать и отправить серверу весь массив фильмов. А так как нам всего-то и надо что добавить в список один элемент, то лучше использовать оператор `$push` или `$addToSet`. Оба оператора добавляют в массив элемент, но второй гарантирует уникальность, то есть предотвращает появление дубликатов. Вот как выглядит искомый запрос:

```
db.users.update( {"favorites.movies": "Casablanca"},
   {$addToSet: {"favorites.movies": "The Maltese Falcon"} },
         false,
         true )
```

Тут почти всё понятно. Первый аргумент, селектор запроса, говорит, что нужно искать пользователей, для которых в списке `movies` есть фильм *Casablanca*. Второй аргумент говорит, что нужно добавить в этот список фильм *The Maltese Falcon* с помощью оператора `$addToSet`. Третий аргумент, `false`, мы пока проигнорируем. А четвертый, `true`, означает, что речь идет о множественном обновлении. По умолчанию операция обновления в MongoDB применяется только к первому документу, отобранному селектором запроса. Если требуется обновить все подходящие документы, то об этом нужно сказать явно. Поскольку мы хотим обновить оба документа – `smith` и `jones`, то необходимо множественное обновление.

Более подробно мы будем рассматривать операции обновления ниже, но перед тем как двигаться дальше, всё же выполните эти примеры.

2.1.4. Удаление данных

Итак, вы теперь знаете, как создавать, читать и обновлять документы в оболочке MongoDB. Последнюю, самую простую, операцию – удаление – мы оставили на закуску.

Если не задавать никаких параметров, то операция удаления `remove` удалит из коллекции все документы. Так, чтобы избавиться от коллекции `foo`, нужно выполнить такую команду:

```
> db.foo.remove()
```

Часто бывает необходимо удалить только часть документов из коллекции; для этого методу `remove()` следует передать селектор запроса. Вот как можно удалить всех пользователей, которые любят город Шайенн (Cheyenne):

```
> db.users.remove({"favorites.cities": "Cheyenne"})
```

Отметим, что операция `remove()` не уничтожает саму коллекцию, а лишь удаляет из нее документы. Можете считать ее аналогом команд SQL DELETE и TRUNCATE.

Чтобы уничтожить коллекцию вместе со всеми построенными над ней индексами, используйте метод `drop()`:

```
> db.users.drop()
```

Создание, чтение, обновление и удаление – основные операции в любой базе данных. Если вы читали внимательно, то теперь можете самостоятельно поэкспериментировать с операциями CRUD в MongoDB. В следующем разделе мы продолжим изучение операций выборки, обновления и удаления, познакомившись с вторичными индексами.

2.2. Создание индексов и применение их в запросах

Индексы обычно создаются для повышения скорости выполнения запросов. К счастью, оболочка MongoDB позволяет создавать индексы безо всякого труда. Если ранее вам не доводилось работать с индексами базы данных, то из этого раздела вам станет ясно, зачем они нужны. А если у вас уже есть опыт работы с индексами, то вы узнаете, как просто они создаются в MongoDB и как можно профилировать выполнение запросов с помощью метода `explain()`.

2.2.1. Создание большой коллекции

Индексировать коллекцию имеет смысл, только когда в ней много документов. Поэтому добавим 200 000 простых документов в коллекцию `numbers`. Поскольку оболочка MongoDB одновременно является интерпретатором JavaScript, то сделать это несложно:

```
for(i=0; i<200000; i++) {
   db.numbers.save({num: i});
}
```

200 000 документов – это немало, так что не удивляйтесь, если на выполнение команды уйдет несколько секунд. По завершении вы можете с помощью парочки запросов убедиться, что все документы на месте:

```
> db.numbers.count()
200000

> db.numbers.find()
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830a"), "num" : 0 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830b"), "num" : 1 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830c"), "num" : 2 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830d"), "num" : 3 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830e"), "num" : 4 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830f"), "num" : 5 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8310"), "num" : 6 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8311"), "num" : 7 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8312"), "num" : 8 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8313"), "num" : 9 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8314"), "num" : 10 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8315"), "num" : 11 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8316"), "num" : 12 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8317"), "num" : 13 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8318"), "num" : 14 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8319"), "num" : 15 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831a"), "num" : 16 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831b"), "num" : 17 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831c"), "num" : 18 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831d"), "num" : 19 }
has more
```

Команда `count()` показывает, что вставлено 200 000 документов. Второй запрос выводит первые 20 результатов. Чтобы показать следующую порцию, выполните команду `it`:

```
> it
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831e"), "num" : 20 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831f"), "num" : 21 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8320"), "num" : 22 }
...
```

Команда `it` просит оболочку вернуть следующий результирующий набор[1].

Имея набор документов приличного размера, попробуем выполнить несколько запросов. Вас уже не удивит, что для поиска документа по его атрибуту `num` достаточно такого простого запроса:

```
> db.numbers.find({num: 500})
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac84fe"), "num" : 500 }
```

[1] Вас, наверное, интересует, что происходит за кулисами. При выполнении любого запроса создается курсор, который позволяет обойти результирующий набор. Оболочка скрывает этот факт, потому прямо сейчас обсуждать его преждевременно. Но если вам не терпится узнать побольше о курсорах и их странностях, то загляните в главы 3 и 4.

Более интересны запросы по диапазону, для которых предназначены специальные операторы `$gt` и `$lt`. Мы уже встречались с ними в главе 1 (`gt` означает *greater than* [больше], а `lt` – *less than* [меньше]). Вот как запросить документы, для которых значение `num` больше 199 995:

```
> db.numbers.find( {num: {"$gt": 199995 }} )
{ "_id" : ObjectId("4bfbf1dedba1aa7c30afcade"), "num" : 199996 }
{ "_id" : ObjectId("4bfbf1dedba1aa7c30afcadf"), "num" : 199997 }
{ "_id" : ObjectId("4bfbf1dedba1aa7c30afcae0"), "num" : 199998 }
{ "_id" : ObjectId("4bfbf1dedba1aa7c30afcae1"), "num" : 199999 }
```

Можно употреблять эти операторы вместе, чтобы задать верхнюю и нижнюю границу:

```
> db.numbers.find( {num: {"$gt": 20, "$lt": 25 }} )
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831f"), "num" : 21 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8320"), "num" : 22 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8321"), "num" : 23 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8322"), "num" : 24 }
```

Как видите, с помощью простого JSON-документа можно сформулировать сложный запрос по диапазону, как на языке SQL. `$gt` и `$lt` – всего два из многочисленных ключевых слов, используемых в языке запросов MongoDB; в последующих главах мы встретим много других примеров.

Разумеется, от таких запросов мало толку, если они выполняются неэффективно. В следующем разделе мы впервые задумаемся об эффективности и начнем изучать имеющиеся в MongoDB средства индексирования.

2.2.2. Индексирование и команда explain()

Если вы долго работали с реляционными базами данных, то, наверное, знакомы с командой SQL `EXPLAIN`. Она описывает путь выполнения запроса и позволяет выявить медленные операции, показывая, какие индексы были использованы. В MongoDB тоже имеется вариант `EXPLAIN` с аналогичной функциональностью. Чтобы понять, как эта команда работает, применим ее к одному из предыдущих запросов. Выполните такую команду:

```
> db.numbers.find( {num: {"$gt": 199995 }} ).explain()
```

Результат должен выглядеть примерно так, как показано в листинге ниже.

Листинг 2.1. Типичная распечатка `explain()` для запроса без использования индексов

```
{
   "cursor" : "BasicCursor",
   "nscanned" : 200000,
   "nscannedObjects" : 200000,
   "n" : 4,
   "millis" : 171,
   "nYields" : 0,
   "nChunkSkips" : 0,
   "isMultiKey" : false,
   "indexOnly" : false,
   "indexBounds" : { }
}
```

Изучение распечатки, выданной `explain()` показывает, что для возврата всего четырех результатов (`n`) серверу пришлось просканировать всю коллекцию из 200 000 (`nscanned`) документов. Тип курсора `BasicCursor` подтверждает, что для формирования результирующего набора индексы не использовались. Столь большая разница между количеством просмотренных и возвращенных документов свидетельствует о неэффективности выполнения запроса. В реальном приложении, когда и коллекция больше, и сами документы объемнее, время выполнения запроса окажется существенно больше 171 миллисекунды, как в данном примере.

Этой коллекции явно недостает индекса. Построить индекс по ключу `num` можно с помощью метода `ensureIndex()`. Введите такую команду:

```
> db.numbers.ensureIndex({num: 1})
```

Как и для любой другой операции MongoDB, например выборки или обновления, методу `ensureIndex()` передается документ, определяющий, по каким ключам следует индексировать. В данном случае документ `{num: 1}` говорит, что над коллекцией `numbers` нужно построить индекс по ключу `num` в порядке возрастания.

Убедиться в том, что индекс действительно построен, позволит метод `getIndexes()`:

```
> db.numbers.getIndexes()
[
   {
      "name" : "_id_",
      "ns" : "tutorial.numbers",
      "key" : {
         "_id" : 1
```

```
        }
    },
    {
        "_id" : ObjectId("4bfc646b2f95a56b5581efd3"),
        "ns" : "tutorial.numbers",
        "key" : {
            "num" : 1
        },
        "name" : "num_1"
    }
]
```

Теперь над этой коллекцией построено два индекса. Первый – по стандартному ключу `_id` – автоматически строится для любой коллекции; второй – по ключу `num` – мы только что создали сами.

Если сейчас выполнить тот же запрос с помощью метода `explain()`, то будет заметна ощутимая разница во времени выполнения, что отражено в листинге ниже.

Листинг 2.2. Распечатка `explain()` для запроса с использованием индекса

```
> db.numbers.find({num: {"$gt": 199995 }}).explain()
{
    "cursor" : "BtreeCursor num_1",
    "indexBounds" : [
        [
            {
                "num" : 199995
            },
            {
                "num" : 1.7976931348623157e+308
            }
        ]
    ],
    "nscanned" : 5,
    "nscannedObjects" : 4,
    "n" : 4,
    "millis" : 0
}
```

Теперь, когда используется индекс по ключу `num`, запрос просматривает только пять документов. В результате общее время выполнения сократилось со 171 мс до менее чем 1 мс.

Если этот пример заинтриговал вас, то подождите до главы 7, которая посвящена индексированию и оптимизации запросов. А пока мы продолжим и рассмотрим основные административные команды, применяемые для получения информации об экземпляре MongoDB.

Заодно познакомимся с тем, как получить в оболочке справку, которая помогает овладеть всеми тонкостями различных команд.

2.3. Основы администрирования

Я обещал в этой главе дать введение в MongoDB сквозь призму JavaScript-оболочки. Вы уже узнали об основах манипулирования данными и индексирования. А сейчас я познакомлю вас с некоторыми способами получения информации о процессе `mongod`. Например, интересно было бы узнать, сколько места занимают различные коллекции или сколько индексов построено над конкретной коллекцией. Описанные ниже команды помогут вам диагностировать проблемы, связанные с производительностью, и следить за своими данными.

Мы также поговорим о командном интерфейсе к MongoDB. Большинство операций с экземпляром MongoDB, отличных от CRUD, – от опроса состояния сервера до проверки целостности файлов данных – реализованы с помощью команд базы данных. Я объясню, что это за команды в контексте MongoDB, и покажу, как просто ими пользоваться. Наконец, никогда не помешает знать, куда обратиться за помощью. Отвечая на этот вопрос, я покажу те места в оболочке, где можно получить справку, полезную для дальнейшего исследования MongoDB.

2.3.1. Получение информации о базе данных

Часто бывает нужно знать, какие базы данных и коллекции существуют в данной системе. К счастью, оболочка MongoDB предоставляет целый ряд команд, дополненных синтаксической глазурью, которые позволяют получить такого рода информацию.

Команда `show dbs` выводит список всех баз данных в системе:

```
> show dbs
admin
local
test
tutorial
```

Команда `show collections` выводит список всех коллекций в текущей базе данных[2]. Если текущей всё еще является база данных `tutorial`, то эта команда выведет коллекции, созданные выше:

[2] Для той же цели подойдет более лаконичная команда `show tables`

```
> show collections
numbers
system.indexes
users
```

Пока что вам незнакома коллекция `system.indexes`. Это специальная коллекция, существующая в любой базе данных. Каждый ее элемент соответствует одному из построенных индексов. Эту коллекцию можно опрашивать напрямую, но более наглядный результат получается при использовании метода `getIndexes()`, с которым мы уже встречались.

Для получения низкоуровневой информации о базах данных и коллекциях полезен метод `stats()`. Вызвав его для какого-либо объекта базы данных, мы получим такой результат:

```
> db.stats()
{
   "collections" : 4,
   "objects" : 200012,
   "dataSize" : 7200832,
   "storageSize" : 21258496,
   "numExtents" : 11,
   "indexes" : 3,
   "indexSize" : 27992064,
   "ok" : 1
}
```

Команда `stats()` применима и к отдельной коллекции:

```
> db.numbers.stats()
{
   "ns" : "tutorial.numbers",
   "count" : 200000,
   "size" : 7200000,
   "storageSize" : 21250304,
   "numExtents" : 8,
   "nindexes" : 2,
   "lastExtentSize" : 10066176,
   "paddingFactor" : 1,
   "flags" : 1,
   "totalIndexSize" : 27983872,
   "indexSizes" : {
      "_id_" : 21307392,
      "num_1" : 6676480
   },
   "ok" : 1
}
```

Некоторые представленные в этих документах значения полезны только в особо сложных ситуациях во время отладки. Но как мини-

мум вы можете узнать, сколько места занимает указанная коллекция и построенные над ней индексы.

2.3.2. Как работают команды

Набор операций с MongoDB – помимо операций вставки, обновления, удаления и выборки, рассмотренных в этой главе выше, – называется *командами базы данных*. Вообще говоря, команды базы данных служат для администрирования, как, например, только что описанная команда `stats()`, но могут также применяться для управления такими базовыми средствами MongoDB, как распределение и редукция (map-reduce).

Но вне зависимости от функциональности у всех команд базы данных есть общая черта: они реализованы как запросы к специальной виртуальной коллекции `$cmd`. Чтобы понять, что под этим понимается, рассмотрим небольшой пример. Вспомните, как вызывалась команда `stats()`:

```
> db.stats()
```

Метод `stats()` – это обертка вокруг метода вызова команды оболочки. Попробуйте ввести следующую эквивалентную команду:

```
> db.runCommand( {dbstats: 1} )
```

Получится точно такой же результат. Обратите внимание, что команда определяется документом `{dbstats: 1}`. Вообще любую имеющуюся команду можно выполнить, передав соответствующий документ методу `runCommand`. Вот, например, как выполнить команду `stats` для коллекции:

```
> db.runCommand( {collstats: 'numbers'} )
```

Результат тот же, что мы видели ранее.

Но чтобы добраться до сути команд базы данных, надо понять, как на самом деле работает метод `runCommand()`. Выяснить это нетрудно, потому что оболочка MongoDB распечатывает код любого метода, если опустить круглые скобки после его имени. Иными словами, вместо команды

```
> db.runCommand()
```

нужно выполнить ту же команду без скобок, и вы увидите реализацию:

```
> db.runCommand
   function (obj) {
```

```
    if (typeof obj == "string") {
        var n = {};
        n[obj] = 1;
        obj = n;
    }
    return this.getCollection("$cmd").findOne(obj);
}
```

Последняя строчка – не что иное, как запрос к коллекции `$cmd`. Строгое определение звучит так: команда базы данных – это запрос к специальной коллекции `$cmd`, в котором селектор определяет саму команду. Вот и всё. А как выполнить команду `stats` для коллекции вручную? Да вот так:

```
db.$cmd.findOne( {collstats: 'numbers'} );
```

Пользоваться вспомогательным методом `runCommand` проще, но всегда полезно знать, что происходит под капотом.

2.4. Получение справки

Сейчас вам уже должно быть очевидно, насколько полезна оболочка MongoDB в качестве платформы для экспериментов с данными и средства администрирования базы. Но раз вы будете проводить в оболочке много времени, то хорошо было бы знать, как получить справку.

Прежде всего, следует обратиться к встроенным командам. Команда `db.help()` выводит список наиболее употребительных методов работы с объектами базы данных. Аналогичный список методов для работы с коллекциями выводит команда `db.foo.help()`.

Поддерживается также автозавершение. Введите первые символы имени любого метода и дважды нажмите клавишу Tab. Будет выведен список всех методов с подходящими именами. Вот, например, список методов работы с коллекциями, имена которых начинаются словом `get`:

```
> db.foo.get
db.foo.getCollection(  db.foo.getIndexSpecs(  db.foo.getName(
db.foo.getDB(          db.foo.getIndexes(     db.foo.getShardVersion(
db.foo.getFullName(    db.foo.getIndices(
db.foo.getIndexKeys(   db.foo.getMongo(
```

Более амбициозные и знакомые с языком JavaScript пользователи могут изучить реализацию любого метода. Пусть, например, вам хочется узнать, как работает метод `save()`. Можно, конечно, покопаться

в исходном коде MongoDB, но есть более простой способ. Достаточно ввести имя метода без круглых скобок. Обычно метод `save()` вызывается следующим образом:

```
> db.numbers.save({num: 123123123});
```

А чтобы ознакомиться с его реализацией, нужно выполнить такую команду:

```
> db.numbers.save
function (obj) {
   if (obj == null || typeof obj == "undefined") {
      throw "can't save a null";
   }
   if (typeof obj._id == "undefined") {
      obj._id = new ObjectId;
      return this.insert(obj);
   } else {
      return this.update({_id:obj._id}, obj, true);
   }
}
```

Внимательно изучив определение этой функции, вы увидите, что `save()` – не более чем обертка вокруг методов `insert()` и `update()`. Если в сохраняемом объекте нет поля `_id`, то оно добавляется и вызывается метод `insert()`, в противном случае производится обновление существующего объекта.

Этот способ изучения методов часто бывает полезен, имейте его в виду, когда продолжите исследовать оболочку MongoDB.

2.5. Резюме

Мы познакомились с документной моделью данных на практике и продемонстрировали типичные операции MongoDB. Вы научились создавать индексы и с помощью команды `explain()` убедились, что индексы повышают производительность выполнения запросов. Вы также умеете получать информацию о базах данных и коллекциях, имеющихся в системе, и знаете всё об «умной» коллекции `$cmd`. А если потребуется помощь, то у вас найдется парочка трюков, которые помогут выбраться на правильную дорогу.

Многому можно научиться, работая в оболочке MongoDB, но ничто не сможет заменить опыт создания реального приложения. Поэтому в следующей главе мы покинем площадку для беззаботных игр с данными и отправимся в реальный цех, где с данными работают. Вы

узнаете, как работают драйверы, а затем, применяя драйвер для Ruby, напишете простое приложение, столкнув MongoDB с настоящими «живыми» данными.

ГЛАВА 3.
Разработка программ для MongoDB

В этой главе:

- Введение в MongoDB API на примере языка Ruby.
- Как работают драйверы.
- Формат BSON и сетевой протокол MongoDB.
- Создание полного приложения.

Пора переходить к делу. Конечно, экспериментируя с оболочкой MongoDB, можно многому научиться, но чтобы по-настоящему оценить эту СУБД, нужно написать для нее какое-нибудь приложение. То есть перейти к программированию и познакомиться с драйверами MongoDB. Выше уже отмечалось, что компания 10gen официально поддерживает на условиях лицензии Apache драйверы для всех популярных языков программирования. В этой книге мы будем работать с драйвером для Ruby, но описываемые принципы универсальны и применимы к другим драйверам. Для особо любознательных читателей в приложении D описываются API драйверов для PHP, Java и C++.

Изучение программирования для MongoDB мы разобьем на три этапа. Сначала вы установите драйвер MongoDB для Ruby и познакомитесь с программированием основных операций CRUD. На этом мы долго не задержимся, потому что API похож на тот, что используется в оболочке. Затем мы копнем глубже и объясним, как устроен интерфейс между драйвером и MongoDB. Не опускаясь на слишком низкий уровень, мы все же покажем, что происходит внутри драйвера. Наконец, мы разработаем простое приложение на Ruby для мо-

ниторинга сайта Twitter. Поработав с реальным набором данных, вы станете лучше понимать, как MongoDB ведет себя в жизни. И в этом последнем разделе мы заодно заложим основы для углубленного изучения предмета в части 2.

Не знаете Ruby?

Ruby – популярный скриптовый язык, довольно простой для восприятия. В приведенных ниже примерах не используются сложные конструкции, поэтому они будут понятны даже программистам, не знакомым с Ruby. Те идиомы Ruby, которые сложно понять без подготовки, объясняются в тексте. Если вы хотите потратить немного времени на беглое знакомство с Ruby, то рекомендую начать с официального 20-минутного пособия по адресу http://mng.bz/THR3.

3.1. MongoDB сквозь призму Ruby

Обычно, когда речь заходит о драйверах, на ум приходят низкоуровневые манипуляции с битами и головоломные интерфейсы. Но, к счастью, языковые драйверы MongoDB не имеют ничего общего с этой картиной; они спроектированы как интуитивно понятные, учитывающие особенности объемлющего языка API, так чтобы в большинстве приложений драйвер MongoDB мог использоваться в качестве единственного интерфейса к базе данных. API драйверов для разных языков устроены единообразно, а, значит, разработчик может легко переходить с одного языка на другой. Предполагается, что программист сможет комфортно и продуктивно работать с любым драйвером MongoDB, не утруждая себя изучением низкоуровневых деталей реализации.

Для начала вы установите драйвер MongoDB для Ruby, подключитесь к базе данных и научитесь выполнять основные операции CRUD. На этом фундаменте мы в конце главы напишем приложение.

3.1.1. Установка и подключение к базе

Драйвер MongoDB для Ruby можно установить с помощью системы управления пакетами RubyGems.

Примечание. Если в вашей системе Ruby не установлен, то по адресу http://www.ruby-lang.org/en/downloads/ вы найдете подробные инструкции по установке. Понадобится также менеджер пакетов RubyGems. О том, как его установить, см. http://docs.rubygems.org/read/chapter/3.

```
gem install mongo
```

Эта команда установит gem-пакеты `mongo` и `bson`[1]. В процессе установки будут напечатаны примерно такие сообщения (номера версий, скорее всего, будут более поздними):

```
Successfully installed bson-1.4.0
Successfully installed mongo-1.4.0
2 gems installed
Installing ri documentation for bson-1.4.0...
Installing ri documentation for mongo-1.4.0...
Installing RDoc documentation for bson-1.4.0...
Installing RDoc documentation for mongo-1.4.0...
```

Начнем с подключения к серверу MongoDB. Прежде всего, убедитесь, что процесс `mongod` запущен. Затем создайте файл `connect.rb` и поместите в него такой код:

```
require 'rubygems'
require 'mongo'

@con = Mongo::Connection.new
@db = @con['tutorial']
@users = @db['users']
```

Первые два предложения `require` обеспечивают загрузку драйвера. А в следующих строчках создается объект соединения, в переменную `@db` записывается ссылка на базу данных `tutorial`, а в переменную `@users` – ссылка на коллекцию `users`. Сохраните и запустите этот файл:

```
$ruby connect.rb
```

Если не возникнет никаких исключений, значит, соединение с MongoDB из Ruby успешно установлено. Не бог весть какое достижение, но подключение к базе данных – первый шаг работы с MongoDB в любом языке. Далее мы воспользуемся этим подключением для вставки документов.

3.1.2. Вставка документов на Ruby

Все драйверы MongoDB спроектированы с учетом конструкций, наиболее естественных для объемлющего языка. В случае JavaScript очевидным выбором являются JSON-объекты, потому что именно в

[1] BSON – это двоичный формат на основе JSON, который применяется в MongoDB для представления документов (подробно рассматривается в следующем разделе). Gem-пакет `bson` позволяет сериализовывать Ruby-объекты в формат BSON и обратно.

формате JSON представлена структура данных документа. А в Ruby больше всего для этой цели подходит хеш. Хеш Ruby мало чем отличается от JSON-объекта; пожалуй, самое заметное различие состоит в том, что в JSON ключи и значения разделяются запятой, а в Ruby – стрелкой (=>)[2].

Если вы прорабатываете примеры, то добавляйте новый код в файл `connect.rb`. Альтернатива – воспользоваться интерактивным интерпретатором Ruby, `irb`. Если запустить `irb`, затребовав в командной строке файл `connect.rb`, то будет установлено соединение с сервером и инициализированы объекты базы данных и коллекции. Затем можно вводить код на Ruby и сразу же получать результаты. Например:

```
$ irb -r connect.rb
irb(main):001:0> id = @users.save({"lastname" => "knuth"})
=> BSON::ObjectId('4c2cfea0238d3b915a000004')
irb(main):002:0> @users.find_one({"_id" => id})
=> {"_id"=>BSON::ObjectId('4c2cfea0238d3b915a000004'), "lastname"=>"knuth"}
```

Сконструируем несколько документов для коллекции *users*, точнее, создадим два документа, описывающие пользователей smith и jones. Каждый документ, представленный в виде хеша Ruby, присваивается переменной:

```
smith = {"last_name" => "smith", "age" => 30}
jones = {"last_name" => "jones", "age" => 40}
```

Чтобы сохранить документы, мы передаем их методу коллекции `insert`. Вызов `insert` возвращает уникальный идентификатор, который мы сохраним в переменной (он нам еще понадобится):

```
smith_id = @users.insert(smith)
jones_id = @users.insert(jones)
```

С помощью простых запросов можно убедиться, что документы действительно сохранены. Как обычно, каждому документу присваивается идентификатор объекта, который хранится в поле `_id`. Поэтому для выборки из коллекции пользователей можно использовать метод `find_one`:

```
@users.find_one({"_id" => smith_id})
@users.find_one({"_id" => jones_id})
```

Если этот код выполняется в `irb`, то возвращаемые значения печатаются на экране. Если же он исполняется из файла скрипта, то для вывода на экран нужно добавить в начало обращение к методу Ruby `p`:

```
p @users.find_one({"_id" => smith_id})
```

[2] В версии Ruby 1.9 можно использовать в качестве разделителя двоеточие, но в интересах обратной совместимости мы будем придерживаться нотации со стрелкой.

Итак, мы успешно вставили два документа из программы на Ruby. Теперь рассмотрим запросы более пристально.

3.1.3. Запросы и курсоры

Выше мы использовали для выборки одиночного результаты метод драйвера find_one. Это оказалось несложно, потому что find_one скрывает некоторые детали выполнения запросов к MongoDB. А сейчас мы рассмотрим стандартный метод find, чтобы понять, как все происходит на самом деле. Вот два примера операций поиска в наборе данных:

```
@users.find({"last_name" => "smith"})
@users.find({"age" => {"$gt" => 20}})
```

Понятно, что первый запрос ищет всех пользователей, для которых фамилия `last_name` равна `smith`, а второй – пользователей, для которых возраст `age` больше 30. Введите второй запрос в `irb`:

```
irb(main):008:0> @users.find({"age" => {"$gt" => 30}})
=> <#Mongo::Cursor:0x10109e118 ns="tutorial.users"
@selector={"age" => "$gt" => 30}}>
```

Сразу бросается в глаза, что метод find возвращает не результирующий набор, а объект курсора. Как и в большинстве СУБД, курсоры возвращают результирующий набор порциями – чтобы повысить эффективность итеративного обхода. Представьте, что в коллекции `users` миллион документов, удовлетворяющих запросу. Не будь курсора, все эти документы пришлось бы вернуть разом. То есть скопировать весь гигантский результирующий набор в память, передать его по сети, а затем десериализовать на стороне клиента. Понадобилось бы очень много ресурсов – и совершенно впустую. Чтобы этого избежать, запрос порождает курсор, который затем используется для получения обозримых порций результирующего набора. Разумеется, пользователю всё это незаметно; когда у курсора запрашиваются дополнительные результаты, драйвер сам обращается к MongoDB, чтобы заполнить буфер курсора.

Более подробно мы поговорим о курсорах в следующем разделе. А сейчас вернемся к примеру и получим результаты запроса с оператором `$gt`:

```
cursor = @users.find({"age" => {"$gt" => 20}})

cursor.each do |doc|
   puts doc["last_name"]
end
```

Используемый здесь итератор Ruby `each` передает каждый результат блоку кода, в котором на консоль выводится атрибут `last_name`. Для тех, кто не знаком с итераторами Ruby, ниже приведен эквивалентный код, в меньшей степени зависящий от особенностей языка:

```
cursor = @users.find({"age" => {"$gt" => 20}})

while doc = cursor.next
   puts doc["last_name"]
end
```

В этом случае используется простой цикл `while`, который обходит курсор и на каждой итерации присваивает локальной переменной `doc` результат, возвращаемый методом курсора `next`.

Тот факт, что нам вообще приходится задумываться о курсорах, может вызвать удивление, потому что в примерах работы с оболочкой из предыдущей главы ничего такого не было. Однако же оболочка пользуется курсорами точно так же, как драйвер; разница только в том, что при вызове `find()` оболочка автоматически выводит первые 20 результатов. Для получения оставшихся результатов можно продолжить итерирование вручную с помощью команды `it`.

3.1.4. Обновление и удаление

Напомним, что для обновления нужно задать по меньшей мере два аргумента: селектор запроса и документ, описывающий обновление. Вот простой пример для драйвера Ruby:

```
@users.update({"last_name" => "smith"}, {"$set" => {"city" => "Chicago"}})
```

Здесь мы ищем первого пользователя, для которого поле `last_name` равно `smith`, и, если такой найден, то записываем в поле `city` значение `Chicago`. Для обновления применяется оператор `$set`.

По умолчанию MongoDB применяет обновление только к одному документу. В данном случае имеется несколько пользователей с фамилией `smith`, но тем не менее обновлен будет лишь один документ. Чтобы применить обновление к конкретному «Смиту», необходимо добавить в селектор запроса дополнительные условия. Если же вы действительно хотите обновить все документы с фамилией `smith`, то нужно задать режим *множественного обновления*. Для этого в третьем аргументе метода `update` передается флаг `:multi => true`:

```
@users.update({"last_name" => "smith"},
   {"$set" => {"city" => "New York"}}, :multi => true)
```

Удалять данные гораздо проще. Для этого служит метод `remove`, который принимает необязательный селектор запроса, позволяющий

удалить только определенные документы. Если же селектор не задан, то из коллекции удаляются все документы. Ниже мы удаляем документы о пользователях, для которых атрибут `age` больше или равен 40:

```
@users.remove({"age" => {"$gte" => 40}})
```

При вызове без аргументов метод `remove` удалит все оставшиеся документы:

```
@users.remove
```

В предыдущей главе отмечалось, что `remove` не уничтожает саму коллекцию. Чтобы удалить коллекцию вместе со всеми построенными над ней индексами, вызовите метод `drop_collection`:

```
connection = Mongo::Connection.new
db = connection['tutorial']
db.drop_collection('users')
```

3.1.5. Команды базы данных

В предыдущей главе мы познакомились с самыми важными командами базы данных – обеими командами `stats`. А сейчас покажем, как выполнять команды с помощью драйвера на примере `listDatabases`. Это одна из многих команд, которые должны выполняться в контексте базы данных `admin`, которая трактуется особым образом, если включена аутентификация. Более подробно об аутентификации и базе данных `admin` см. главу 10.

Первым делом создадим объект Ruby, ссылающийся на базу данных `admin`, а затем передадим спецификацию запроса методу `command`:

```
@admin_db = @con['admin']
@admin_db.command({"listDatabases" => 1})
```

В ответ мы получаем хеш Ruby со списком всех существующих баз данных, в котором для каждой базы указано занимаемое ей место на диске:

```
{
   "databases" => [
      {
         "name" => "tutorial",
         "sizeOnDisk" => 218103808,
         "empty" => false
      },
      {
         "name" => "admin",
         "sizeOnDisk" => 1,
         "empty" => true
```

```
        },
        {
            "name" => "local",
            "sizeOnDisk" => 1,
            "empty" => true
        }
    ],
    "totalSize" => 218103808,
    "ok" => true
}
```

После того как вы привыкнете к представлению документов в виде хешей Ruby, переход от API оболочки к драйверам покажется безболезненным. Но ничего страшного, если вы пока чувствуете неуверенность при работе с MongoDB из Ruby; в разделе 3.3 мы еще попрактикуемся. А пока немного отвлечемся на рассказ о том, как работают драйверы MongoDB. Это прольет свет на некоторые особенности дизайна MongoDB и поможет лучше подготовиться к эффективному использованию драйверов.

3.2. Как работают драйверы

Сейчас было бы естественно поинтересоваться тем, что происходит за кулисами, когда вы отправляете команду из драйвера или оболочки MongoDB. В этом разделе мы отдернем занавес и посмотрим, как драйвер сериализует данные и передает их серверу.

Любой драйвер MongoDB выполняет три основных функции. Во-первых, он генерирует идентификаторы объектов MongoDB, то есть те самые значения, которые по умолчанию записываются в поле `_id` документа. Во-вторых, драйвер преобразует языково-зависимое представление документа во внутренний двоичный формат BSON и обратно. В примерах выше драйвер преобразует хеши Ruby в формат BSON, а затем производит обратное преобразование из формата BSON, возвращаемого сервером, в хеши Ruby.

Последняя функция драйвера – обмениваться с сервером данными через TCP-сокет по сетевому протоколу MongoDB. Детали этого протокола выходят за рамки настоящей книги. Но вопрос о порядке взаимодействия через сокеты и, в частности, о том, нужно ли ожидать ответа в случае операции записи, важен, и мы займемся им в этом разделе.

3.2.1. Генерация идентификатора объекта

В любом документе MongoDB должен быть первичный ключ, уникальный в пределах коллекции. Этот ключ хранится в поле докумен-

та _id. Разработчик вправе присваивать значение ключу самостоятельно, но, если он этого не сделает, то MongoDB будет использовать идентификатор объекта. Перед тем как отправлять документ серверу, драйвер проверяет, присутствует ли в нем поле _id. Если нет, то генерируется идентификатор объекта и записывается в _id.

Поскольку идентификаторы объектов в MongoDB глобально уникальны, то можно присваивать документу идентификатор на стороне клиента, не опасаясь получить дубликат. Вы, конечно, уже встречались ранее с идентификаторами объектов, но, возможно, не обратили внимания, что они состоят из 12 байтов, структурированных, как показано на рис. 3.1.

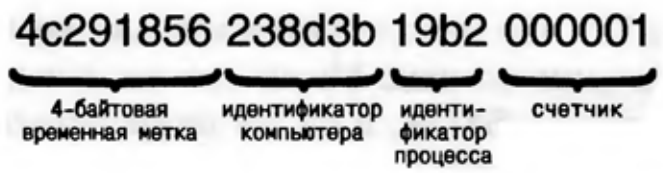


Рис. 3.1. Формат идентификатора объекта в MongoDB

В старших четырех байтах находится стандартная временная метка Unix, содержащая количество секунд от начала «эпохи». Следующие три байта – идентификатор компьютера, а следующие два – идентификатор процесса. Последние три байта – локальный для процесса счетчик, который увеличивается при генерации каждого следующего идентификатора объекта.

Попутное преимущество использования идентификаторов объектов MongoDB заключается в наличии временной метки. Большинство драйверов позволяют извлечь эту метку и тем самым узнать время создания документа с точностью до секунды. В частности, в драйвере для Ruby можно вызвать метод `generation_time` идентификатора объекта и получить время его создания в виде объекта Ruby `Time`:

```
irb(main):002:0> id = BSON::ObjectId.new
=> BSON::ObjectId('4c41e78f238d3b9090000001')
irb(main):003:0> id.generation_time
=> Sat Jul 17 17:25:35 UTC 2010
```

Естественно, идентификаторы объектов можно использовать для запросов по диапазону времени создания документа. Например, чтобы найти все документы, созданные в октябре и ноябре 2010 года, можно создать два идентификатора объекта, в которых временные метки будут представлять границы диапазона, а затем выполнить запрос по диапазону поля _id. Так как Ruby предоставляет метод для генерации идентификатора объекта по объекту `Time`, то соответствующий код получается тривиальным:

```
oct_id = BSON::ObjectId.from_time(Time.utc(2010, 10, 1))
nov_id = BSON::ObjectId.from_time(Time.utc(2010, 11, 1))
```

```
@users.find({'_id' => {'$gte' => oct_id, '$lt' => nov_id}})
```

Я объяснил, зачем нужны идентификаторы объектов MongoDB и что означают отдельные байты. Осталось узнать, как они кодируются. Это тема следующего раздела, в котором мы обсудим формат BSON.

3.2.2. Формат BSON

BSON – это двоичный формат представления документов в MongoDB. Он используется как для хранения данных, так и для кодирования команд. Иначе говоря, все документы хранятся на диске в формате BSON, и он же применяется для описания запросов и команд. Поэтому любой драйвер MongoDB должен уметь выполнять преобразование из языково-зависимого представления документа в формат BSON и обратно.

Спецификация BSON определяет, какие типы данных можно использовать в MongoDB. Знание о типах данных, представимых с помощью BSON, и о том, как они кодируются, очень помогает использовать MongoDB эффективно и диагностировать возникающие проблемы с производительностью.

На момент написания этой книги спецификация BSON включала 19 типов данных. Это означает, что для сохранения в базе данных MongoDB любое присутствующее в документе значение должно допускать преобразование в один из этих типов. Поддерживаются многие стандартные типы: строка в кодировке UTF-8, 32- и 64-разрядное целое, числа с двойной точностью, булевские величины, временная метка и дата/время в часовом поясе UTC. Но есть и типы, специфичные для MongoDB. Например, для описанного выше идентификатора объекта отведен особый тип. Имеется тип `binary` для непрозрачных двоичных объектов. Есть даже тип `symbol` для языков, которые поддерживают это понятие.

На рис. 3.2 показано, как хеш Ruby сериализуется в корректный BSON-документ. Документ Ruby содержит идентификатор объекта и строку. В начале соответствующего BSON-документа находится заголовок, описывающий размер документа (в данном случае 38 байтов). Затем идут две пары ключ-значение. Каждая пара начинается байтом, обозначающим тип, затем следует завершающаяся нулем строка с именем ключа, а вслед за ней – значение. Весь документ завершается нулевым байтом.

Хотя знать формат BSON вдоль и поперек необязательно, опыт показывает, что знакомство с ним идет разработчику на пользу. Рас-

Рис. 3.2. Преобразование из Ruby в BSON

смотрим всего один пример: идентификатор объекта можно представить в виде строки или в виде значения типа «object ID». Поэтому в оболочке следующие два запроса эквивалентны:

```
db.users.find({_id : ObjectId('4c41e78f238d3b9090000001')});
db.users.find({_id : '4c41e78f238d3b9090000001'})
```

Но лишь один из них будет соответствовать значению в поле `_id`, а какой именно, зависит от того, как хранятся идентификаторы документов в коллекции `users`: как значения типа «object ID» или как BSON-строки, содержащие шестнадцатеричные представления идентификаторов[3]. А рассказал я это, чтобы убедить вас в том, что даже поверхностное знание BSON способно здорово помочь при диагностике простых ошибок.

[3] Кстати, хранить идентификаторы объектов MongoDB следует в виде значений типа «object ID», а не строк. И не только чтобы соблюсти соглашение о хранении идентификаторов, но и потому что строки занимают в два с лишним раза больше места.

3.2.3. Передача по сети

Помимо генерации идентификаторов объектов и преобразования в формат BSON, у драйвера MongoDB есть еще одна очевидная функция: взаимодействие с сервером базы данных. Как уже отмечалось, взаимодействие производится через TCP-сокет по специальному сетевому протоколу[4]. Это низкоуровневый аспект, малоинтересный большинству разработчиков приложений. Важно лишь понимать, когда драйвер дожидается ответа от сервера, а когда просто отправляет команду и «забывает» о ней.

Я уже рассказывал о том, как работают запросы на выборку; очевидно, что на каждый запрос должен быть ответ. Напомню, что запрос инициализируется, когда программа вызывает метод `next` курсора. В этот момент запрос отправляется серверу, а в ответ возвращается порция документов. Если этой порцией ответ исчерпывается, то больше обращаться к серверу не нужно. Но если результатов больше, чем помещается в первый ответ сервера, то для получения следующей порции серверу посылается команда `getmore`. По мере обхода курсора серверу посылаются дополнительные команды `getmore`, пока запрос не будет полностью выполнен.

С точки зрения поведения сети, во всем этом нет никаких сюрпризов. Но ситуация меняется, когда речь заходит об операциях записи (вставка, обновление и удаление). Дело в том, что при записи драйвер по умолчанию не ждет ответа от сервера, то есть, когда вы вставляете документ, драйвер посылает данные в сокет и предполагает, что операция записи завершится успешно. Это возможно, в частности, потому что генерация идентификатора объекта производится на стороне клиента, – раз первичный ключ документа уже известен, то нет нужды ждать, пока сервер вернет его.

Такая стратегия «выстрелил и забыл» многих пользователей настораживает, но, к счастью, она настраивается. Все драйверы реализуют режим безопасной записи, который можно включить для любой операции записи (вставки, обновления или удаления). В Ruby безопасная вставка производится так:

```
@users.insert({"last_name" => "james"}, :safe => true)
```

При записи в безопасном режиме драйвер включает в сообщение о вставке специальную команду `getlasterror`. Тем самым достигаются две вещи. Во-первых, поскольку `getlasterror` – команда и, стало быть, подразумевает обращение к серверу, то гарантируется, что сооб-

[4] Некоторые драйверы поддерживают также взаимодействие через Unix-сокеты.

щение о записи дошло до сервера. Во-вторых, эта команда проверяет, что на текущем соединении не было ошибки сервера. Если ошибка была, то драйвер возбудит исключение, которое можно обработать. Использовать безопасный режим имеет смысл, когда нужно гарантировать, что критически важные операции записи дошли до сервера, а также в случае, когда имеются основания ожидать конкретной ошибки. Например, часто требуется обеспечить уникальность значения. Так, над коллекцией данных о пользователе может быть построен уникальный индекс по полю `username`. При наличии такого индекса попытка вставки документа с повторяющимся значением `username` завершится ошибкой, но единственный способ узнать об этом во время вставки – воспользоваться безопасным режимом.

В большинстве случаев будет благоразумно включать безопасный режим по умолчанию. А в тех частях приложениях, где записываются не слишком ценные данные, но требуется обеспечить высокую производительность, этот режим можно отключать. Решить, что важнее, не всегда просто, поэтому имеется еще несколько параметров безопасного режима. Но более подробное обсуждение этой темы мы отложим до главы 8.

Сейчас вы, наверное, лучше понимаете, как работают драйверы, и сгораете от нетерпения, мечтая приступить к разработке реального приложения. В следующем разделе мы воспользуемся всем, что знаем, и с помощью драйвера для Ruby напишем простое приложение для мониторинга сайта Twitter.

3.3. Разработка простого приложения

Мы напишем простое приложение для архивации и отображения «твитов». Можете считать его компонентом более крупного приложения, которое позволяет пользователю следить за поисковыми терминами, относящимися к его бизнесу. На этом примере мы продемонстрируем, как легко обрабатывать и преобразовывать в документы MongoDB ответы в формате JSON, возвращаемые API сайта Twitter и ему подобными. Если бы вы использовали для решения этой задачи реляционную базу данных, то сначала пришлось бы разработать схему, насчитывающую, скорее всего, несколько таблиц, а затем эти таблицы создать. Здесь ничего такого не требуется, и тем не менее мы сохраним структуру документов, описывающих «твиты», и сможем эффективно искать их.

Приложение, которое мы назовем TweetArchiver, будет состоять из двух компонентов: архиватор и визуализатор. Архиватор будет вызывать API поиска, предоставляемый Twitter, и сохранять найденные «твиты», а визуализатор – показывать результаты в браузере.

3.3.1. Подготовка

Нашему приложению понадобятся три библиотеки Ruby. Установить их можно следующим образом:

```
gem install mongo
gem install twitter
gem install sinatra
```

Полезно завести конфигурационный файл, который смогут использовать скрипты архиватора и визуализатора. Создайте файл config.rb и поместите в него такой код:

```
DATABASE_NAME = "twitter-archive"
COLLECTION_NAME = "tweets"
TAGS = ["mongodb", "ruby"]
```

В начале задаются имена базы данных и коллекции, с которыми будет работать приложение. Затем определяется массив поисковых терминов, которые мы отправляем Twitter API.

Далее следует написать скрипт архиватора. Начнем с класса `TweetArchiver`. Конструктору этого класса передается поисковый термин. Затем мы вызываем метод `update` объекта `TweetArchiver`, который вызывает Twitter API и сохраняет полученные результаты в коллекции MongoDB.

Первым делом реализуем конструктор класса:

```
def initialize(tag)
   connection = Mongo::Connection.new
   db = connection[DATABASE_NAME]
   @tweets = db[COLLECTION_NAME]

   @tweets.create_index([['id', 1]], :unique => true)
   @tweets.create_index([['tags', 1], ['id', -1]])

   @tag = tag
   @tweets_found = 0
end
```

Метод `initialize` создает объекты соединения, базы данных и коллекцию, в которой будут храниться «твиты». Он также создает два индекса. У каждого «твита» будет поле `id` (не путайте с внутренним

полем _id, зарезервированным для нужд MongoDB), в котором хранится идентификатор «твита», присвоенный Twitter. Индекс по этому полю *уникален*, чтобы случайно не вставить один и тот же «твит» дважды.

Мы также создаем составной индекс по полям tags (в порядке возрастания) и id (в порядке убывания). Порядок сортировки имеет значение в основном при создании составных индексов, и выбирать его следует с учетом ожидаемых запросов. Поскольку мы собираемся запрашивать информацию о конкретном теге и показывать результаты в порядке от новых к старым, то выбранный порядок гарантирует, что индекс можно будет использовать как для фильтрации, так и для сортировки результатов. Как видите, направление 1 означает сортировку *по возрастанию*, а направление -1 – *по убыванию*.

3.3.2. Сбор данных

MongoDB позволяет вставлять данные любой структуры. Поскольку нам не нужно заранее знать, какие поля присутствуют в результате, то даже если Twitter изменит свой API в части возвращаемых значений, на нашем приложении это практически не отразится. Напротив, в случае использования РСУБД любое изменение Twitter API (и, более общо, любое изменение источника данных) потребовало бы модификации схемы. В MongoDB приложению, быть может, и придется подстраиваться под новую структуру данных, но сама база способна автоматически обработать документы с любой схемой.

Библиотека Ruby Twitter возвращает хеши Ruby, которые можно напрямую передавать объекту коллекции MongoDB. Добавим в класс TweetArchiver следующий метод экземпляра:

```
def save_tweets_for(term)
   Twitter::Search.new.containing(term).each do |tweet|
      @tweets_found += 1
      tweet_with_tag = tweet.to_hash.merge!({"tags" => [term]})
      @tweets.save(tweet_with_tag)
   end
end
```

Перед тем как сохранять документ с «твитом», мы вносим одну мелкую модификацию. Чтобы упростить последующие запросы, мы добавляем поисковый термин в атрибут tags. Затем модифицированный документ передается методу save. Ниже приведен полный текст класса архиватора.

Листинг 3.1. Класс для отбора «твитов» и их сохранения в базе данных MongoDB

```
require 'rubygems'
require 'mongo'
require 'twitter'

require 'config'

class TweetArchiver

   # Создать экземпляр TweetArchiver
   def initialize(tag)
      connection = Mongo::Connection.new
      db = connection[DATABASE_NAME]
      @tweets = db[COLLECTION_NAME]

      @tweets.create_index([['id', 1]], :unique => true)
      @tweets.create_index([['tags', 1], ['id', -1]])

      @tag = tag
      @tweets_found = 0
   end

   def update
      puts "Запускается поиск в Twitter для '#{@tag}'..."
      save_tweets_for(@tag)
      print "Сохранено твитов: #{@tweets_found}.\n\n"
   end

   private

   def save_tweets_for(term)
      Twitter::Search.new(term).each do |tweet|
         @tweets_found += 1
         tweet_with_tag = tweet.to_hash.merge!({"tags" => [term]})
         @tweets.save(tweet_with_tag)
      end
   end
end
```

Осталось только написать скрипт, который будет запускать `TweetArchiver` для каждого поискового термина. Создайте файл update.rb и поместите в него такой код:

```
require 'config'
require 'archiver'

TAGS.each do |tag|
   archive = TweetArchiver.new(tag)
   archive.update
end
```

Теперь запустите этот скрипт:

```
ruby update.rb
```

Будут напечатаны сообщения о том, сколько «твитов» найдено и сохранено. Убедиться в том, что скрипт отработал, можно, открыв оболочку MongoDB и опросив коллекцию напрямую:

```
> use twitter-archive
switched to db twitter-archive
> db.tweets.count()
30
```

Чтобы поддерживать архив в актуальном состоянии, этот скрипт можно запускать раз в несколько минут из задания cron. Но это уже административные детали. Важно то, что для поиска и сохранения «твитов» оказалось достаточно написать несколько строк кода[5]. Перейдем теперь к задаче отображения результатов.

3.3.3. Визуализация архива

При разработке простого приложения для показа результатов мы воспользуемся написанным на Ruby веб-каркасом Sinatra. Создайте файл viewer.rb и поместите его в тот же каталог, где находятся остальные скрипты. Затем создайте подкаталог `views` и поместите в него файл с именем tweets.erb. Дерево проекта должно выглядеть следующим образом:

```
- config.rb
- archiver.rb
- update.rb
- viewer.rb
- /views
- tweets.erb
```

Теперь введите в файл viewer.rb следующий код.

Листинг 3.2. Простое приложение Sinatra для отображения результатов поиска в архиве «твитов»

```
require 'rubygems'
require 'mongo'
require 'sinatra'

require 'config'

configure do
```

❶ Подключаем библиотеки

[5] Можно было бы обойтись и гораздо меньшим числом строк. Оставляем это в качестве упражнения для читателя.

```
   db = Mongo::Connection.new[DATABASE_NAME]
   TWEETS = db[COLLECTION_NAME]                    ❷ Создаем коллекцию «твитов»
end

get '/' do
   if params['tag']
      selector = {:tags => params['tag']}          ❸ Динамически строим селектор запроса
   else
      selector = {}                                ❹ Или используем пустой селектор
   end

@tweets = TWEETS.find(selector).sort(["id", -1])   ❺ Отправляем запрос

   erb :tweets                                     ❻ Отрисовываем представление
end
```

В начальных строчках подключаются необходимые библиотеки и конфигурационный файл (❶). Затем идет блок конфигурирования, в котором создается соединение с MongoDB и в константе `TWEETS` сохраняется ссылка на коллекцию `tweets` (❷).

Содержательная часть приложения находится в строках, начиная с `get '/' do`. Код в этом блоке обрабатывает запросы, адресованные корневому URL приложения. Сначала мы строим селектор запроса. Если в URL присутствует параметр `tags`, то создается селектор, отбирающий только указанные теги (❸). В противном случае создается пустой селектор, который вернет все хранящиеся в коллекции документы (❹). После этого отправляется запрос (❺). Теперь вы уже знаете, что переменной `@tweets` присваивается не результирующий набор, а курсор. Обход этого курсора производится в представлении.

В последней строке (❻) отрисовывается файл представления tweets.erb, полный код которого приведен ниже.

Листинг 3.3. HTML-разметка с внедренным кодом Ruby для визуализации «твитов»

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang='en' xml:lang='en' xmlns='http://www.w3.org/1999/xhtml'>
<head>
   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>

   <style>
      body {
         background-color: #DBD4C2;
         width: 1000px;
         margin: 50px auto;
      }

      h2 {
```

```
        margin-top: 2em;
      }
   </style>

</head>

<body>

<h1>Tweet Archive</h1>

<% TAGS.each do |tag| %>
   <a href="/?tag=<%= tag %>"><%= tag %></a>
<% end %>

<% @tweets.each do |tweet| %>
   <h2><%= tweet[,text'] %></h2>
   <p>
      <a href="http://twitter.com/<%= tweet[,from_user'] %>">
         <%= tweet[,from_user'] %>
      </a>
      on <%= tweet[,created_at'] %>
   </p>

   <img src="<%= tweet[,profile_image_url'] %>" width="48" />
<% end %>

</body>
</html>
```

Это в основном обычная HTML-разметка с вкраплениями ERB-кода[6]. Наиболее интересные части – два итератора – находятся ближе к концу. В первом строится список тегов, каждый из которых представлен ссылкой на относящийся к нему результирующий набор. Второй итератор (начинается с `@tweets.each`) обходит все «твиты» и отображает текст, дату создания и изображение из профиля пользователя. Запустите приложение и посмотрите, как выглядит результат.

```
$ ruby viewer.rb
```

Если не случится каких-либо ошибок, то на консоли будет напечатано стандартное приветственное сообщение Sinatra:

```
$ ruby viewer.rb
== Sinatra/1.0.0 has taken the stage on 4567 for development
with backup from Mongrel
```

Теперь вы можете ввести в адресной строке браузера URL http://localhost:4567. Появится страница, похожая на изображенную на

[6] *ERB* означает *embedded Ruby* (внедренный Ruby). Каркас Sinatra прогоняет файл tweets.erb через процессор ERB и интерпретирует Ruby-код между скобками <% и %> в контексте приложения.

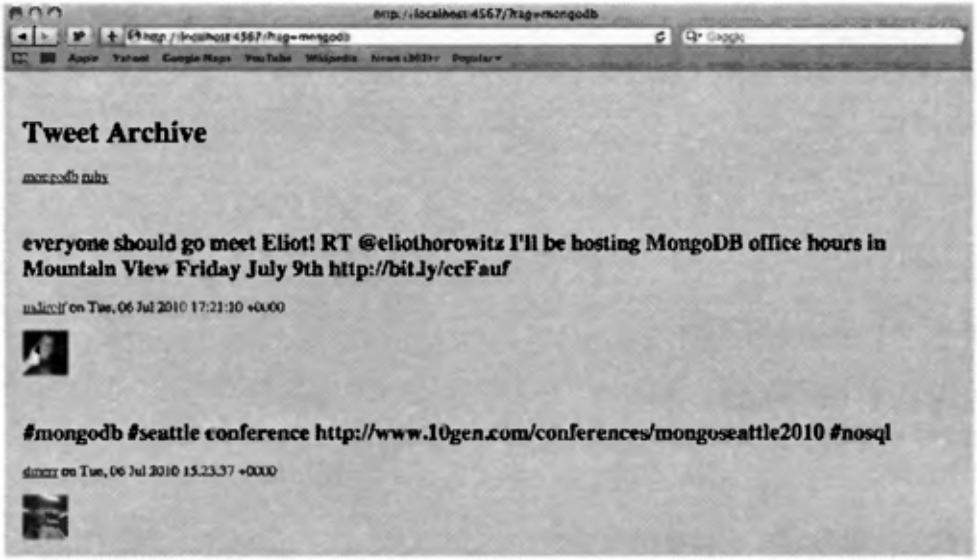


Рис. 3.3. Результат работы приложения Tweet Archiver в браузере

рис. 3.3. Попробуйте пощелкать по ссылкам в верхней части, чтобы ограничить результаты одним тегом.

Вот и всё приложение. Согласен, простенькое, но все-таки демонстрирует, насколько легко работать с MongoDB. Вам не пришлось заранее описывать схему, вы воспользовались вторичными индексами для ускорения поиска и предотвращения дубликатов, и интеграция с языком программирования также оказалась относительно несложной.

3.4. Резюме

Выше мы рассмотрели основы работы с MongoDB из программ на языке Ruby. Вы видели, как просто представляются в Ruby документы и как похож API для выполнения операций CRUD на API оболочки MongoDB. Мы слегка коснулись внутреннего устройства – поговорили о том, как работают драйверы, и познакомились с идентификаторами объектов, форматом BSON и сетевым протоколом MongoDB. Наконец, мы разработали простое приложение для демонстрации того, как MongoDB работает с реальными данными. Конечно, вы еще не чувствуете, что овладели MongoDB в совершенстве, но перспектива написания приложений для этой СУБД уже не должна казаться отдаленной.

Начиная с главы 4, мы начнем углубляться. Точнее, рассмотрим, как создать интернет-магазин с MongoDB в качестве хранилища данных. Проект огромный, поэтому мы сосредоточим внимание только на нескольких аспектах серверной части. Я расскажу о некоторых моделях данных предметной области и объясню, как вставлять и запрашивать такого рода данные.

Часть 2.

РАЗРАБОТКА ПРИЛОЖЕНИЯ ДЛЯ MongoDB

Вторая часть книги посвящена углубленному изучению документной модели данных MongoDB, языка запросов и операций CRUD (создания, чтения, обновления и удаления).

Мы наполним эти темы конкретикой в процессе разработки модели данных интернет-магазина и операций CRUD, необходимых для работы с этими данными. Таким образом, предмет каждой главы излагается в порядке сверху вниз: сначала мы приводим примеры из области электронной торговли, а затем систематически заполняем пробелы. Возможно, при первом чтении вы захотите только познакомиться с примерами, а детали отложить на потом. Или наоборот.

В главе 4 мы поговорим о некоторых принципах проектирования схемы, а затем опишем модели данных для товаров, категорий, пользователей, заказов и отзывов. Потом мы узнаем о том, как в MongoDB организованы база данных, коллекции и документы. В заключение будет приведен перечень основных типов данных BSON.

Глава 5 посвящена языку запросов и агрегатным функциям в MongoDB. Мы расскажем о том, как формулировать типичные запросы к модели данных, разработанной в предыдущей главе, и на практике освоим некоторые виды агрегирования. Затем мы детально обсудим семантику операторов запроса. Заканчивается глава рассказом о функциях распределения-редукции и группировки.

Изложением материала об операциях обновления и удаления в MongoDB глава 6 замыкает круг и объясняет, почему была выбрана именно такая модель данных для электронной торговли. Вы узнаете, как работать с иерархией категорий и как организовать транзакционное управление запасами. Наконец, мы подробно обсудим операторы обновления и мощную команду `findAndModify`.

ГЛАВА 4. Документо-ориентированные данные

В этой главе:

- Проектирование схемы.
- Модели данных для интернет-магазина.
- Базы данных, коллекции и документы.

В этой главе мы ближе познакомимся с документо-ориентированным моделированием данных и с тем, как организованы данные в MongoDB – на уровне базы данных, коллекции и документа. Я начну с краткого обсуждения общих принципов проектирования схемы. Это полезно, потому что многие пользователи MongoDB ранее проектировали схемы только традиционных реляционных СУБД. Изложение этих принципов заложит основы для второй половины этой главы, где будет рассмотрено проектирование схемы для интернет-магазина. Попутно мы увидим, чем эта схема отличается от эквивалентной реляционной схемы, и узнаем о том, как в MongoDB представляются типичные связи между сущностями, например, типа один-ко-многим и многие-ко-многим. Сконструированная здесь схема интернет-магазина будет затем использоваться при обсуждении запросов, агрегирования и обновления в последующих главах.

Поскольку документы – это плоть и кровь MongoDB, то последнюю часть этой главы я посвящу некоторым мелким деталям и особым случаям, относящимся к документам и их окружению. Для этого потребуется более глубоко, чем до сих пор, вникнуть в базы данных, коллекции и документы. Но если вы дочитаете до конца, то

познакомитесь с самыми темными закоулками и ограничениями документной модели данных в MongoDB. Возможно, вы не раз будете возвращаться к этому последнему разделу, потому что в нем описаны многие подводные камни, с которыми сталкиваешься при практической эксплуатации MongoDB.

4.1. Принципы проектирования схемы

Проектирование схемы базы данных – это процедура выбора наилучшего представления набора данных с учетом возможностей СУБД, природы данных и требований приложения. Принципы проектирования схемы для реляционных баз данных уже давно устоялись. В этом случае предлагается стремиться к *нормализованной* модели, которая позволяет обеспечить возможность запросов общего вида и избежать такого обновления данных, которое могло бы привести к несогласованности. К тому же, проверенные практикой приемы уберегают разработчика от изобретения велосипеда, например: как моделировать связи один-ко-многим и многие-ко-многим. Впрочем, проектирование схемы никогда не было точной наукой, даже в случае реляционных баз данных. Для приложений, критичных к производительности или работающих с неструктурированными данными, может понадобиться более общая модель. Некоторые приложения предъявляют настолько жесткие требования к системе хранения данных и масштабированию, что проектировщики вынуждены ломать все каноны проектирования. Хорошим примером такого рода может служить сайт FriendFeed; вы не пожалеете, потратив время на чтение статьи о примененной в нем неортодоксальной модели данных (http://mng.bz/ycG3).

Если вы пришли к MongoDB из мира РСУБД, то, возможно, вас неприятно удивит отсутствие твердых правил проектирования схемы. Конечно, выработаны некоторые рекомендации, но по-прежнему обычно существует несколько одинаково хороших способов построить модель заданного набора данных. В этом разделе предполагается, что при проектировании схемы можно руководствоваться принципами, но реальность такова, что сами принципы очень подвижны. Чтобы вам было о чем подумать, предлагаю несколько вопросов, которые нужно задать себе при моделировании данных для *любой* СУБД.

- *Что представляет собой структурная единица данных?* В РСУБД имеются таблицы, состоящие из строк и столбцов.

В хранилище ключей и значений имеются ключи, указывающие на аморфные значения. В MongoDB структурной единицей является BSON-документ.

- *Как опрашивать и обновлять данные?* Определившись со структурным делением данных, надо понять, как ими манипулировать. В РСУБД предлагаются произвольные запросы и операция соединения. В MongoDB произвольные запросы также допускаются, но соединение не поддерживается. В простых хранилищах ключей и значений разрешена только выборка значений по одному ключу.
  Базы данных различаются и по допустимым видам операций обновления. РСУБД позволяет обновлять записи изощренными способами с использованием языка SQL, а также помещать несколько операций обновления в транзакцию, гарантирующую атомарность и возможность отката. MongoDB не поддерживает транзакции, зато поддерживает различные способы атомарного обновления внутренних структур сложного документа. В простых хранилищах ключей и значений возможность обновления значения иногда присутствует, но при каждом обновлении значение полностью переписывается.
  Важно уяснить, что для построения оптимальной модели данных необходимо хорошо понимать возможности СУБД. Если вы хотите научиться моделировать данные в MongoDB, то для начала должны разобраться, на какие виды запросов и обновлений эта СУБД рассчитана.
- *Каковы типичные способы доступа к данным в вашем приложении?* Помимо знания структурных единиц данных и возможностей СУБД, вы должны еще четко представлять потребности конкретного приложения. Если вы прочитали вышеупомянутую статью о сайте FriendFeed, то понимаете, как особенности приложения могут вынудить отойти от твердых принципов моделирования схемы. Итоговый вывод заключается в том, что для выбора идеальной модели данных нужно ответить на многочисленные вопросы о приложении. Каково соотношение операций чтения и записи? Какого рода запросы будет предъявлять приложение? Как обновляются данные? Какие ожидаются проблемы с параллельным доступом? Насколько хорошо структурированы данные?

Оптимальный проект схемы всегда является результатом глубокого понимания используемой СУБД, правильного представления

о требованиях приложения и просто опыта. Примеры в этой главе и принципы проектирования схемы, изложенные в приложении В, составлены так, чтобы помочь вам выработать внутреннее ощущение правильности при проектировании схем для MongoDB.

4.2. Проектирование модели данных для интернет-магазина

При демонстрации хранилищ данных нового поколения обычно говорят о социальных сетях – нормой служат приложения типа Twitter. К сожалению, модели данных для таких приложений обычно довольно просты. Именно поэтому в этой и следующих главах мы будем рассматривать гораздо более насыщенную предметную область электронной торговли. Достоинство ее в том, что можно применить много знакомых паттернов моделирования. К тому же, нетрудно сообразить, как товары, категории, отзывы и заказы обычно моделируются в РСУБД. Поэтому приводимые ниже примеры будут более поучительными, так как вы сможете сопоставить их с ранее усвоенными представлениями о проектировании схемы.

Электронную торговлю принято считать предметной областью, предназначенной исключительно для РСУБД, и это справедливо по двум причинам. Во-первых, для интернет-магазинов обычно необходимы транзакции, а транзакции – это неотъемлемая часть РСУБД. Во-вторых, до недавнего времени считалось, что предметные области, нуждающиеся в развитых моделях данных и сложных запросах, лучше всего ложатся именно на РСУБД. В примерах ниже второе положение ставится под сомнение.

Но прежде чем идти дальше, уместно будет сделать замечание о рамках проекта. Разработать серверную часть интернет-магазина целиком нам не позволит объем книги. Вместо этого мы отберем некоторые сущности, характерные для электронной торговли – товары, категории, пользователи, заказы, отзывы о товарах – и покажем, как их можно смоделировать в MongoDB. Для каждой сущности я приведу пример документа, а затем расскажу о некоторых средствах СУБД, которые подкрепляют структуру документа.

Для многих разработчиков *модель данных* неразрывно связана с *отображением на объекты*, и именно поэтому вы, наверное, пользовались такими библиотеками объектно-реляционного отображения, как Hibernate для Java или ActiveRecord для Ruby. Без подобных библио-

тек трудно обойтись, если хочешь эффективно создавать приложения для РСУБД. Но в MongoDB необходимость в них не так насущна. Отчасти это связано с тем, что документ и так является чем-то вроде объектного представления. А отчасти обусловлено драйверами, предоставляющими высокоуровневый интерфейс к MongoDB. Зачастую приложение для MongoDB можно написать, пользуясь только интерфейсом драйвера.

Но тем не менее системы объектно-реляционного отображения удобны, так как упрощают проверку корректности данных, контроль типов и реализацию ассоциаций. Существует ряд достаточно зрелых ORM-систем для MongoDB, которые предоставляют дополнительный уровень абстракции поверх языковых драйверов, и при работе над большим проектом стоит подумать об их использовании[1]. Однако, есть ORM или нет, в конечном счете вы всегда имеете дело с документами. Поэтому в этой главе мы сосредоточимся на документах как таковых. Понимание структуры документов в хорошо спроектированной схеме MongoDB позволит вам работать с базой данных осмысленно как с применением ORM-системы, так и без нее.

4.2.1. Товары и категории

Товары и категории – столпы, на которых покоится сайт любого интернет-магазина. Для представления товаров в нормализованной модели для РСУБД требуется много таблиц. Всегда существует таблица для основной информации о товаре, например названии и артикуле. Но помимо нее есть и другие таблицы, связывающие товар с информацией об отгрузке и историей изменения цен. Если система допускает товары с произвольными атрибутами, то для определения и хранения этих атрибутов нужен сложный набор таблиц – как в примере системы Magento, приведенном в главе 1. Такая схема с многочисленными таблицами возможна благодаря способности РСУБД соединять таблицы.

Моделировать товар в MongoDB проще. Поскольку коллекции не нуждаются в схеме, в любом документе о товаре есть место для произвольных динамических атрибутов. А за счет применения массивов для хранения внутренних структур документа обычно можно уместить многотабличное реляционное представление в одну коллекцию MongoDB. Ниже приведен пример конкретного документа о товаре, который продается в магазине для садоводов.

[1] Дополнительные сведения о самых последних версиях ORM-систем для своего любимого языка можно найти на сайте http://mongodb.org.

Листинг 4.1. Пример документа с описанием товара

```
doc =
{  _id: new ObjectId("4c4b1476238d3b4dd5003981"),
   slug: "wheel-barrow-9092",
   sku: "9092",
   name: "Extra Large Wheel Barrow",
   description: "Heavy duty wheel barrow...",

   details: {
      weight: 47,
      weight_units: "lbs",
      model_num: 4039283402,
      manufacturer: "Acme",
      color: "Green"
   },

   total_reviews: 4,
   average_review: 4.5,

   pricing: {
      retail: 589700,
      sale: 489700,
   },

   price_history: [
      {  retail: 529700,
         sale: 429700,
         start: new Date(2010, 4, 1),
         end: new Date(2010, 4, 8)
      },

      {  retail: 529700,
         sale: 529700,
         start: new Date(2010, 4, 9),
         end: new Date(2010, 4, 16)
      },
   ],

   category_ids: [new ObjectId("6a5b1476238d3b4dd5000048"),
                  new ObjectId("6a5b1476238d3b4dd5000049")],

   main_cat_id: new ObjectId("6a5b1476238d3b4dd5000048"),

   tags: ["tools", "gardening", "soil"],
}
```

Этот документ содержит основные поля: `name`, `sku` и `description`. В поле `_id` хранится также стандартный идентификатор объекта MongoDB. Кроме того, мы определили краткое название

wheel-barrow-9092 для формирования осмысленного URL-адреса. Пользователи MongoDB иногда жалуются на присутствие некрасивых идентификаторов объектов в URL. Конечно, кому захочется видеть такой URL:

```
http://mygardensite.org/products/4c4b1476238d3b4dd5003981
```

Понятные пользователю идентификаторы выглядят куда лучше:

```
http://mygardensite.org/products/wheel-barrow-9092
```

В общем случае я рекомендую заводить поле для краткого названия (slug), если URL-адрес документа генерируется программой. По такому полю необходимо строить уникальный индекс, чтобы его значение можно было использовать в качестве первичного ключа. В предположении, что документы хранятся в коллекции products, создать уникальный индекс можно так:

```
db.products.ensureIndex({slug: 1}, {unique: true})
```

Поскольку по полю slug построен уникальный индекс, вставлять документы о товарах следует в безопасном режиме, чтобы можно было узнать, завершилась ли вставка нормально или с ошибкой. В случае ошибки можно будет присвоить товару другое краткое название. При добавлении тачки (wheelbarrow) в каталог программа должна будет сгенерировать для нее уникальное краткое название. Вот как производится вставка на Ruby:

```
@products.insert({:name => "Extra Large Wheel Barrow",
   :sku => "9092",
   :slug => "wheel-barrow-9092"},
   :safe => true)
```

Обратите внимание на параметр :safe => true. Если вставка не возбудит исключение, значит, краткое название уникально. В противном случае программа должна будет выбрать другое краткое название.

Следующий ключ, details, указывает на поддокумент, содержащий различные сведения о товаре. Здесь мы храним вес, единицу измерения веса и номер модели по каталогу производителя. При желании сюда можно поместить дополнительные атрибуты. Например, если магазин торгует семенами, то можно хранить ожидаемую урожайность и время созревания, а для газонокосилок можно указать мощность, тип топлива и приспособления для мульчирования. Контейнер details – подходящее место для подобных динамических атрибутов.

В том же документе можно хранить текущую и старые цены. Ключ `pricing` указывает на объект, содержащий розничную и распродажную цену, а `price_history` ссылается на целый массив данных о ценах. Такое хранение нескольких документов – общеупотребительная техника отслеживания версий.

Далее идет массив тегов, ассоциированных с товаром. Подобный пример мы уже видели в главе 1, но сама техника настолько распространена, что стоит повторить еще раз. Поскольку по ключу, ссылающемуся на массив, можно построить индекс, то это самый простой способ хранить ассоциированные с объектом теги, обеспечив в то же время возможность эффективного поиска по ним.

На как насчет связей? Да, мы можем использовать поддокументы и массивы для хранения детальных сведений о товаре, цен и тегов в одном документе. Но тем не менее, иногда возникает необходимость связывать документы из разных коллекций. За примером далеко ходить не надо – товары должны быть ассоциированы с категориями. Обычно связь между товарами и категориями имеет тип многие-ко-многим, то есть один товар может принадлежать нескольким категориям, а в каждой категории может быть много товаров. В РСУБД для представления связей многие-ко-многим используются связующие таблицы, в которых хранятся пары соответственных ключей из каждой таблицы. А оператор SQL `join` позволяет с помощью одного запроса выбрать товар вместе со всеми его категориями или категорию со всеми входящими в нее товарами.

MongoDB не поддерживает соединения, поэтому для реализации связей многие-ко-многим нужна иная стратегия. В приведенном выше документе, описывающем тачку, имеется поле `category_ids`, содержащее массив идентификаторов объектов. Каждый идентификатор играет роль указателя на поле `_id` в некотором документе, описывающем категорию. Для справки приведем пример такого документа.

Листинг 4.2. Документ, описывающий категорию

```
doc =
{  _id: new ObjectId("6a5b1476238d3b4dd5000048"),
   slug: "gardening-tools",
   ancestors: [{ name: "Home",
                 _id: new ObjectId("8b87fb1476238d3b4dd500003"),
                 slug: "home"
               },

               { name: "Outdoors",
                 _id: new ObjectId("9a9fb1476238d3b4dd5000001"),
```

```
                slug: "outdoors"
              }
  ],

  parent_id: new ObjectId("9a9fb1476238d3b4dd5000001"),

  name: "Gardening Tools",
  description: "Gardening gadgets galore!",
}
```

Вернитесь к документу с описанием товара и посмотрите на идентификаторы объектов в поле `category_ids`. Вы увидите, что этот товар связан с только что показанной категорией Gardening Tools (Садовые инструменты). Наличие ключа-массива `category_ids` в документе, описывающем товар, позволяет формулировать все те же запросы, что для связи типа многие-ко-многим. Например, чтобы найти все товары в категории Gardening Tools, нужно написать такой простой код:

```
db.products.find({category_ids => category['_id']})
```

Чтобы найти все категории для данного продукта, используется оператор `$in` – аналог оператора SQL `IN`:

```
db.categories.find({_id: {$in: product['category_ids']}})
```

Пояснив, как реализуется связь многие-ко-многим, я хотел бы сказать несколько слов о самом документе с описанием категории. В нем вы видите стандартные поля `_id`, `slug`, `name` и `description`. С ними все понятно, чего не скажешь о массиве родительских документов `ancestors`. К чему это избыточное хранение столь обширной информации о родительских категориях в каждом документе? Дело в том, что множество категорий принято организовывать в виде древовидной иерархии, а способов представления такой иерархии в базе данных несколько[2]. Какой подход выбрать, зависит от приложения. В данном случае, поскольку MongoDB не поддерживает соединения, мы решили пойти на денормализацию и хранить названия родительских категорий в каждой дочерней категории. Таким образом, при запросе категории Gardening Products нам не придется выполнять дополнительные запросы для получения названий и URL-адресов родительских категорий Outdoors (Вне дома) и Home (Начало).

Кто-то, возможно, сочтет такую денормализацию неприемлемой. Существуют и другие способы представления дерева, один из них

[2] Два таких метода, списки смежности и вложенные множества, описаны в моей статье на сайте для разработчиков MySQL по адресу http://mng.bz/83w4.

обсуждается в приложении В. Но пока постарайтесь принять допущение, что оптимальная схема определяется потребностями приложения, а не теоретическими постулатами. Когда в следующих двух главах вы увидите примеры запросов на поиск и обновление этой структуры, то подоплека такого решения станет яснее.

4.2.2. Пользователи и заказы

При рассмотрении модели пользователей и заказов мы встречаемся еще с одним видом связей: один-ко-многим. Иными словами, у каждого пользователя может быть много заказов. В РСУБД вы завели бы внешний ключ в таблице заказов; в MongoDB действует аналогичное соглашение. Взгляните на следующий листинг.

Листинг 4.3. Заказ в интернет-магазине, содержащий позиции, цены и адрес доставки

```
doc =
{  _id: ObjectId("6a5b1476238d3b4dd5000048")
   user_id: ObjectId("4c4b1476238d3b4dd5000001")

   state: "CART",

   line_items: [
      {  _id: ObjectId("4c4b1476238d3b4dd5003981"),
         sku: "9092",
         name: "Extra Large Wheel Barrow",
         quantity: 1,
         pricing: {
            retail: 5897,
            sale: 4897,
         }
      },

      {  _id: ObjectId("4c4b1476238d3b4dd5003981"),
         sku: "10027",
         name: "Rubberized Work Glove, Black",
         quantity: 2,
         pricing: {
            retail: 1499,
            sale: 1299
         }
      }
   ],

   shipping_address: {
      street: "588 5th Street",
      city: "Brooklyn",
```

```
      state: "NY",
      zip: 11215
   },

   sub_total: 6196
}
```

Во втором атрибуте заказа, `user_id`, хранится идентификатор `_id` пользователя. Это не что иное как указатель на пользователя, описанного в листинге 4.4 (мы рассмотрим его чуть ниже). При такой организации легко формулировать запросы к объектам по любую сторону связи. Вот как просто найти все заказы данного пользователя:

```
db.orders.find({user_id: user['_id']})
```

Запрос на поиск пользователя, которому принадлежит конкретный заказ, ничуть не сложнее:

```
user_id = order['user_id']
db.users.find({_id: user_id})
```

Используя идентификатор объекта в качестве ссылки, легко смоделировать связь один-ко-многим между пользователями и заказами.

Теперь обратимся к другим интересным аспектам документа с описанием заказа. Мы используем общие возможности, предоставляемые документной моделью данных. Как видите, документ включает массив позиций и адрес доставки. В нормализованной реляционной модели эти атрибуты хранились бы в отдельных таблицах. Здесь же позиции заказа представлены поддокументами, каждый из которых описывает один товар в корзине. Атрибут `shipping_address` (адрес доставки) указывает на объект, содержащий поля адреса.

Обсудим, чем хорошо такое представление. Во-первых, оно легко укладывается в голове. Вся концепция заказа – позиции, адрес доставки, информация о ценах – инкапсулирована в одной сущности. Единственный запрос к базе данных позволяет получить объект заказа целиком. Более того, в документе, описывающем заказ, хранится вся информация о товарах, актуальная на момент приобретения. Наконец, как вы увидите ниже и, возможно, догадываетесь уже сейчас, такой документ легко найти и модифицировать.

Документ с описанием пользователя устроен аналогично, в нем хранится список документов, описывающих адреса и платежные реквизиты. И, кроме того, на верхнем уровне находятся атрибуты, типичные для любой модели пользователя. Как и в случае краткого названия товара, по полю `username` строится уникальный индекс.

Листинг 4.4. Документ с описанием пользователя содержит информацию об адресах и платежных реквизитах

```
{  _id: new ObjectId("4c4b1476238d3b4dd5000001"),
   username: "kbanker",
   email: "kylebanker@gmail.com",
   first_name: "Kyle",
   last_name: "Banker",
   hashed_password: "bd1cfa194c3a603e7186780824b04419",

   addresses: [
      {  name: "home",
         street: "588 5th Street",
         city: "Brooklyn",
         state: "NY",
         zip: 11215},

      {  name: "work",
         street: "1 E. 23rd Street",
         city: "New York",
         state: "NY",
         zip: 10010}
   ],

   payment_methods: [
      {  name: "VISA",
         last_four: 2127,
         crypted_number: "43f6ba1dfda6b8106dc7",
         expiration_date: new Date(2014, 4)
      }
   ]
}
```

4.2.3. Отзывы

Мы завершим наш пример описанием модели отзывов о товарах. С реляционной точки зрения, с каждым товаром может быть связано несколько отзывов. В данном случае связь кодируется с помощью ссылки на идентификатор объекта `product_id`, как показано в следующем примере отзыва.

Листинг 4.5. Документ, представляющий отзыв о товаре

```
{  _id: new ObjectId("4c4b1476238d3b4dd5000041"),
   product_id: new ObjectId("4c4b1476238d3b4dd5003981"),
   date: new Date(2010, 5, 7),
   title: "Amazing",
   text: "Has a squeaky wheel, but still a darn good wheel barrow.",
```

```
    rating: 4,

    user_id: new ObjectId("4c4b1476238d3b4dd5000041"),
    username: "dgreenthumb",
    helpful_votes: 3,

    voter_ids: [ new ObjectId("4c4b1476238d3b4dd5000041"),
       new ObjectId("7a4f0376238d3b4dd5000003"),
       new ObjectId("92c21476238d3b4dd5000032")
    ]
}
```

Большинство остальных атрибутов не нуждаются в пояснениях. Мы храним дату, заголовок и текст отзыва, выставленную пользователем оценку и идентификатор пользователя. Но, быть может, вам непонятно, зачем хранить еще и имя пользователя. Ведь в реляционной СУБД вы могли бы найти имя пользователя, соединив отзыв с таблицей *users*. Однако MongoDB не поддерживает соединений, поэтому остается два пути: либо запрашивать коллекцию пользователей для каждого отзыва, либо пойти на денормализацию. Выполнять дополнительный запрос для каждого отзыва дорого и вряд ли необходимо, если принять во внимание, что запрашиваемый атрибут (имя пользователя), скорее всего, никогда не изменяется. Сразу скажем, что можно было бы пойти по пути нормализации – для отображения всех отзывов достаточно было бы двух запросов к MongoDB. Но мы проектируем схему для типичного случая. Да, при таком подходе обновление имени пользователя обойдется дороже, потому что его придется изменить всюду, где оно встречается, но повторим, что это случается настолько редко, что имеет смысл предпочесть выбранное нами решение.

Также стоит обратить внимание на решение хранить оценки в самом объекте отзыва. Часто пользователям разрешено голосовать за отзывы, выражая свое мнение об их полезности. Мы сохраняем идентификаторы всех проголосовавших пользователей в массиве `voter_ids`. Это позволяет предотвратить повторное голосование одним и тем же пользователем, а также найти все отзывы, за которые голосовал данный пользователь. Отметим еще, что мы кэшируем общее число пользователей, которые сочли данный отзыв полезным, – среди прочего, это позволяет сортировать отзывы по полезности.

Итак, мы закончили рассмотрение основных объектов в модели данных интернет-магазина. Если вы впервые сталкиваетесь с моделированием в MongoDB, то убедить себя в ценности такой модели можно, пожалуй, только уверовав в нее. Но не пугайтесь, все особен-

ности работы с ней – добавление уникальных голосов, модификация заказов, интеллектуальный поиск товаров – будут рассмотрены и объяснены в следующих двух главах, посвященных соответственно выборке и обновлению.

4.3. Технические детали: о базах данных, коллекциях и документах

Теперь мы на какое-то время расстанемся с интернет-магазином и обсудим некоторые детали работы с базами данных, коллекциями и документами. Речь пойдет в основном об определениях, специальных возможностях и пограничных случаях. Если вам интересно, как MongoDB распределяет место для файлов данных, какие типы данных разрешены в документах, в чем достоинства ограниченных коллекций, то читайте дальше.

4.3.1. Базы данных

База данных – это логическое и физическое собрание коллекций. В настоящем разделе мы рассмотрим, как создаются и удаляются базы данных. Кроме того, мы опустимся на нижний уровень и расскажем, как MongoDB выделяет место для баз данных в файловой системе.

Управление базами данных

В MongoDB не существует явного способа создать базу данных. База создается автоматически при записи в принадлежащую ей коллекцию. Взгляните на следующий код на Ruby:

```
@connection = Mongo::Connection.new
@db = @connection['garden']
```

В предположении, что база данных еще не существует, она не будет создана на диске даже после исполнения этого кода. Пока что мы лишь создали экземпляр класса `Mongo::DB`. И только после того, как вы что-то запишете в коллекцию, создаются файлы данных. Продолжим:

```
@products = @db['products']
@products.save({:name => "Extra Large Wheel Barrow"})
```

При вызове метода save коллекции products драйвер просит MongoDB вставить документ с описанием товара в пространство имен garden.products. Если такое пространство имен еще не существует, оно создается; часть этого процесса – создание базы данных garden на диске.

Чтобы уничтожить базу данных, то есть удалить все хранящиеся в ней коллекции, нужно выполнить специальную команду. На Ruby база данных garden удаляется так:

```
@connection.drop_database('garden')
```

А в оболочке MongoDB нужно вызвать метод dropDatabase():

```
use garden
db.dropDatabase();
```

Будьте осторожны при удалении баз данных – отменить эту операцию невозможно.

Распределение файлов данных

При создании базы данных MongoDB выделяет на диске место для файлов. В них хранятся все коллекции, индексы и разного рода метаданные, относящиеся к базе. Файлы данных находятся в каталоге, указанном в параметре dbpath при запуске mongod. Если этот параметр не задан, то mongod сохраняет файлы в каталоге /data/db[3]. Вот как выглядит этот каталог после создания базы данных garden:

```
$ cd /data/db
$ ls -al
drwxr-xr-x 6 kyle admin       204 Jul 31 15:48 .
drwxrwxrwx 7 root admin       238 Jul 31 15:46 ..
-rwxr-xr-x 1 kyle admin  67108864 Jul 31 15:47 garden.0
-rwxr-xr-x 1 kyle admin 134217728 Jul 31 15:46 garden.1
-rwxr-xr-x 1 kyle admin  16777216 Jul 31 15:47 garden.ns
-rwxr-xr-x 1 kyle admin         6 Jul 31 15:48 mongod.lock
```

Прежде всего, обратите внимание на файл mongod.lock, в котором хранится идентификатор процесса сервера[4]. В именах файлов базы данных фигурирует имя той базы, к которой они относятся. Первым создается файл garden.ns. Расширение *ns* означает *namespaces*

[3] c:\data\db в Windows.

[4] Ни в коем случае не удаляйте и не изменяйте этот файл, если только не пытаетесь восстановиться после нештатного останова. Если при запуске mongod вы увидите сообщение об ошибке, в котором упоминается lock-файл, то, скорее всего, сервер был остановлен нештатно и требуется восстановление. Мы вернемся к этой теме в главе 10.

(пространства имен). У каждой коллекции и индекса в базе данных есть свое пространство имен, и в этом файле хранятся метаданные обо всех пространствах имен. По умолчанию размер ns-файла составляет 16 МБ, что позволяет сохранить примерно 24 000 пространств имен. Это означает, что суммарное количество коллекций и индексов в базе данных не должно превышать 24 000. Маловероятно, что вам столько понадобится, но в том случае, когда этого все же не хватает, можно увеличить размер файла, задав параметр сервера `--nssize`.

Помимо файла пространств имен, MongoDB выделяет место для коллекций и индексов – в файлах, имена которых заканчиваются увеличивающимися целыми числами, начиная с 0. Так в листинге каталоге вы видите два файла данных: `garden.0` размером 64 МБ и `garden.1` размером 128 МБ. Начальный размер этих файлов у многих пользователей вызывает шок. Но MongoDB предпочитает выделять место заранее, чтобы как можно больше данных хранилось в непрерывной области диска. В этом случае повышается вероятность того что данные, затрагиваемые операциями выборки и обновления, будут находиться в соседних участках, а не разбросаны по всему диску.

По мере добавления данных в базу MongoDB создает дополнительные файлы данных. Размер каждого следующего файла в два раза больше размера предыдущего – и так до тех пор, пока не будет достигнут максимальный размер 2 ГБ. Таким образом, размер файла `garden.2` составит 256 МБ, файла `garden.3` – 512 МБ и так далее. Такая, достаточно распространенная, стратегия основана на предположении о том, что если объем данных растет с постоянной скоростью, то предварительные размеры файлов данных тоже должны расти. Очевидно, что при этом разность между размерами выделенного и фактически используемого места может оказаться довольно большой[5].

Команда `stats` позволяет узнать, сколько места выделено и сколько используется:

```
> db.stats()
{
   "collections" : 3,
   "objects" : 10004,
   "avgObjSize" : 36.005,
   "dataSize" : 360192,
   "storageSize" : 791296,
   "numExtents" : 7,
   "indexes" : 1,
```

[5] Это может оказаться проблемой в системах, где место на диске дефицитно. В таком случае можно воспользоваться параметрами сервера `--noprealloc` и `--smallfiles`.

```
    "indexSize" : 425984,
    "fileSize" : 201326592,
    "ok" : 1
}
```

Здесь поле `fileSize` содержит общий размер всех файлов, созданных для базы данных. Это просто сумма размеров файлов `garden.0` и `garden.1`. Разница между полями `dataSize` и `storageSize` более тонкая. Первое содержит фактический размер всех BSON-объектов в базе данных, второе включает дополнительное место, зарезервированное для роста коллекций, а также нераспределенное место, занятое удаленными объектами[6]. Наконец, поле `indexSize` показывает суммарный размер всех индексов в базе данных. Важно следить за суммарным размером индексов, поскольку оптимальная производительность достигается, когда все используемые индексы помещаются в оперативную память. Я разовью эту мысль в главах 7 и 10, когда буду говорить о диагностике и разрешении проблем, связанных с производительностью.

4.3.2. Коллекции

Коллекции – это контейнеры структурно или концептуально схожих документов. В этом разделе я подробнее расскажу о создании и удалении коллекций. А затем представлю специальные ограниченные коллекции MongoDB и на примерах покажу, как сервер работает с коллекциями.

Управление коллекциями

В предыдущем разделе вы видели, что коллекции создаются неявно – в результате вставки документов в определенное пространство имен. Но поскольку существуют разные типы коллекций, MongoDB предоставляет также специальную команду для их создания. В оболочке она вызывается следующим образом:

```
db.createCollection("users")
```

При создании стандартной коллекции можно заранее выделить для нее указанное число байтов. Обычно это не требуется, но при необходимости делается так:

```
db.createCollection("users", {size: 20000})
```

---

[6] Технически место для коллекций выделяется в файле данных блоками, которые называются *экстентами*. Поле `storageSize` показывает, сколько всего места выделено под экстенты коллекций.

Имя коллекции может содержать цифры, буквы и знак точки (.), но начинаться должно с буквы или цифры. На внутреннем уровне имя коллекции определяется именем пространства имен, которое включает также имя базы данных, которой принадлежит коллекция. Таким образом, коллекция `products` в сообщениях, отправляемых серверу или получаемых от него, называется `garden.products`. Такое полное имя коллекции не должно быть длиннее 128 символов.

Иногда полезно включать в имя коллекции знак `.` для создания своего рода виртуальных пространств имен. Например, можно представить себе коллекции с такими именами:

```
products.categories
products.images
products.reviews
```

Но имейте в виду, что это не более чем способ организации; сервер трактует коллекции с именами, содержащими точку, точно так же, как любые другие.

Об удалении документов из коллекции и уничтожении самих коллекций я уже говорил. Еще стоит упомянуть о возможности переименования коллекций. В оболочке для этого предназначен метод `renameCollection`, например:

```
db.products.renameCollection("store_products")
```

Ограниченные коллекции

Помимо стандартных, существуют так называемые *ограниченные коллекции* (capped collection). Изначально они предназначались для высокопроизводительного протоколирования. От стандартных коллекций они отличаются тем, что имеют фиксированный размер. Это означает, что по достижении максимального размера последующие вставки в ограниченную коллекцию затирают самые старые хранящиеся в ней документы. Поэтому пользователь избавлен от необходимости вручную урезать коллекцию в случае, когда интерес представляют только недавние данные.

Чтобы понять, как можно использовать ограниченные коллекции, представьте, что требуется отслеживать действия пользователей на сайте: просмотр товара, добавление в корзину, оформление покупки, оплату и т. д. Можно написать скрипт, который будет имитировать протоколирование этих действий в ограниченной коллекции. Он позволит наблюдать кое-какие интересные свойства таких коллекций. Вот простой пример для демонстрации.

Листинг 4.6. Имитация протоколирования действий пользователей в ограниченной коллекции

```
require 'rubygems'
require 'mongo'

VIEW_PRODUCT = 0
ADD_TO_CART  = 1
CHECKOUT     = 2
PURCHASE     = 3

@con = Mongo::Connection.new
@db = @con['garden']

@db.drop_collection("user.actions")

@db.create_collection("user.actions", :capped => true, :size => 1024)

@actions = @db['user.actions']

40.times do |n|
   doc = {
      :username => "kbanker",
      :action_code => rand(4),
      :time => Time.now.utc,
      :n => n
   }

   @actions.insert(doc)
end
```

Сначала мы создаем ограниченную коллекцию `users.actions` размером 1 КБ с помощью метода `DB#create_collection`[7]. Затем вставляем 20 документов в журнал. Каждый документ содержит имя пользователя, код действия (целое число от 0 до 3) и временную метку. Мы включили еще порядковый номер *n*, чтобы можно было сказать, какие документы устарели. Теперь опросим эту коллекцию из оболочки:

```
> use garden
> db.user.actions.count();
10
```

Хотя мы вставили 20 документов, в коллекции их оказалось только 10. Если запросить список документов, то станет понятно, почему:

---

[7] В оболочке эквивалентная команда – `db.createCollection("users.actions", {capped: true, size: 1024})`

```
db.user.actions.find();
{ "_id" : ObjectId("4c55f6e0238d3b201000000b"), "username" : "kbanker",
"action_code" : 0, "n" : 10, "time" : "Sun Aug 01 2010 18:36:16" }
{ "_id" : ObjectId("4c55f6e0238d3b201000000c"), "username" : "kbanker",
"action_code" : 4, "n" : 11, "time" : "Sun Aug 01 2010 18:36:16" }
{ "_id" : ObjectId("4c55f6e0238d3b201000000d"), "username" : "kbanker",
"action_code" : 2, "n" : 12, "time" : "Sun Aug 01 2010 18:36:16" }
...
```

Документы возвращаются в том порядке, в котором вставлялись Глядя на значения n, можно сделать вывод, что самый ранний доку мент – десятый из вставленных, то есть документы с номерами от 0 дс 9 уже устарели. Поскольку максимальный размер этой ограниченной коллекции – 1024 байта, то можно заключить, что длина каждого до кумента составляет примерно 100 байтов. О том, как подтвердить эт предположение, мы узнаем в следующем разделе.

Но сначала я хотел бы отметить еще два различия между огра ниченными и стандартными коллекциями. Во-первых, по умолчани над ограниченной коллекцией не создается индекс по полю _id. Эт оптимизация во имя производительности; без индекса вставка зани мает меньше времени. Если индекс по _id необходим, можете пос троить его вручную. Ограниченную коллекцию без индекса удобн представлять себе как структуру данных, доступ к которой произ водится последовательно, а не в произвольном порядке. Имея это виду, MongoDB предоставляет специальный оператор сортировки который возвращает документы в естественном порядке вставки[8]. В предыдущем запросе элементы коллекции возвращались в прямом естественном порядке. Чтобы получить их в обратном естественном порядке, следует использовать оператор сортировки $natural:

```
> db.user.actions.find().sort({"$natural": -1});
```

Помимо естественного упорядочения документов и отсутствия индекса, ограниченные коллекции налагают некоторые ограничения на операции CRUD. Во-первых, запрещается удалять из ограничен ной коллекции отдельные документы, а, во-вторых, нельзя выпол нять операции обновления, приводящие к увеличению размера до кумента[9].

---

[8] Естественным называется порядок, в котором документы хранятся на диске.

[9] Поскольку изначально ограниченные коллекции были задуманы для протоколиро вания, удалять и обновлять документы считалось ненужным, так как это усложнил бы код, отвечающий за устаревание документов. Ценой отказа от этих возможнос тей удалось сохранить исходную простоту и эффективность ограниченных коллек ций.

Системные коллекции

Дизайн MongoDB отчасти основан на внутреннем использовании коллекций для собственных целей. Поэтому всегда существуют две специальные системные коллекции: `system.namespaces` и `system.indexes`. Запрос к первой дает все пространства имен, определенные в текущей базе данных:

```
> db.system.namespaces.find();
{ "name" : "garden.products" }
{ "name" : "garden.system.indexes" }
{ "name" : "garden.products.$_id_" }
{ "name" : "garden.user.actions", "options" :
      { "create": "user.actions", "capped": true, "size": 1024 } }
```

Во второй коллекции, `system.indexes`, хранятся определения всех индексов в текущей базе данных. Так, чтобы получить список индексов в базе данных `garden`, нужно выполнить следующий запрос:

```
> db.system.indexes.find();
{ "name" : "_id_", "ns" : "garden.products", "key" : { "_id" : 1 } }
```

Коллекции `system.namespaces` и `system.indexes` стандартные, но MongoDB также использует ограниченные коллекции для репликации. Любой член набора реплик протоколирует все выполненные им операции записи в специальной ограниченной коллекции `oplog.rs`. Вторичные узлы читают эту коллекцию последовательно и применяют хранящиеся в ней операции к своим базам. Более подробно мы обсудим системные коллекции в главе 9.

4.3.3. Документы и вставка

В завершение этой главы мы сообщим некоторые детали о вставке документов.

Сериализация документов, типы и ограничения

В предыдущей главе отмечалось, что перед отправкой серверу MongoDB любой документ необходимо сериализовать в формат BSON; затем драйвер производит дересериализацию результата из формата BSON в представление, специфичное для конкретного языка. Большинство драйверов предлагают простой интерфейс для сериализации и десериализации; знать, как он работает, полезно на случай, если вы захотите посмотреть, что именно посылается серверу. Например, рассказывая об ограниченных коллекциях, мы сделали разумное предположение о том, что размер документа составляет примерно

100 байтов. Это предположение легко проверить, воспользовавшись BSON-сериализатором, встроенным в драйвер Ruby:

```
doc = {
   :_id => BSON::ObjectId.new,
   :username => "kbanker",
   :action_code => rand(5),
   :time => Time.now.utc,
   :n => 1
}

bson = BSON::BSON_CODER.serialize(doc)

puts "Документ #{doc.inspect} занимает #{bson.length} байтов в BSON"
```

Метод `serialize` возвращает массив байтов. Выполнив приведенный выше код, вы получите BSON-объект длиной 82 байта, что несильно отличается от нашей оценки. Узнать размер BSON-объекта в оболочке тоже несложно:

```
> doc = {
   _id: new ObjectId(),
   username: "kbanker",
   action_code: Math.ceil(Math.random() * 5),
   time: new Date(),
   n: 1
}

> Object.bsonsize(doc);
82
```

Те же 82 байта. Разность между 82 и 100 байтами обусловлена обычными для хранения коллекции и документа накладными расходами.

Десериализация BSON-объекта производится так же просто. Попробуйте выполнить следующий код:

```
deserialized_doc = BSON::BSON_CODER.deserialize(bson)
puts "Это наш документ, десериализованный из BSON:"
puts deserialized_doc.inspect
```

Обратите внимание, что произвольный хеш Ruby сериализовать не получится. Чтобы сериализация завершилась успешно, имена ключей должны быть допустимы, а для любого значения должно существовать преобразование в какой-то тип BSON. В качестве имени ключа допустима строка, завершающаяся нулем, длиной не более 255 байтов. Строка может состоять из любых ASCII-символов с тремя ограничениями: она не должна начинаться символом `$`, не должна содержать символов `.` и может включать нулевой байт только

в последней позиции. При программировании на Ruby в качестве ключей хеша можно использовать символы, но на этапе сериализации они будут преобразованы в эквивалентные строки.

Не стоит выбирать длинные имена ключей, поскольку они хранятся в самом документе. В этом проявляется отличие от РСУБД, где имена столбцов хранятся отдельно от строк с данными. Поэтому, если при работе с BSON выбрать имя ключа `dob` вместо `date_of_birth`, то экономия на каждом документе составит 10 байтов. Вроде мелочь, но если таких документов миллиард, то только за счет выбора более короткого имени ключа удастся сэкономить почти 10 ГБ дискового пространства. Это, конечно, не значит, что нужно из кожи вон лезть ради коротких имен, – проявляйте здравый смысл. Тем не менее, если ожидается, что массив данных будет велик, то укорачивание имени ключа поможет сэкономить место.

Кроме того, все значения в документе должны допускать преобразование в один из типов BSON. Таблица типов BSON, с примерами и примечаниями, имеется на сайте http://bsonspec.org. Здесь я лишь отмечу некоторые важные моменты и подводные камни.

Строки

Все строковые значения должны быть представлены в кодировке UTF-8. Хотя она стала уже почти стандартной, в ряде случаев по-прежнему используются старые кодировки. В связи с этим у пользователей часто возникают проблемы при импорте в MongoDB данных из унаследованных систем. Решение, как правило, заключается в том, чтобы конвертировать данные в UTF-8 перед вставкой, а, если это затруднительно, то сохранить текст в виде двоичного типа BSON.[10]

Числа

В BSON определены три числовых типа: `double`, `int` и `long`. Поэтому можно представить любое число с плавающей точкой в формате IEEE и любое целое со знаком длиной до 8 байтов. При сериализации целых в динамическом языке драйвер автоматически выбирает между `int` и `long`. На самом деле, существует всего одна часто встречающаяся ситуация, когда тип числа нужно задавать явно, – вставка числовых данных в JavaScript-оболочке. К сожалению, JavaScript поддерживает только один числовой тип `Number`, эквивалентный числу с двойной точностью в формате IEEE. Поэтому если требуется

[10] Кстати, если вы незнакомы с кодировками символов, то просто обязаны прочесть широко известное введение Джоэла Сполски (http://mng.bz/LVO6). А если вы при этом пишете на Ruby, то стоит еще почитать серию статей Джеймса Эдварда Грея о кодировках в Ruby 1.8 и 1.9 (http://mng.bz/wc4I).

в оболочке сохранить целое число, то нужно явно воспользоваться одним из классов `NumberLong()` или `NumberInt()`. Выполните следующий пример:

```
db.numbers.save({n: 5});
db.numbers.save({ n: NumberLong(5) });
```

Вы сохранили два документа в коллекции `numbers`. Но, хотя числовые значения были одинаковы, в первом случае число сохранено как `double`, а во втором – как `long`. Запрос всех документов, в которых `n` равно 5, возвращает оба документа:

```
> db.numbers.find({n: 5});
{ "_id" : ObjectId("4c581c98d5bbeb2365a838f9"), "n" : 5 }
{ "_id" : ObjectId("4c581c9bd5bbeb2365a838fa"), "n" : NumberLong( 5 ) }
```

Однако второе значение, как видите, помечено как длинное целое. По-другому в этом можно убедиться, предъявив запрос по типу BSON с помощью специального оператора `$type`. Каждый тип BSON идентифицируется целым числом, начиная с 1. Спецификация BSON на сайте http://bsonspec.org говорит, что числам с двойной точностью соответствует тип 1, а 64-разрядным целым – тип 18. Следовательно, запрос по типу можно сформулировать следующим образом:

```
> db.numbers.find({n: {$type: 1}});
{ "_id" : ObjectId("4c581c98d5bbeb2365a838f9"), "n" : 5 }

> db.numbers.find({n: {$type: 18}});
{ "_id" : ObjectId("4c581c9bd5bbeb2365a838fa"), "n" : NumberLong( 5 ) }
```

Таким образом, подтверждается различие в способе хранения. Вряд ли вам доведется использовать оператор `$type` в реальных программах, но для отладки он, как видите, весьма полезен.

И еще одна проблема, которая часто возникает в связи с числовыми типами BSON, – отсутствие поддержки для типа `decimal`. Таким образом, если вы планируете хранить в MongoDB денежные величины, то придется использовать целый тип и представлять значения в центах (или копейках).

Дата и время

Тип `datetime` используется в BSON для хранения временных значений. Время представляется 64-разрядным целым числом, равным количеству миллисекунд от «эпохи» Unix в часовом поясе UTC (мировое координированное время). Отрицательное значение означает момент времени до «эпохи»[11].

[11] Эпохой Unix называется полночь 1 января 1970 года по мировому координированному времени.

Уместно сделать два замечания. Во-первых, записывая даты в JavaScript, не забывайте, что в этом языке нумерация месяцев начинается с 0. Это означает, что объект `new Date(2011, 5, 11)` представляет 11 *июня* 2011 года. Во-вторых, BSON-сериализатор в драйвере Ruby ожидает, что время представлено объектом `Time` в UTC. Следовательно, нельзя использовать классы, поддерживающие часовые пояса, так как в BSON-типе `datetime` эту информацию невозможно закодировать.

Специальные типы

Но что если требуется сохранить время вместе с часовым поясом? Иногда базовых типов BSON недостаточно. И хотя не существует способа создать нестандартный тип BSON, можно получить собственный виртуальный тип, комбинируя значения примитивных типов. Например, сохранить время с часовым поясом можно в документе, который на Ruby выглядит так:

```
{:time_with_zone =>
   {  :time => Time.utc.now,
      :zone => "EST"
   }
}
```

Нетрудно написать приложение, которое будет прозрачно для пользователя обрабатывать такие составные представления. Именно так это обычно и делается в реальных программах. Например, MongoMapper – написанная на Ruby система объектного отображения для MongoDB – позволяет определить для любого объекта методы `to_mongo` и `from_mongo`, способные адаптироваться к такого рода составным типам.

Ограничения на размер документа

В MongoDB версии 2.0 размер BSON-документа ограничен 16 МБ[12]. Ограничение существует по двум взаимосвязанным причинам. Во-первых, чтобы предотвратить создание чрезмерно громоздких моделей данных. Хотя плохую модель можно придумать и не выходя за рамки ограничения в 16 МБ, оно все же не позволит создать документы с несуразно глубокой вложенностью – ошибка, которую часто допускают новички. С такими документами трудно работать, часто лучше поместить внутренние документы в отдельные коллекции.

[12] Эта величина зависит от версии сервера и от версии к версии увеличивается. Чтобы узнать, какое ограничение действует для вашего сервера, запустите из оболочки скрипт `db.ismaster` и посмотрите на поле `maxBsonObjectSize`. Если такого поля нет, значит ограничение составляет 4 МБ (а вы работаете с очень старой версией MongoDB).

Вторая причина ограничения в 16 МБ связана с производительностью. Для обработки запроса к большому документу этот документ следует скопировать в буфер в памяти сервера перед отправкой клиенту. Такое копирование обходится дорого, особенно (как то часто бывает на практике) если клиенту весь документ не нужен[13]. Кроме того отправленный документ еще предстоит передать по сети и десериализовать в драйвере. Это становится особенно накладно в случае, когда запрашивается сразу несколько многомегабайтных документов.

Вывод таков: очень большие объекты целесообразно разбивать на части, для чего следует изменить модель данных, заведя дополнительные коллекции. Но отметим, что с простым хранением больших двоичных объектов, например изображений или видео, дело обстоит несколько иначе. О том, как с ними обращаться, рассказано в приложении С.

Массовая вставка

Если имеются корректные документы, то вставить их нетрудно. По большей части информация о деталях вставки документов, в том числе о генерации идентификаторов объектов, о действиях на сетевом уровне и о безопасном режиме, приведена в главе 3. Но один оставшийся вопрос – о массовой вставке – мы обсудим здесь.

Любой драйвер позволяет вставлять сразу несколько документов. Особенно это удобно, когда требуется вставить сразу много данных, как, например, при начальном импорте или переходе с другой СУБД. Вспомните приведенный выше пример вставки 40 документов в коллекцию `user.actions`. Взглянув на код, вы увидите, что мы вставляли документы по одному. А в следующем фрагменте сначала строится массив из 40 документов, а потом целиком передается методу `insert`:

```
docs = (0..40).map do |n|
   {  :username => "kbanker",
      :action_code => rand(5),
      :time => Time.now.utc,
      :n => n
   }
end

@col = @db['test.bulk.insert']
@ids = @col.insert(docs)
puts "Ид документов после массовой вставки: #{@ids.inspect}"
```

[13] В следующей главе мы увидим, что всегда можно указать, какие поля документа возвращать, и тем самым уменьшить размер ответа. Но если вы часто так поступаете, то имеет смысл пересмотреть модель данных.

В случае массовой вставки метод `insert` возвращает не один идентификатор объекта, а массив, содержащий идентификаторы всех вставленных документов. Пользователи часто спрашивают, каков оптимальный размер порции вставляемых документов, но это зависит от стольких факторов, что дать точный ответ невозможно. Скажем лишь, что оптимальное количество находится в промежутке от 10 до 200. В данном случае следует положиться на эталонное тестирование. Сервер налагает только одно ограничение – суммарный объем любой операции вставки не должен превышать 16 МБ. Опыт показывает, что наиболее эффективная вставка далеко не достигает этого предела.

4.4. Резюме

В этой главе мы изучили обширный материал. Примите поздравления, что забрались так далеко!

Мы начали с теоретического обсуждения вопроса о проектировании схемы, а затем перешли к разработке наброска модели данных для интернет-магазина. На этом примере вы поняли, как могут выглядеть документы в производственной системе, и получили пищу для более конкретных размышлений о различиях между схемами в РСУБД и MongoDB.

В конце главы мы более подробно познакомились с базами данных, коллекциями и документами; возможно, вы еще не раз будете возвращаться к этому разделу для получения справочной информации. Я рассказал только о началах MongoDB, по-настоящему мы еще никуда не продвинулись. Но все изменится в следующей главе, где мы будем изучать мощь произвольных запросов.

ГЛАВА 5. Запросы и агрегирование

В этой главе:

- Запросы к модели данных электронного магазина.
- Подробнее о языке запросов MongoDB.
- Агрегирование с помощью распределения-редукции и группировки.

В MongoDB язык SQL не используется. Его место занимает JSON-подобный язык запросов. Мы уже встречались с ним на страницах этой книги, а теперь обратимся к более содержательным примерам, характерным для реальных приложений. Конкретно, мы вернемся к модели данных интернет-магазина, описанной в предыдущей главе, и продемонстрируем различные запросы к ней: поиск по `_id`, запросы по диапазону, упорядочение и проецирование. Мы также дадим обзор языка запросов MongoDB в целом и детально рассмотрим все имеющиеся в нем операторы.

К запросам вплотную примыкает тема агрегирования. Запросы позволяют получить данные в том виде, в каком они хранятся в базе, а агрегатные функции суммируют и изменяют форму данных. Сначала мы покажем, как производится агрегирование набора данных для интернет-магазина, уделив особое внимание функциям группировки и распределения-редукции в MongoDB. Затем будет приведено полное справочное руководство по этим функциям.

Имейте в виду, что пока вы читаете эту главу, работа над языком запросов и агрегатными функциями в MongoDB продолжается, и в каждой версии появляются новые возможности. На данный момент овладение запросами и агрегированием в MongoDB означает не

столько знание всех закоулков языка, сколько умение находить оптимальный способ решения повседневных задач. В этой главе я на примерах продемонстрирую самые прямые пути. К концу главы вы будете интуитивно чувствовать, как ведут себя запросы и агрегирование в MongoDB, сможете применить полученные знания к проектированию схемы приложения.

5.1. Запросы в приложении для интернет-магазина

В этом разделе мы продолжим исследование модели данных интернет-магазина, начатое в предыдущей главе. Мы определили структуру документов, описывающих товары, категории, пользователей, заказы и обзоры. А теперь посмотрим, как опрашивать эти сущности в типичном приложении для электронной торговли. Некоторые запросы совсем просты. Например, поиск по полю `_id` для вас уже не тайна за семью печатями. Но мы рассмотрим и более сложные случаи, например опрос и отображение иерархии категорий, а также фильтрацию списков товаров. Не будем также забывать о производительности и покажем, какие индексы способны увеличить скорость обработки некоторых запросов.

5.1.1. Товары, категории и обзоры

В большинстве интернет-магазинов есть по меньшей мере два визуальных представления товаров и категорий. Первое – начальная страница товара, на которой описывается сам товар, приводятся обзоры и информация о том, к каким категориям относится товар. Второе – список товаров, где пользователь может переходить из одной категории в другую и видеть миниатюры всех товаров в выбранной категории. Начнем со страницы товара, которая во многих отношениях проще.

Допустим, что URL-адреса страниц товаров построены на основе краткого названия. В таком случае получить все данные, необходимые для построения страницы товара, можно с помощью трех запросов:

```
db.products.findOne({'slug': 'wheel-barrow-9092'})
db.categories.findOne({'_id': product['main_cat_id']})
db.reviews.find({'product_id': product['_id']})
```

Первый находит товар с кратким названием `wheel-barrow-9092`. Имея товар, можно получить информацию о его категории с помощью простого запроса по полю `_id` к коллекции `categories`. Нако-

нец, последний запрос извлекает все обзоры, относящиеся к данному товару.

Обратите внимание, что в двух первых запросах используется метод find_one, а в последнем – метод find. Эти методы предоставляет любой драйвер MongoDB, и было бы неплохо напомнить, в чем разница между ними. В главе 3 мы говорили, что find возвращает объект курсора, тогда как findOne – документ. Приведенный выше вызов метода findOne эквивалентен следующему:

```
db.products.find({'slug': 'wheel-barrow-9092'}).limit(1)
```

Если вы ожидаете получить только один документ, то findOne вернет его, если он, конечно, существует. Если же требуется вернуть несколько документов, то не забывайте, что нужно использовать метод find и что этот метод возвращает курсор. Затем нужно будет где-то в приложении этот курсор обойти.

Теперь вернемся к запросам на странице описания товара. Вас ничего не раздражает? Если вам кажется, что в запросе на получение отзывов не хватает ограничений, то вы правы. Этот запрос отбирает все отзывы о данном товаре, что вряд ли годится, когда таких отзывов сотни. В большинстве приложений множество отзывов разбивается на страницы, и для этой цели в MongoDB предусмотрены параметры `skip` и `limit`. С их помощью выделить одну страницу множества документов с отзывами можно так:

```
db.reviews.find({'product_id': product['_id']}).skip(0).limit(12)
```

Кроме того, хорошо бы отображать отзывы в каком-то определенном порядке, то есть отсортировать результаты запроса, например, по количеству проголосовавших за полезность отзыва. Это делается так:

```
db.reviews.find({'product_id': product['id']}).sort(
   {helpful_votes: -1}).limit(12)
```

Здесь мы просим MongoDB вернуть первые 12 отзывов, отсортировав их по количеству голосов в порядке убывания. Имея в своем распоряжении возможности сортировки, пропуска и ограничения результата, остается решить, а надо ли вообще разбивать на страницы. Для этого можно запросить общее количество документов, а затем определить, какая страница отзывов нужна. Вот теперь с запросами, необходимыми для формирования страницы товара, все стало ясно:

```
product = db.products.findOne({'slug': 'wheel-barrow-9092'})
category = db.categories.findOne({'_id': product['main_cat_id']})
reviews_count = db.reviews.count({'product_id': product['_id']})
```

```
reviews = db.reviews.find({'product_id': product['_id']}).
                          skip((page_number - 1) * 12).
                          limit(12).
                          sort({'helpful_votes': -1})
```

Для таких запросов необходимы индексы. С краткими названиями мы уже разобрались – это альтернативные первичные ключи, поэтому по ним нужно строить уникальный индекс. Кроме того, как вы помните, для всех стандартных коллекций автоматически строится индекс по полю `_id`. Однако еще нужно построить индексы по всем полям, которые используются как внешние ссылки. В данном случае это поля `user_id` и `product_id` в коллекции обзоров.

Разобравшись с запросами на странице товара, можно заняться страницей с перечнем товаров. На ней отображается список товаров, принадлежащих одной категории. Здесь же имеются ссылки на родительскую и соседние категории.

Поскольку страница перечня товаров определяется категорией, то в запросах будет присутствовать краткое название категории:

```
category = db.categories.findOne({'slug': 'outdoors'})
siblings = db.categories.find({'parent_id': category['_id']})
products = db.products.find({'category_id': category['_id']}).
                          skip((page_number - 1) * 12).
                          limit(12).
                          sort({helpful_votes: -1})
```

Соседними называются категории с одинаковым идентификатором родителя, поэтому запрос на поиск соседей формулируется очень просто. Поскольку в документе о товаре содержится массив идентификаторов категорий, то задача нахождения всех товаров в данной категории тоже тривиальна. Для разбиения на страницы можно применить ту же технику, что для отзывов, только теперь сортировка производится по усредненному рейтингу товаров. Можно представить себе и другие способы сортировки (по названию, по цене и т. д.). Достаточно просто изменить поле сортировки[1].

Существует частный случай страницы списка товаров – просмотр корневых категорий, в которых нет товаров. В этом случае достаточно запросить категории с идентификатором родителя `nil`:

```
categories = db.categories.find({'parent_id': nil})
```

[1] Необходимо продумать вопрос об эффективности сортировки. Можно положиться на помощь имеющегося индекса, но это означает, что при добавлении новых вариантов сортировки растет и число индексов, и в конечном итоге затраты на обслуживание этих индексов могут превысить разумные пределы. В особенности это относится к ситуации, когда количество товаров в каждой категории невелико. В главе 8 мы вернемся к этой теме, но уже сейчас имеет смысл задуматься о компромиссах.

5.1.2. Пользователи и заказы

Запросы в предыдущем разделе ограничивались поиском и сортировкой. При рассмотрении пользователей и заказов мы копнем глубже, поскольку понадобится получить некоторые отчеты о заказах.

Но начнем с более простого вопроса – аутентификации пользователей. При входе в приложение пользователь указывает свое имя и пароль. Поэтому можно ожидать, что часто будет встречаться такой запрос:

```
db.users.findOne({username: 'kbanker',
      hashed_password: 'bd1cfa194c3a603e7186780824b04419'})
```

Если такой пользователь существует и пароль правилен, то будет возвращен весь документ с описанием пользователя, в противном случае запрос не вернет ничего. Ничего страшного в таком запросе нет. Но если вы хотите добиться максимальной производительности, то стоит немного оптимизировать – выбирать только поле `_id`, необходимое для создания сеанса. Вспомните, в документе о пользователе хранится адрес, метод платежа и прочие персональные данные. Зачем передавать всё это по сети и десериализовывать в драйвере, если нужно всего одно поле? Для выборки части полей применяется операция проецирования:

```
db.users.findOne({username: 'kbanker',
      hashed_password: 'bd1cfa194c3a603e7186780824b04419'},
      {_id: 1})
```

Теперь ответ содержит только взятое из документа поле `_id`:

```
{ _id: ObjectId("4c4b1476238d3b4dd5000001") }
```

К коллекции `users` могут предъявляться и другие запросы. Например, в административном интерфейсе часто требуется искать пользователей по различным критериям, например, по полю `last_name`:

```
db.users.find({last_name: 'Banker'})
```

Так делать можно, но найдены будут только точные соответствия. Однако не всегда известно, как пишется фамилия пользователя. В таком случае хорошо бы иметь возможность неточного поиска. Пусть, например, известно, что фамилия начинается с *Ba*. В SQL для формулирования требуемого запроса мы воспользовались бы предикатом `LIKE`:

```
SELECT * from users WHERE last_name LIKE 'Ba%'
```

Его семантическим эквивалентом в MongoDB является регулярное выражение:

```
db.users.find({last_name: /^Ba/})
```

Как и в РСУБД при выполнении префиксного поиска (как в запросе выше) может использоваться индекс[2].

Когда речь заходит о маркетинге, часто требуется искать целые группы пользователей. Например, чтобы найти всех пользователей, проживающих в Верхнем Манхэттене, можно было бы предъявить запрос по диапазону почтовых индексов:

```
db.users.find({'addresses.zip': {$gte: 10019, $lt: 10040}})
```

Напомним, что в каждом документе о пользователе хранится массив, состоящий из одного или нескольких адресов. Приведенный выше запрос найдет документ, если хотя бы в одном из хранимых адресов присутствует почтовый индекс (zip), попадающий в указанный диапазон. Чтобы запрос выполнялся эффективно, нужно построить индекс по полю `addresses.zip`.

Но группировка по месту проживания – не самый эффективный способ получить информацию для принятия решения. Гораздо более осмысленно группировать пользователей по приобретаемым ими товарам. В данном случае для этого потребуется двухэтапный запрос: сначала отбираем заказы, в которых присутствует конкретный продукт, а затем – ассоциированных с этими заказами пользователей[3]. Пусть нужно найти всех пользователей, купивших большую тачку. Снова применяем точечную нотацию для адресации элементов массива `line_items` и поиска по заданному артикулу (sku):

```
db.orders.find({'line_items.sku': "9092"})
```

Можно также ограничить результирующий набор заказами, сделанными в определенный период времени. Для этого достаточно добавить в условие запроса минимальную дату заказа:

```
db.orders.find({'line_items.sku': "9092",
      'purchase_date': {$gte: new Date(2009, 0, 1)}})
```

Если такие запросы предъявляются часто, то имеет смысл построить составной индекс по артикулу и дате покупки (именно в этом порядке). Такой индекс строится следующим образом:

```
db.orders.ensureIndex({'line_items.sku': 1, 'purchase_date': 1})
```

---

[2] Для тех, кто не знаком с регулярными выражениями, скажем, что выражение `/^Ba/` читается так: «начало строки, затем *B*, затем *a*».

[3] У имеющих опыт работы с реляционными базами данных невозможность выполнить соединение между заказами и пользователями может вызвать раздражение. Но не поддавайтесь ему. Такого рода соединения на стороне клиента в MongoDB – обычная вещь

От запроса к коллекции orders нам нужно получить только список идентификаторов пользователей. Поэтому можно повысить эффективность, воспользовавшись проецированием. В следующем фрагменте мы сначала говорим, что нас интересует только поле user_id, затем преобразуем полученный результат в простой массив идентификаторов и, наконец, используем этот массив, чтобы предъявить запрос к коллекции users с помощью оператора $in:

```
user_ids = db.orders.find({'line_items.sku': "9092",
      purchase_date: {'$gt': new Date(2009, 0, 1)}},
      {user_id: 1, _id: 0}).toArray().map(function(doc) { return doc['_id'] })
users = db.users.find({_id: {$in: user_ids}})
```

Такая техника опроса коллекций с помощью массива идентификаторов и оператора $in эффективна, если количество элементов в массиве не превышает нескольких тысяч. Для более крупных наборов данных (если тачку купили миллионы пользователей) рациональнее поместить идентификаторы пользователей во временную коллекцию и затем обработать ее последовательно.

В следующей главе вы встретитесь с другими примерами запросов к этим данным, а еще ниже узнаете, как получать из данных полезную информацию с помощью агрегатных функций MongoDB. Но и того, что мы уже знаем, достаточно для более глубокого изучения языка запросов MongoDB: мы объясним, как устроен его синтаксис в целом, и рассмотрим все операторы.

5.2. Язык запросов MongoDB

Настало время рассмотреть язык запросов MongoDB во всем его блеске. Я начну с общего описания запросов, их семантики и типов. Затем перейду к обсуждению курсоров, поскольку любой запрос в MongoDB по существу сводится к созданию курсора и выборке из него результатов. После этого я представлю классификацию имеющихся в MongoDB операторов запроса[4].

5.2.1. Селекторы запроса

Начнем с обзора селекторов запроса, обращая особое внимание на то, какие запросы можно выразить с их помощью.

---

[4] Если вы не помешаны на деталях, то раздел, посвященный классификации, при первом чтении можно спокойно пропустить.

Сопоставление с селектором

Проще всего сформулировать запрос, в котором пары ключ-значение, заданные в селекторе, точно совпадают с парами, хранящимися в искомом документе. Вот два примера:

```
db.users.find({last_name: "Banker"})
db.users.find({first_name: "Smith", age: 40})
```

Второй запрос читается так: «найти всех пользователей, для которых поле `first_name` равно Smith *и* поле `age` равно 40». Отметим, что если задано несколько пар ключ-значение, то совпадать должны все, – условия запроса неявно связываются булевским оператором AND. О том, как выразить запрос со связкой OR, рассказано ниже в разделе о булевских операторах.

Диапазоны

Часто требуется найти документы, в которых значения некоторых полей попадают в определенный диапазон. В SQL для этой цели имеются операторы <, <=, > и >=; эквивалентные операторы в MongoDB называются `$lt`, `$lte`, `$gt` и `$gte`. Мы уже встречались с ними на страницах этой книги, ведут они себя в полном соответствии с ожиданиями. Однако начинающие часто путаются, когда нужно употребить в одном запросе несколько операторов. Типичная ошибка – повторение ключа поиска:

```
db.users.find({age: {$gte: 0}, age: {$lte: 30})
```

Поскольку одинаковые ключи не могут находиться на одном и тот же уровне в одном и том же документе, то такой селектор неправилен, и применен будет лишь один из операторов задания диапазона. Правильно этот запрос записывается так:

```
db.users.find({age: {$gte: 0, $lte: 30})
```

И еще один сюрприз, относящийся к операторам диапазона, – типы. Запрос по диапазону найдет документ, только если тип хранимого значения совпадает с типом заданного[5]. Пусть, например, имеется такая коллекция документов:

```
{ "_id" : ObjectId("4caf82011b0978483ea29ada"), "value" : 97 }
{ "_id" : ObjectId("4caf82031b0978483ea29adb"), "value" : 98 }
{ "_id" : ObjectId("4caf82051b0978483ea29adc"), "value" : 99 }
{ "_id" : ObjectId("4caf820d1b0978483ea29ade"), "value" : "a" }
{ "_id" : ObjectId("4caf820f1b0978483ea29adf"), "value" : "b" }
{ "_id" : ObjectId("4caf82101b0978483ea29ae0"), "value" : "c" }
```

[5] Отметим, что числовые типы – integer, long integer и double – с точки зрения таких запросов эквивалентны.

И к ней предъявляется такой запрос:

```
db.items.find({value: {$gte: 97})
```

Если вы думаете, что запрос должен вернуть все шесть документов, потому что численными эквивалентами строк являются целые числа 97, 98 и 99, то вы ошибаетесь. Этот запрос вернет только документы, в которых хранятся целые значения. Если требуется сравнивать строки, то запрос нужно переписать так:

```
db.items.find({value: {$gte: "a"})
```

Но проблема ограничения на типы никогда не возникнет, если вы не будете в одной коллекции под одним ключом хранить значения разных типов. В общем случае это разумный совет, которому стоит последовать.

Операторы над множествами

Три оператора – `$in`, `$all` и `$nin` – принимают на входе список значений и ведут себя как предикаты. Оператор `$in` возвращает документ, если хотя бы одно значение в списке совпадает с ключом поиска. Его можно использовать, чтобы вернуть все товары, принадлежащие дискретному множеству категорий. Если список

```
[ObjectId("6a5b1476238d3b4dd5000048"),
   ObjectId("6a5b1476238d3b4dd5000051"),
   ObjectId("6a5b1476238d3b4dd5000057")
]
```

представляет категории «газонокосилки», «ручной инструмент» и «рабочая одежда», то запрос на поиск всех товаров, принадлежащих хотя бы одной из этих категорий, выглядит так:

```
db.products.find({main_cat_id: { $in:
                  [ObjectId("6a5b1476238d3b4dd5000048"),
                   ObjectId("6a5b1476238d3b4dd5000051"),
                   ObjectId("6a5b1476238d3b4dd5000057") ] } } )
```

По-другому оператор `$in` можно трактовать как своего рода булевский оператор дизъюнкции OR для сравнения с одним атрибутом. При такой трактовке предыдущий запрос можно прочитать так: «найти все товары, для которых категория равна "газонокосилки" или "ручной инструмент" или "рабочая одежда"». Но если требуется объединить по OR результаты сравнения с несколькими атрибутами, то придется воспользоваться оператором `$or`, описанным в следующем разделе.

Оператор `$in` часто применяется к спискам идентификаторов. Выше в этой главе уже встречался пример запроса с оператором `$in`, который возвращал всех пользователей, купивших конкретный товар.

Оператор `$nin` возвращает документ, только если ни один из сравниваемых элементов не совпал. Так, этот оператор можно применить для поиска всех товаров, не являющихся ни белыми, ни синими:

```
db.products.find('details.color': { $nin: ["black", "blue"] }
```

Наконец, оператор `$all` возвращает документ, если все элементы совпадают с ключом поиска. Так, его можно с успехом использовать для поиска всех товаров, имеющих одновременно теги *gift* (подарок) и *garden* (сад):

```
db.products.find(tags: { $all: ["gift", "garden"] }
```

Естественно, этот запрос имеет смысл, только если в атрибуте `tags` хранится массив строк, например:

```
{  name: "Bird Feeder",
   tags: [ "gift", "birds", "garden" ]
}
```

При использовании операторов над множествами имейте в виду, что при вычислении `$in` и `$all` могут использоваться индексы, а при вычислении `$nin` – не могут, поэтому требуется полное сканирование коллекции. Следовательно, старайтесь употреблять `$nin` в сочетании с каким-нибудь другим поисковым термом, позволяющим использовать индекс. А еще лучше попробуйте отыскать другой способ формулирования запроса. Например, можно хранить атрибут, наличие которого означает условие, эквивалентное запросу с `$nin`. Так, условие `{timeframe: {$nin: ['morning', 'afternoon']}}` можно выразить проще: `{timeframe: 'evening'}`.

Булевские операторы

В MongoDB имеются следующие булевские операторы: `$ne`, `$not`, `$or`, `$and` и `$exists`.

Оператор `$ne`, *не равно*, работает, как и ожидается. На практике лучше применять его в сочетании еще с каким-нибудь оператором, иначе запрос будет выполняться неэффективно из-за невозможности воспользоваться индексом. Например, `$ne` можно использовать для поиска всех товаров производства фирмы ACME, не имеющих тега *gardening* (садоводство):

```
db.products.find('details.manufacturer': 'ACME', tags: {$ne: "gardening"} }
```

`$ne` работает с ключами, которые указывают на одиночное значение или на массив, как показывает этот пример, где производится сравнение с массивом `tags`.

Если оператор `$ne` означает отрицание равенства с указанным значением, то `$not` означает отрицание результата какого-либо другого оператора MongoDB или сопоставления с регулярным выражением. Но прежде чем прибегать к `$not`, вспомните, что для большинства операторов имеется противоположный оператор (`$in` и `$nin`, `$gt` и `$lte` и т. д.); в таких случаях использовать `$not` не следует. Применяйте `$not` только тогда, когда у оператора или регулярного выражения нет противоположной формы. Например, чтобы найти всех пользователей, фамилии которых не начинаются с буквы *B*, вполне можно использовать `$not`:

```
db.users.find(last_name: {$not: /^B/} }
```

Оператор `$or` выражает логическую дизъюнкцию значений разных ключей. Это существенно: если речь идет о значениях одного и того же ключа, то лучше использовать `$in`. Так, тривиальный запрос на поиск товаров синего или зеленого цвета формулируется следующим образом:

```
db.products.find('details.color': {$in: ['blue', 'green']} }
```

Но для поиска товаров, которые либо синего цвета, либо произведены компанией ACME, нужно использовать `$or`:

```
db.products.find({ $or: [{'details.color': 'blue'}, 'details.manufacturer':
      'ACME'}] })
```

Оператор `$or` принимает на входе массив селекторов запроса, в котором каждый селектор может быть произвольно сложным, в частности, содержать другие операторы[6].

Как и `$or`, оператор `$and` тоже принимает массив селекторов. Так как MongoDB считает, что условия в любом селекторе, содержащем более одного ключа, соединены связкой AND, то оператор `$and` следует использовать, только когда семантику AND невозможно выразить более простым способом. Пусть, например, требуется найти все товары, помеченные одним из тегов *gift* или *holiday* и одним из тегов *gardening* или *landscaping*. Единственный способ выразить такой запрос – записать конъюнкцию двух запросов с `$in`:

---

[6] За исключением `$or`.

```
db.products.find({$and: [
         {tags: {$in: ['gift', 'holiday']}},
         {tags: {$in: ['gardening', 'landscaping']}}
      ]
   }
)
```

И напоследок обсудим оператор `$exists`. Он необходим, так как коллекции не предполагают наличия фиксированной схемы, и поэтому иногда возникает необходимость запросить документы, в которых имеется определенный ключ. Напомним, что мы планировали использовать атрибут товара `details` для хранения произвольных полей, к примеру `color` (цвет). Но если лишь для некоторых товаров заданы цвета, то можно запросить те, для которых цвета не заданы:

```
db.products.find({'details.color': {$exists: false}})
```

Можно сформулировать и противоположный запрос:

```
db.products.find({'details.color': {$exists: true}})
```

Здесь мы по сути дела проверяем существование. Но есть и другой, практически эквивалентный, способ проверки существования: сравнение атрибута со значением `null`. С применением этой техники первый запрос можно переписать так:

```
db.products.find({'details.color': null})
```

А второй – так:

```
db.products.find({'details.color': {$ne: null}})
```

Сопоставление с поддокументами

В некоторых сущностях модели данных интернет-магазина встречаются ключи, которые указывают на один вложенный объект. Хорошим примером может служить атрибут товара `details`. Вот часть документа, записанная в формате JSON:

```
{  _id: ObjectId("4c4b1476238d3b4dd5003981"),
   slug: "wheel-barrow-9092",
   sku: "9092",
   details: {
      model_num: 4039283402,
      manufacturer: "Acme",
      manufacturer_id: 432,
      color: "Green"
   }
}
```

Для запроса к таким объектам ключи разделяются точкой. Например, чтобы найти все товары, произведенные компанией Acme нужно выполнить такой запрос:

```
db.products.find({'details.manufacturer_id': 432});
```

Уровень вложенности в подобных запросах не ограничен. Допустим, что документ представлен в слегка измененном виде:

```
{  _id: ObjectId("4c4b1476238d3b4dd5003981"),
   slug: "wheel-barrow-9092",
   sku: "9092",
      details: {
         model_num: 4039283402,
         manufacturer: { name: "Acme",
                         id: 432 },
         color: "Green"
   }
}
```

Тогда ключ в первом селекторе должен содержать две точки:

```
db.products.find({'details.manufacturer.id': 432});
```

Но можно проводить сравнение не только с отдельным атрибутом поддокумента, но и с объектом в целом. Пусть, например, MongoDB используется для хранения рыночной позиции товара. Чтобы сэкономить место, мы отказываемся от стандартного идентификатора объекта и заменяем его составным ключом, который состоит из биржевого тикера и временной метки. Вот как может выглядеть типичный документ[7]:

```
{  _id: {sym: 'GOOG', date: 20101005}
   open: 40.23,
   high: 45.50,
   low: 38.81,
   close: 41.22
}
```

Теперь найти сводку для тикера GOOG за 5 октября 2010 можно с помощью следующего запроса по `_id`:

```
db.ticks.find({_id: {sym: 'GOOG', date: 20101005} });
```

Важно понимать, что при подобном сопоставлении с целым объектом производится побайтовое сравнение, а это означает, что поря

[7] В ситуации, когда требуется высокая пропускная способность, желательно свести размер документа к минимуму. Отчасти этого можно добиться за счет использования коротких названий ключей. Поэтому вместо *open* можно было бы назвать ключ просто *o*.

док ключей имеет значение. Следующий запрос не эквивалентен предыдущему и не найдет приведенный в качестве примера документ:

```
db.ticks.find({_id: {date: 20101005, sym: 'GOOG'} });
```

Хотя в JSON-документах, введенных в оболочке, порядок следования ключей сохраняется, для других языковых драйверов это может быть не так. Например, в Ruby 1.8 порядок ключей в хеше не сохраняется. Чтобы в этом случае все же сохранить порядок, нужно воспользоваться объектом класса `BSON::OrderedHash`:

```
doc = BSON::OrderedHash.new
doc['sym'] = 'GOOG'
doc['date'] = 20101005
@ticks.find(doc)
```

Не забывайте проверять, поддерживаются ли в используемом вами языке упорядоченные словари; если нет, то языковой драйвер MongoDB обязательно будет содержать альтернативную реализацию, поддерживающую упорядоченность.

Массивы

Именно массивы придают модели данных значительную часть ее выразительной мощи. Как мы видели, в массивах можно хранить строки, идентификаторы объектов и даже другие документы. Благодаря массивам мы получаем развитые, но в то же время понятные документы. Разумно будет предположить, что MongoDB должна иметь простые средства для опроса и индексирования массивов. И это действительно так: простейшие запросы к массивам выглядят как запросы к документам любого другого типа. Возьмем снова теги товаров. Они представлены в виде списка строк:

```
{  _id: ObjectId("4c4b1476238d3b4dd5003981"),
   slug: "wheel-barrow-9092",
   sku: "9092",
   tags: ["tools", "equipment", "soil"] }
```

Запрос на поиск товаров с тегом *soil* (почва) записывается тривиально, с применением того же синтаксиса, что для запроса одиночного значения:

```
db.products.find({tags: "soil"})
```

Важно отметить, что при выполнении этого запроса может использоваться индекс по полю `tags`. Если вы построите такой индекс и запустите запрос с `explain()`, то увидите, что при создании курсора используется B-дерево:

```
db.products.ensureIndex({tags: 1})
db.products.find({tags: "soil"}).explain()
```

Если требуется более точный контроль над запросом к массиву, то нотация с точкой позволяет опросить значение в конкретной позиции массива. Вот, например, как ограничить предшествующий запрос только первым из помечающих товар тегов:

```
db.products.find({'tags.0': "soil"})
```

Теги вряд ли имеет смысл опрашивать таким образом, но представьте, что речь идет об адресах пользователя. Они могут быть представлены массивом поддокументов:

```
{  _id: ObjectId("4c4b1476238d3b4dd5000001")
   username: "kbanker",
   addresses: [
      {  name: "home",
         street: "588 5th Street",
         city: "Brooklyn",
         state: "NY",
         zip: 11215},
      {  name: "work",
         street: "1 E. 23rd Street",
         city: "New York",
         state "NY",
         zip 10010},
   ]
}
```

Можно постулировать, что в нулевом элементе массива всегда хранится основной адрес доставки. Поэтому, чтобы найти всех пользователей, для которых основный адрес доставки находится в Нью-Йорке, можно было бы задать нулевую позицию и дополнительную точку для указания поля `state`:

```
db.users.find({'addresses.0.state': "NY"})
```

Можно также опустить номер позиции и задать только имя поля. Следующий запрос вернет документ о пользователе, если в Нью-Йорке находятся *все* адреса, присутствующие в списке:

```
db.users.find({'addresses.state': "NY"})
```

Как и раньше, по полю вложенного документа можно построить индекс:

```
db.users.ensureIndex({'addresses.state': 1})
```

Отметим, что одна и та же точечная нотация применяется вне зависимости от того, указывает поле на один поддокумент или на

массив поддокументов. Это очень удобно, и такая согласованность вселяет уверенность. Но в случае, когда в запросе упоминается несколько атрибутов в массиве подобъектов, может возникнуть неоднозначность. Пусть, например, требуется найти всех пользователей, домашний адрес которых находится в Нью-Йорке. Как можно было сформулировать такой запрос?

```
db.users.find({'addresses.name': 'home', 'addresses.state': 'NY'})
```

Проблема, однако, в том, что в этом запросе поля могут браться из разных адресов. Иными словами, запрос будет удовлетворен, даже если один адрес помечен как домашний («home»), а другой находится в Нью-Йорке, но мы-то хотим, чтобы оба атрибута были взяты из *одного и того же* адреса. К счастью, существует оператор, позволяющий решить эту задачу. Чтобы наложить ограничение, требующее, чтобы несколько условий относились к одному поддокументу, нужно применить оператор `$elemMatch`. Правильный запрос записывается так:

```
db.users.find({addresses: {$elemMatch: {name: 'home', state: 'NY'}}})
```

Оператор `$elemMatch` следует использовать, когда логика требует сравнивать два и более атрибутов, принадлежащих одному поддокументу.

Из операторов, относящихся к массивам, осталось обсудить только `$size`. Он позволяет опрашивать размер массива. Например, чтобы найти всех пользователей, имеющих ровно три адреса, нужно предъявить такой запрос:

```
db.users.find({addresses: {$size: 3}})
```

На момент написания этой книги оператор `$size` не позволял воспользоваться индексами и допускал только сравнение на равенство (невозможно задать диапазон размеров)[8]. Поэтому чтобы выполнять любые запросы, в которых фигурирует размер массива, следует сохранять его в специальном атрибуте документа и обновлять при любом изменении массива. Например, можно было бы добавить в документ о пользователе поле `address_length`, а затем построить по нему индекс и предъявлять любые запросы – как на точное равенство, так и по диапазону.

JavaScript

Если не удается выразить запрос с помощью описанных выше средств, то можно воспользоваться языком JavaScript. Существует

[8] Об изменениях в этом вопросе см. https://jira.mongodb.org/browse/SERVER-478

специальный оператор `$where`, который позволяет передать в запрос произвольное выражение JavaScript. В контексте JavaScript ключевое слово `this` обозначает ссылку на текущий документ. Рассмотрим такой, несколько искусственный, пример:

```
db.reviews.find({$where: "function() { return this.helpful_votes > 3; }"}))
```

Имеется также сокращенная нотация для записи подобных выражений:

```
db.reviews.find({$where: "this.helpful_votes > 3"}))
```

Хотя этот запрос работает, применять его на практике *не следует* так как эквивалентное условие легко выразить стандартными средствами языка запросов. Дело в том, что при вычислении выражений JavaScript индексы не используются, и к тому же возникают заметные накладные расходы из-за необходимости вычислять их в контексте интерпретатора JavaScript. Поэтому предъявлять JavaScript-запросы имеет смысл только тогда, когда средств стандартного языка запросов не хватает. Если такая необходимость все же возникает, то попытайтесь скомбинировать выражение JavaScript хотя бы с одним стандартным оператором. Стандартный оператор позволит уменьшить размер результирующего набора, а, значит, и количество документов загружаемых в контекст JS. На простом примере продемонстрируем когда это разумно.

Предположим, что для каждого пользователя вычислен коэффициент надежности рейтинга. Это число, которое при умножении на рейтинг пользователя дает нормализованную величину. Предположим также, что при запросе отзывов требуется возвращать только отзывы тех пользователей, для которых нормализованный рейтинг больше 3. Вот как можно было бы записать такой запрос:

```
db.reviews.find({user_id: ObjectId("4c4b1476238d3b4dd5000001"),
                 $where: "(this.rating * .92) > 3"})
```

Этот запрос отвечает обеим рекомендациям: используется стандартный запрос по предположительно индексированному полю `user_id` и JavaScript-выражение, обеспечивающее возможности, выходящие за рамки стандартного языка запросов.

Помимо потенциального снижения производительности, следует помнить также о возможности *атаки внедрением* JavaScript. Подобная атака становится возможна, если программа позволяет включать в запрос код, поступивший от пользователя. Хотя таким способом пользователь не сможет ничего записать или удалить, он теоретичес

ки получает шанс прочитать конфиденциальные данные. Абсолютно небезопасный JavaScript-запрос на Ruby мог бы выглядеть так:

```
@users.find({$where => "this.#{attribute} == #{value}"})
```

В предположении, что пользователь способен контролировать значения `attribute` и `value`, он сможет искать в коллекции документы, в которых произвольный атрибут имеет произвольное значение. Хотя это не самое страшное вторжение, всё же лучше такую возможность предотвратить.

Регулярные выражения

В начале этой главы мы видели пример использования в запросе регулярного выражения. Тогда я показал, как с помощью префиксного выражения `/^Ba/` можно найти все фамилии, начинающиеся с *Ba*, и отметил, что при выполнении такого запроса может быть использован индекс. На самом деле, возможности гораздо шире. MongoDB скомпилирована с библиотекой PCRE (http://mng.bz/hxmh), которая поддерживает богатейший арсенал регулярных выражений.

Но если не считать префиксных запросов, описанных выше, запросы с регулярными выражениями не могут использовать индексы. Поэтому к ним относится та же рекомендация, что для запросов с JavaScript-выражениями, – применять в сочетании хотя бы с одним стандартным поисковым термом. Вот, например, как сформулировать запрос на получение отзывов данного пользователя, содержащих в тексте слово *best* или *worst*. Отметим, что для сравнения без учета регистра букв можно использовать флаг[9] `i`:

```
db.reviews.find({user_id: ObjectId("4c4b1476238d3b4dd5000001"),
                 text: /best|worst/i })
```

Если в используемом языке есть встроенный тип регулярных выражений, то им также можно воспользоваться для выполнения запроса. Эквивалентный запрос на Ruby можно записать так:

```
@reviews.find({:user_id => BSON::ObjectId("4c4b1476238d3b4dd5000001"),
               :text => /best|worst/i })
```

Если объемлющий язык не поддерживает тип регулярных выражений, то к вашим услугам специальные операторы `$regex` и `$options`. Находясь в оболочке, тот же самый запрос можно записать по-другому:

---

[9] Использование флага подавления учета регистра всегда мешает воспользоваться индексами, даже в префиксных запросах.

```
db.reviews.find({user_id: ObjectId("4c4b1476238d3b4dd5000001"),
                 text: {$regex: "best|worst", $options: "i" }})
```

Прочие операторы запросов

Есть еще два оператора, которые не попадают ни в одну категорию, поэтому описываются в отдельном разделе. Первый, `$mod`, позволяет искать документы, применяя операцию деления с остатком Например, чтобы найти все заказы, в которых подитог нацело делится на 3, можно выполнить такой запрос:

```
db.orders.find({subtotal: {$mod: [3, 0]}})
```

Как видите, оператор `$mod` принимает на входе массив с двумя значениями: первое – делитель, второе – ожидаемый остаток. Таким образом технически этот запрос читается так: «найти все документы, в которых поле `subtotal` при делении на 3 дает остаток 0». Пример, конечно, надуманный, но идея из него ясна. Применяя оператор `$mod`, не забывайте что при выполнении таких запросов индекс не используется.

Второй оператор, `$type`, позволяет искать по BSON-типу значения. Я не рекомендую хранить в одной коллекций документы с разными типами значений в одном поле, но если такая необходимость все же возникнет, то у вас есть оператор для опроса типа. Недавно эта возможность пригодилась, когда я консультировал пользователя, у которого запросы по `_id` не всегда возвращали то, что нужно Проблема оказалась в том, что идентификаторы хранились иногда как строки, а иногда как настоящие идентификаторы объектов. Их BSON-типы равны 2 и 7 соответственно. Начинающий пользователь легко может не обратить внимания на эту тонкость.

Для исправления ситуации понадобилось сначала найти все документы, в которых идентификатор хранился в виде строки. Это как раз и позволяет сделать оператор `$type`:

```
db.users.find({_id: {$type: 2}})
```

5.2.2. Дополнительные средства

В любом запросе должен быть селектор, пусть даже пустой. Но, помимо него, существует целый ряд дополнительных средств, позволяющих оказать воздействие на результирующий набор. Ниже я опишу их.

Проецирование

Операция проецирования позволяет указать, какое подмножество всех полей документа включать в результирующий набор. Особенно

она полезна, когда документы велики, потому что снижает накладные расходы на передачу по сети и десериализацию. Проекция чаще всего определяется в виде набора возвращаемых полей:

```
db.users.find({}, {username: 1})
```

Этот запрос возвращает только два поля из документов о пользователях: `username` и `_id`, которое всегда включается по умолчанию.

Иногда нужно наоборот исключить некоторые поля. Например, в наших документах о пользователях хранятся адреса доставки и методы платежа, но обычно они не нужны. Чтобы исключить эти поля, добавьте их в проекцию, указав значение 0:

```
db.users.find({}, {addresses: 0, payment_methods: 0})
```

Помимо включения и исключения полей, можно также возвращать диапазон значений, хранящихся в массиве. Например, можно было бы хранить отзывы о товаре в самом документе с описанием товара. Но все равно было бы желательно разбивать множество отзывов на страницы, и для этой цели служит оператор `$slice`. Вот как выглядят запросы, возвращающие соответственно первые 12 или последние 5 отзывов:

```
db.products.find({}, {reviews: {$slice: 12}})
db.products.find({}, {reviews: {$slice: -5}})
```

Оператор `$slice` может также принимать массив из двух элементов, представляющих соответственно количество пропускаемых и верхнюю границу количества возвращаемых документов. Следующий запрос пропускает первые 24 отзыва и возвращает не более 12 отзывов:

```
db.products.find({}, {reviews: {$slice: [24, 12]}})
```

Наконец, отметим, что оператор `$slice` не налагает ограничений на состав возвращаемых полей. При необходимости делайте это явно. Например, чтобы вернуть только рейтинги отзывов, предыдущий запрос следует модифицировать так:

```
db.products.find({}, {reviews: {$slice: [24, 12]}, 'reviews.rating': 1})
```

Сортировка

Любой результирующий набор можно отсортировать по одному или нескольким полям в порядке возрастания или убывания. Простая сортировка отзывов по рейтингу в порядке убывания выглядит так:

```
db.reviews.find({}).sort({rating: -1})
```

Естественно, интереснее было бы сортировать сначала по полезности отзыва, а потом по рейтингу товара:

```
db.reviews.find({}).sort({helpful_votes:-1, rating: -1})
```

При такого рода составной сортировке порядок имеет значение. Выше мы уже отмечали, что если вводить JSON-документы в оболочке, то порядок ключей сохраняется. Но поскольку хеши Ruby не упорядочены, то в этом языке порядок сортировки задается с помощью массива массивов:

```
@reviews.find({}).sort([['helpful_votes', -1], [rating, -1]])
```

Задание сортировки в MongoDB не вызывает особых сложностей, но следует помнить о двух моментах. Во-первых, как сортировать в том порядке, в котором записи вставлялись; для этого служит оператор `$natural`, который обсуждался в главе 4. Во-вторых, – и это более существенно – как гарантировать использование для сортировки индексов. Мы вернемся к этой теме в главе 8, но если вам уже сейчас невтерпеж воспользоваться сортировкой, можете забежать вперед.

Пропуск и ограничение

Никакой тайны в семантике параметров `skip` и `limit` нет. Они работают в полном соответствии с ожиданиями.

Однако следует знать об особенностях задания большого (скажем, более 10 000) значения `skip` – для выполнения таких запросов сервер должен просканировать столько документов, сколько указано в `skip`. Пусть, например, требуется разбить множество, состоящее из миллиона документов, отсортированное по убыванию даты, на страницы по 10 результатов. Это означает, что для отображения страницы с номером 50 000 придется пропустить 500 000 документов, что чудовищно неэффективно. Гораздо лучше будет вообще опустить параметр `skip`, а вместо него добавить в запрос условие на диапазон, которое говорит, где начинается следующий результирующий набор. Таким образом, запрос

```
db.docs.find({}).skip(500000).limit(10).sort({date: -1})
```

превращается в такой:

```
db.docs.find({date: {$gt: previous_page_date}}).limit(10).sort({date: -1})
```

Второй запрос будет сканировать гораздо меньше документов, чем первый. Правда, есть одна сложность: если даты `date` не уникаль-

ны, то один и тот же документ может быть показан несколько раз. Существует много способов решения этой проблемы, но мы оставим это в качестве упражнения для читателя.

5.3. Агрегирование заказов

Простой пример агрегирования в MongoDB вы уже видели при обсуждении команды `count`, которая применялась для разбиения на страницы. В большинстве СУБД имеется как функция `count`, так и много других встроенных агрегатных функций для вычисления суммы, среднего, дисперсии и т. п. Всё это включено в планы развития MongoDB, но пока они не еще не воплощены в жизнь, для реализации любой агрегатной функции – от вычисления простой суммы до стандартного отклонения – можно использовать команды `group` и `map-reduce`.

5.3.1. Группировка отзывов по пользователям

Часто бывает нужно знать, какие пользователи давали наиболее полезные отзывы. Поскольку приложение позволяет голосовать за отзывы, то технически возможно подсчитать общее число голосов, отданных за все отзывы одного пользователя, а также усреднить число голосов по всем отзывам каждого пользователя. Для сбора этой статистики можно отобрать все отзывы и произвести несложную обработку на стороне клиента. А можно и воспользоваться командой MongoDB `group`, чтобы получить результат от сервера.

Команда `group` принимает как минимум три аргумента. Первый, `key`, определяет, как будут группироваться данные. Мы хотим сгруппировать результаты по пользователям, поэтому ключом группировки будет `user_id`. Второй аргумент называется функцией `reduce`; это написанная на JavaScript функция, которая агрегирует результирующий набор. Последний аргумент `group` – начальный документ, передаваемый функции `reduce`.

Звучит все это сложнее, чем есть на самом деле. Чтобы разобраться, приглядимся внимательнее к начальному документу и соответствующей ему функции `reduce`:

```
initial = {review: 0, votes: 0};

reduce = function(doc, aggregator) {
   aggregator.reviews += 1.0;
```

```
    aggregator.votes += doc.votes;
}
```

Как видите, в начальном документе определены значения, которые мы хотим получить для каждого ключа группировки. Иначе говоря, после завершения `group` мы ожидаем получить результирующий набор, в котором для каждого `user_id` будет представлено общее число написанных им отзывов и сумма голосов, отданных за все эти отзывы. Эти величины и вычисляются функцией `reduce`. Допустим, что я написал пять отзывов. Это означает, что пять документов с отзывами будут помечены моим идентификатором пользователя. Каждый из пяти документов будет передан функции `reduce` в виде аргумента `doc`. В самом начале в качестве аргумента `aggregator` передается документ `initial`. По мере обработки каждого документа значения в агрегаторе увеличиваются.

Вот как можно выполнить команду `group` из JavaScript-оболочки.

Листинг 5.1. Использование команды MongoDB `group`

```
results = db.reviews.group({
    key:      {user_id: true},
    initial: {reviews: 0, votes: 0.0},
    reduce:  function(doc, aggregator) {
                              aggregator.reviews += 1;
                              aggregator.votes += doc.votes;
    }
    finalize: function(doc) {
                              doc.average_votes = doc.votes / doc.reviews;
    }
})
```

Вы, наверное, обратили внимание на дополнительный аргумент, переданный `group`. Мы поставили задачу вычислить среднее число голосов, поданных за отзыв. Но среднее можно найти только после того, как известны общее число голосов за все отзывы и общее число отзывов. Именно для этого и предназначен финализатор. Это JavaScript-функция, которая применяется к каждому результату группировки перед тем, как команда `group` вернет управление. В данном случае финализатор усредняет число голосов по отзывам.

Ниже представлены результаты запуска агрегирования для тестового набора данных.

Листинг 5.2. Результат выполнения команды `group`

```
[
    {  user_id: ObjectId("4d00065860c53a481aeab608"),
```

```
    aggregator.votes += doc.votes;
}
```

Как видите, в начальном документе определены значения, которые мы хотим получить для каждого ключа группировки. Иначе говоря, после завершения `group` мы ожидаем получить результирующий набор, в котором для каждого `user_id` будет представлено общее число написанных им отзывов и сумма голосов, отданных за все эти отзывы. Эти величины и вычисляются функцией `reduce`. Допустим, что я написал пять отзывов. Это означает, что пять документов с отзывами будут помечены моим идентификатором пользователя. Каждый из пяти документов будет передан функции `reduce` в виде аргумента `doc`. В самом начале в качестве аргумента `aggregator` передается документ `initial`. По мере обработки каждого документа значения в агрегаторе увеличиваются.

Вот как можно выполнить команду `group` из JavaScript-оболочки.

Листинг 5.1. Использование команды MongoDB `group`

```
results = db.reviews.group({
    key:      {user_id: true},
    initial: {reviews: 0, votes: 0.0},
    reduce:  function(doc, aggregator) {
                              aggregator.reviews += 1;
                              aggregator.votes += doc.votes;
    }
    finalize: function(doc) {
                              doc.average_votes = doc.votes / doc.reviews;
    }
})
```

Вы, наверное, обратили внимание на дополнительный аргумент, переданный `group`. Мы поставили задачу вычислить среднее число голосов, поданных за отзыв. Но среднее можно найти только после того, как известны общее число голосов за все отзывы и общее число отзывов. Именно для этого и предназначен финализатор. Это JavaScript-функция, которая применяется к каждому результату группировки перед тем, как команда `group` вернет управление. В данном случае финализатор усредняет число голосов по отзывам.

Ниже представлены результаты запуска агрегирования для тестового набора данных.

Листинг 5.2. Результат выполнения команды `group`

```
[
    {  user_id: ObjectId("4d00065860c53a481aeab608"),
```

Прежде всего, вспомните, что переменная this всегда ссылается на текущий обрабатываемый документ. В первой строке мы получаем целое число, обозначающее месяц, в котором был создан заказ[10]. Затем вызывается функция emit(). Это специальный метод, который должна вызывать любая функция распределения. Первым аргументом emit() является ключ группировки, вторым – обычно документ, содержащий редуцируемые значения. В данном случае мы группируем по месяцам, а редуцировать собираемся подитог и количество позиций из каждого заказа. Код функции reduce поможет разобраться в том, что происходит:

```
reduce = function(key, values) {
   var tmpTotal = 0;
   var tmpItems = 0;

   tmpTotal += doc.order_total;
   tmpItems += doc.items_total;

   return ( {total: tmpTotal, items: tmpItems} );
}
```

Функции reduce передается ключ и массив, содержащий одно или несколько значений. При написании функции редукции вы должны агрегировать эти значения желаемым способом и вернуть единственное значение. Из-за итеративной природы механизма map-reduce функция reduce может вызываться несколько раз, и ваша программа должна быть к этому готова. На практике это означает, что значение возвращаемое reduce, по структуре должно быть идентично значению, которое порождает (emit) функция map. Присмотревшись внимательнее, вы поймете, что в нашем случае так и есть.

В оболочке метод mapReduce ожидает получить функции map и reduce в качестве аргументов. Но в нашем примере ему передается еще два аргумента. Первый – фильтр запроса, который ограничивает агрегирование заказами, созданными с начала 2010 года. Второй аргумент – имя выходной коллекции.

```
filter = {purchase_date: {$gte: new Date(2010, 0, 1)}}
db.orders.mapReduce(map, reduce, {query: filter, out: 'totals'})
```

Результаты сохраняются в коллекции totals, которую можно опросить так же, как любую другую. В следующем листинге приведень

[10] Поскольку в JavaScript нумерация месяцев начинается с нуля, то номер месяца из меняется в диапазоне от 0 до 11. Чтобы получить осмысленное числовое представление месяца, нужно добавить к его номеру 1. Затем мы добавляем тире – и номер года. Таким образом, ключи выглядят так: 1-2011, 2-2011 и т. д.

результаты запроса к одной из таких коллекций. В поле _id хранится ключ группировки – комбинация года и месяца, а в поле value – вычисленные в ходе редукции итоги.

Листинг 5.3. Запрос коллекции, созданной в результате применения map-reduce

```
> db.totals.find()
{ _id: "1-2011", value: { total: 32002300, items: 59 }}
{ _id: "2-2011", value: { total: 45439500, items: 71 }}
{ _id: "3-2011", value: { total: 54322300, items: 98 }}
{ _id: "4-2011", value: { total: 75534200, items: 115 }}
{ _id: "5-2011", value: { total: 81232100, items: 121 }}
```

Приведенные примеры дают общее представление о практических возможностях агрегирования, имеющихся в MongoDB. В следующих разделах мы рассмотрим ряд технических деталей.

5.4. Агрегирование в деталях

В этом разделе я расскажу о некоторых дополнительных деталях агрегатных функций в MongoDB.

5.4.1. Максимум и минимум

Часто требуется найти минимальное и максимальное значения ключа в коллекции. СУБД на основе SQL предоставляют для этой цели специальные функции min() и max(), но в MongoDB их нет. Вместо этого запрос следует формулировать вручную. Чтобы найти максимум, можно отсортировать коллекцию по убыванию значений ключа и ограничить результирующий набор одним документом. Для нахождения минимума достаточно изменить порядок сортировки на противоположный. Например, чтобы найти отзыв, за который подано наибольшее число голосов, нужно выполнить следующий запрос:

```
db.reviews.find({}).sort({helpful_votes: -1}).limit(1)
```

В поле helpful_votes возвращенного документа будет находиться максимальное значение этого поля. Для нахождения минимума изменяем порядок сортировки:

```
db.reviews.find({}).sort({helpful_votes: 1}).limit(1)
```

Для запуска этого запроса в производственной системе лучше бы построить индекс по полю helpful_votes. А чтобы найти максимальное количество голосов для конкретного продукта, понадобится

составной индекс по полям product_id и helpful_votes. Если вам непонятно, почему, читайте главу 7.

5.4.2. Команда distinct

Команда MongoDB distinct – простейший способ получить список различных значений ключа. Она работает как для одиночных ключей, так и для ключей, значениями которых являются массивы. По умолчанию distinct применяется ко всей коллекции, но ее можно ограничить с помощью селектора запроса.

Команду distinct можно использовать для получения списка уникальных тегов в коллекции товаров:

```
db.products.distinct("tags")
```

Вот так всё просто. Если вы хотите работать с подмножеством коллекции products, то передайте во втором аргументе селектор. В запросе ниже отбираются различные теги для товаров из категории «Садовые инструменты»:

```
db.products.distinct("tags",
      {category_id: ObjectId("6a5b1476238d3b4dd5000048")})
```

Ограничения агрегатных команд. Несмотря на всю свою полезность, команды distinct и group подвержены существенному ограничению: они не могут возвращать результирующий набор объемом более 16 МБ. Это не ограничение именно данных команд, оно действует для всех исходных результирующих наборов. distinct и group реализованы в виде команд, то есть запросов к специальной коллекции $cmd, а, коль скоро это запросы, то к ним применяется указанное ограничение. Если distinct или group не могут справиться с агрегированным набором, то придется прибегнуть к механизму map-reduce, который позволяет сохранять результаты в коллекции, а не возвращать их напрямую.

5.4.3. Команда group

Команда group, как и distinct, реализована в виде команды базы данных и потому подпадает под действие того же ограничения на размер – 16 МБ. Кроме того, чтобы уменьшить потребление памяти, group не обрабатывает более 10 000 уникальных ключей. Если операция агрегирования укладывается в эти пределы, то group – оптимальный выбор, потому что зачастую она работает быстрее, чем map-reduce.

Вы уже видели искусственный пример группировки отзывов по пользователям. Теперь вкратце рассмотрим параметры команды group:

- `key` – документ, описывающий, по каким полям группировать. Например, чтобы группировать по полю `category_id`, нужно в качестве ключа задать `{category_id: true}`. Ключ может быть составным. Так, чтобы сгруппировать множество отзывов по `user_id` и `rating`, нужно задать ключ `{user_id: true, rating: true}`. Параметр `key` обязателен, если только не используется `keyf`.
- `keyf` – JavaScript-функция, которая, будучи применена к документу, генерирует ключ для этого документа. Это полезно, когда ключ группировки вычисляется динамически. Например, если требуется сгруппировать результирующий набор по дню недели, в который создавался документ, но это значение не хранится, то для генерации ключа можно воспользоваться функцией:

```
function(doc) {
     return {day: doc.created_at.getDay();
}
```

  Эта функция генерирует ключи вида `{day: 1}`. Отметим, что параметр `keyf` обязателен, если только не используется `key` для задания стандартного ключа.
- `initial` – документ, используемый в качестве основы для порождения результатов агрегирования. При первом обращении к функции `reduce` этот начальный документ используется в качестве первого значения агрегатора и обычно содержит все агрегируемые ключи. Например, если для каждой группы вычисляется сумма поданных голосов и общее число документов в группе, то начальный документ будет выглядеть так: `{vote_sum: 0.0, doc_count: 0}`. Этот параметр обязателен.
- `reduce` – JavaScript-функция, выполняющая агрегирование. Она принимает два аргумента: текущий документ и документ-агрегатор, в котором хранятся результаты агрегирования. Начальным значением агрегатора будет документ `initial`. Вот пример функции `reduce` для агрегирования голосов и количества документов:

```
function(doc, aggregator) {
     aggregator.doc_count += 1;
     aggregator.vote_sum += doc.vote_count;
}
```

Отметим, что функция reduce не обязана что-то возвращать; она просто должна модифицировать объект-агрегатор. Параметр reduce обязателен.

- cond – селектор запроса, который отбирает подлежащие агрегированию документы. Если вы не хотите агрегировать всю коллекцию, то должны передать селектор. Например, чтобы агрегировать только отзывы, за которые подано более пяти голосов, нужно задать такой селектор запроса: {vote_count: {$gt: 5}}.
- finalize – JavaScript-функция, которая применяется ко всем результирующим документам перед возвратом результирующего набора. Эта функция позволяет выполнить постобработку результатов операции группировки. Типичный пример – усреднение. Можно использовать уже существующие в результате группировки значения для порождения нового содержащего среднее:

```
function(doc) {
     doc.average = doc.vote_count / doc.doc_count;
}
```

Поначалу команда group выглядит сложно. Но, немного попрактиковавшись, вы к ней привыкнете.

5.4.4. Map-reduce

Возможно, вам непонятно, зачем MongoDB поддерживает одновременно group и map-reduce, коль скоро их функциональность очень похожа. На самом деле, сначала никакого другого механизма агрегирования, кроме group, в MongoDB не было, map-reduce появился позже по двум взаимосвязанным причинам. Во-первых, операции в стиле map-reduce стали господствующей тенденцией, поэтому включение в продукт нового способа мышления показалось правильным решением[11]. Во-вторых, этот механизм гораздо более практичен: для обхода больших наборов данных, особенно в конфигурации с несколькими сегментами, необходим распределенный агрегатор. Парадигма map-reduce отлично приспособлена для этого.

[11] Многие разработчики впервые узнали о парадигме map-reduce из знаменитой опубликованной Google статьи о распределенных вычислениях (http://labs.google.com/papers/mapreduce.html). Идеи этой статьи легли в основу Hadoop, каркаса с открытым кодом, предназначенного для распределенной обработки больших наборов данных. После этого идеи map-reduce пошли в массы. Например, в СУБД CouchDB парадигма map-reduce применяется для построения индексов.

У механизма map-reduce много параметров. Ниже они перечислены со всеми по-византийски изощренными деталями.

- map – JavaScript-функция, применяемая к каждому документу. Она должна вызывать метод emit() для порождения агрегируемых ключей и значений. В контексте этой функции this является ссылкой на текущий документ. К примеру, чтобы сгруппировать результаты по идентификатору пользователя и получить в итоге общее число голосов и документов, нужно определить такую функцию распределения:

```
function() {
     emit(this.user_id, {vote_sum: this.vote_count, doc_
count: 1});
}
```

- reduce – JavaScript-функция, которая получает ключ и список значений. Она обязана возвращать значения с такой же структурой, как у значений, находящихся в переданном массиве values. Обычно функция reduce обходит список значений и агрегирует их. Вот как в том же примере, что и выше, редуцировать распределенные этой функции значения:

```
function(key, values) {
     var vote_sum = 0;
     var doc_sum = 0;

     values.forEach(function(value) {
        vote_sum += value.vote_sum;
        doc_sum += value.doc_sum;
     });

     return {vote_sum: vote_sum, doc_sum: doc_sum};
}
```

  Отметим, что значение параметра key часто вообще не используется при агрегировании.

- sort – применяемая к запросу сортировка. Наиболее полезна в сочетании с параметром limit. Это позволяет применить map-reduce к 1000 самых последних (с точки зрения времени создания) документов.
- limit – целое число, задающее максимальное количество отбираемых и сортируемых документов.
- out – определяет, как возвращать выходной набор. Чтобы вернуть его в виде результата самой команды, задайте значение {inline: 1}. Но это будет работать, только если размер результирующего набора не превышает 16 МБ.

Альтернатива – поместить результаты в выходную коллекцию. Для этого значением `out` должна быть строка, задающая имя коллекции.

Одна из проблем с записью в выходную коллекцию заключается в том, что открывается возможность затереть данные сгенерированные при предыдущем прогоне аналогичной команды `map-reduce`. Поэтому существует два варианта: объединить новые результаты со старыми или редуцировать старые данные. В первом случае, когда задан параметр `{merge: "collectionName"}`, новые результаты перезаписывают старые с тем же ключом. Во втором – когда задан параметр `{reduce: "collectionName"}` – существующие значения ключа редуцируются вместе с новыми значениями путем вызова функции `reduce`. Второй вариант особенно полезен для выполнения итеративной последовательности распределения и редукции, когда требуется интегрировать новые данные в уже существующий результат агрегирования. При прогоне задачи `map-reduce` для коллекции можно задать селектор запроса, ограничивающий агрегируемый набор данных.

- `finalize` – JavaScript-функция, применяемая к каждому результирующему документу после завершения фазы редукции
- `scope` – документ, в котором задаются значения, которые должны быть глобально доступны функциям `map`, `reduce` и `finalize`.
- `verbose` – булевское значение; если оно равно `true`, то в документ, возвращаемый командой, включается статистика о времени выполнения задачи `map-reduce`.

К сожалению, подумывая о применении `map-reduce` и `group` в MongoDB, следует иметь в виду одно важное ограничение: быстродействие. Для больших наборов данных эти агрегатные функции часто работают не так быстро, как хотелось бы пользователям. Отнести это почти целиком можно на счет применяемого в MongoDB движка JavaScript. Трудно достичь высокой производительности с однопоточным интерпретирующим (не компилирующим) движком.

Но не надо отчаиваться. Команды `map-reduce` и `group` широко используются и во многих ситуациях достаточны. Но даже если это не так, существует альтернатива и надежда на будущее. Альтернатива состоит в том, чтобы выполнять агрегирование в другом месте. Для очень больших наборов данных обработка с помощью класте

ра Hadoop показала замечательные результаты. А надежда на будущее – это новый набор агрегатных функций, в которых используется компилируемый многопоточный код. Их выпуск планируется спустя некоторое время после выхода версии MongoDB 2.0; с текущим положении дел можно познакомиться на странице https://jira.mongodb.org/browse/SERVER-447.

5.5. Резюме

Запросы и агрегирование – важнейшая часть интерфейса MongoDB. Поэтому познакомившись с материалом этой главы, смело приступайте к тестированию этих механизмов. Если вы не уверены в том, какое сочетание операторов решит задачу, к вашим услугам всегда есть оболочка. Далее мы будем довольно часто пользоваться запросами MongoDB, и следующая глава станет весомым тому подтверждением. В ней мы займемся вопросом об обновлении документов. Поскольку запросы как правило играют ключевую роль при обновлении, то будут даны дополнительные разъяснения, касающиеся языка запросов.

ГЛАВА 6.
Обновление, атомарные операции и удаление

В этой главе:

- Обновление документов.
- Атомарная обработка документов.
- Иерархия категорий и управление запасами.

Под обновлением понимается запись в существующий документ. Для эффективного выполнения этой операции необходимо понимать, какие бывают структуры документов, а также какие выражения MongoDB позволяет использовать в запросах. Пример модели данных интернет-магазина, рассмотренной в двух предыдущих главах, уже дал вам представление о том, как проектируются и опрашиваются схемы. Этими знаниями мы воспользуемся при изучении обновления.

Точнее, мы более подробно поговорим о том, почему при моделировании иерархии категорий мы пошли по пути денормализации и почему такая структура разумна с точки зрения обновления в MongoDB. Мы рассмотрим также управление запасами и попутно решим несколько нетривиальных задач, касающихся параллелизма. Вы узнаете о многих новых операторах обновления, познакомитесь с приемами, в которых используется атомарность операций обновления, и освоите всю мощь команды `findAndModify`. После многочисленных примеров мы перейдем к разделу, посвященному техническим деталям операторов обновления. Я включил также замечания о параллелизме и оптимизации, а в конце главы скажу несколько важных слов об удалении данных в MongoDB.

К концу главы вы будете знать обо всех операциях CRUD в MongoDB и располагать арсеналом средств, необходимых для про

ектирования приложений, в которых в полной мере задействуется интерфейс и модель данных MongoDB.

6.1. Краткий экскурс в обновление документов

Для обновления документа в MongoDB есть два способа. Можно либо заменить документ целиком, либо воспользоваться той или иной комбинацией операторов обновления для модификации конкретных полей документа. Чтобы заложить фундамент для более сложных примеров, я начну эту главу с простой демонстрации обоих способов. А затем расскажу, когда следует предпочесть один способ другому.

Для начала вспомним пример документа с описанием пользователя. В нем хранится фамилия и имя, почтовый адрес и адреса доставки. Без сомнения иногда придется обновлять почтовый адрес, поэтому с него и начнем. Чтобы заменить документ целиком, мы сначала должны запросить его, потом модифицировать на стороне клиента и, наконец, выполнить команду обновления модифицированного документа. Вот как это выглядит на Ruby:

```
user_id = BSON::ObjectId("4c4b1476238d3b4dd5000001")
doc = @users.find_one({:_id => user_id})

doc['email'] = 'mongodb-user@10gen.com'
@users.update({:_id => user_id}, doc, :safe => true)
```

Зная `_id` пользователя, мы запрашиваем документ. Затем локально модифицируем его, в данном случае изменяем атрибут `email`. Потом модифицированный документ передается методу `update`. Последняя строка говорит: «Найди в коллекции `users` документ с данным `_id` и замени его тем, что я передал».

Это модификация путем замены; а теперь рассмотрим модификацию с помощью оператора:

```
@users.update({:_id => user_id},
   {  '$set' => {:email => 'mongodb-user@10gen.com'}},
      :safe => true)
```

В этом примере использован оператор `$set`, один из нескольких специальных операторов обновления, который позволяет изменить почтовый адрес с помощью одного запроса к серверу. В данном случае запрос обновления гораздо конкретнее: найти документ с описанием данного пользователя и записать в поле `email` значение `mongodbuser@10gen.com`.

Ну а как насчет еще одного примера? На этот раз мы добавим новый адрес доставки в список адресов. Вот как это делается путем замены всего документа:

```
doc = @users.find_one({:_id => user_id})

new_address = {
      :name => "work",
      :street => "17 W. 18th St.",
      :city => "New York",
      :state => "NY",
      :zip => 10011
}
doc['shipping_addresses'].append(new_address)
@users.update({:_id => user_id}, doc)
```

А вот – направленное обновление:

```
@users.update({:_id => user_id},
   {  '$push' => {:addresses =>
      {  :name => "work",
         :street => "17 W. 18th St.",
         :city => "New York",
         :state => "NY",
         :zip => 10011
      }
   }
})
```

Как и раньше, при обновлении целиком мы запрашиваем документ у сервера, модифицируем его и отправляем назад. Сама команда обновления ничем не отличается от использованной при обновлении почтового адреса. Напротив, для направленного обновления применяется другой оператор, `$push`, который добавляет новый адрес в существующий массив `addresses`.

Теперь, познакомившись с обоими способами обновления, вы можете поразмыслить о том, когда предпочесть тот или иной метод. Какой кажется вам интуитивно более понятным? Какой, по вашему мнению, эффективнее?

Модификация заменой – более общий подход. Представьте, что приложение выводит HTML-форму для изменения сведений о пользователе. Техника замены всего документа позволяет передать все отправленные данные (после проверки, конечно) MongoDB; код обновления не зависит от того, какие атрибуты пользователя изменены. В частности, если вы собираетесь написать систему объектного отображения для MongoDB, которая должна как-то абстрагировать

операции обновления, то полная замена будет, скорее всего, разумным умолчанием[1].

Однако направленная модификация гораздо эффективнее. Прежде всего, нет нужды в начальном запросе серверу для получения исходного документа. Но не менее важно и то, что документ, описывающий обновление, мал по размеру. Если вы применяете обновление заменой и средний размер документа составляет 100 КБ, то эти 100 КБ придется посылать серверу при каждом обновлении! А теперь сравните с обновлением с помощью операторов `$set` и `$push` – в примерах выше документы, описывающие эти операции, занимают меньше 100 байтов вне зависимости от размера модифицируемого документа. Поэтому применение направленного обновления часто требует меньше времени на сериализацию и передачу данных.

Кроме того, направленные операции позволяют обновлять документы атомарно. Например, если требуется инкрементировать счетчик, то обновление заменой – далеко не идеальный подход; единственный способ произвести изменение атомарно – воспользоваться каким-то вариантом оптимистической блокировки. А в случае направленного обновления можно воспользоваться оператором `$inc` для атомарной модификации счетчика. Это означает, что даже при большом числе одновременно выполняемых обновлений каждый `$inc` применяется изолированно – обновляет данные полностью или не обновляет вовсе[2].

Оптимистическая блокировка. Оптимистической блокировкой или оптимистическим управлением параллелизмом называется техника, позволяющая корректно обновить запись без ее блокирования. Понять, что это такое, проще всего, вспомнив вики-сайты. В каждый момент времени редактировать страницу вики может несколько пользователей. Но вот чего нужно любой ценой избежать, так это ситуации, когда пользователь редактирует и обновляет уже неактуальную версию страницы. Тут-то и приходит на помощь протокол оптимистической блокировки. Когда пользователь пытается сохранить изменения, вместе с данными передается временная метка. Если

[1] Такая стратегия применяется в большинстве систем объектного отображения для MongoDB, и причины понятны. Если пользователю предоставляется возможность моделировать произвольно сложные сущности, то сформировать команду обновления заменой гораздо проще, чем вычислять оптимальную комбинацию специальных операторов обновления.

[2] В документации по MongoDB для обозначения того, что я называю *направленным обновлением*, применяется термин *атомарное обновление*. Эта новая терминология – попытка прояснить употребление слова *атомарный*. На самом деле, все операции обновления выполняются сервером атомарно на уровне документа. Операторы обновления называются атомарными, потому что они позволяют обновить документ, не запрашивая его предварительно

эта метка старше, чем время последнего сохранения страницы, то операция обновления отменяется. Такая стратегия позволяет одновременно редактировать страницу нескольким пользователям, что гораздо лучше альтернативы – требования захватить блокировку перед тем, как приступать к редактированию страницы.

Познакомившись с различными вариантами обновления, вы теперь готовы оценить стратегии, изложенные в следующем разделе Мы вернемся к модели данных интернет-магазина и ответим на некоторые трудные вопросы об управлении этими данными в производственной системе.

6.2. Обновление данных интернет-магазина

Нетрудно привести хрестоматийные примеры обновления того или иного атрибута в документе MongoDB. Но в реальном приложении возникают всякие осложнения, и для обновления атрибута одной строки кода может оказаться недостаточно. В следующих разделах я буду использовать модель данных интернет-магазина, с которой мы работали в двух предыдущих главах, и на ее примере продемонстрирую операции обновления, характерные для реального интернет-магазина. Некоторые из них покажутся вам интуитивно очевидными другие – не очень. Но в целом вы сможете лучше оценить подоплеку схемы, разработанной в главе 4, и понять различные возможности и ограничения языка обновления в MongoDB.

6.2.1. Товары и категории

В этом разделе я приведу два примера направленного обновления Сначала мы рассмотрим вычисление среднего рейтинга товара, а потом решим более сложную задачу – управление иерархией категорий.

Средний рейтинг товара

Для обновления данных о товарах применяются различные стратегии. Если речь идет об интерфейсе администратора для редактирования сведений о товаре, то проще всего было бы извлечь весь документ с описанием товара, внести в него изменения, произведенные пользователем, и заменить документ целиком. В других ситуациях необходимо изменить лишь парочку значений, и тогда имеет прямой

смысл прибегнуть к направленному обновлению. Именно так обстоит дело в случае среднего рейтинга товара. Поскольку пользователям необходимо сортировать товары по среднему рейтингу, то разумно хранить рейтинг в самом документе о товаре и обновлять его при каждом добавлении или удалении отзыва.

Вот один из способов выполнить такое обновление:

```
average = 0.0
count   = 0
total   = 0
cursor = @reviews.find({:product_id => product_id}, :fields => ["rating"])
while cursor.has_next? && review = cursor.next()
   total += review['rating']
   count += 1
end

average = total / count

@products.update({:_id => BSON::ObjectId("4c4b1476238d3b4dd5003981")},
   {'$set' => {:total_reviews => count, :average_review => average}})
```

Здесь мы суммируем поля `rating` по всем отзывам о товаре, а затем вычисляем среднее. Попутно вычисляется общее количество отзывов, что избавляет от необходимости лишний раз обращаться к функции базы данных `count`. Зная общее число отзывов и среднее значение, мы можем выполнить направленное обновление с помощью оператора `$set`.

Читателю, озабоченному производительностью, идея агрегирования всех отзывов о товаре при каждом обновлении может показаться отталкивающей. Предложенный здесь метод при всей своей консервативности в большинстве ситуаций вполне пригоден. Но возможны и другие подходы. Например, можно завести в документе о товаре дополнительное поле, в котором будет храниться текущая сумма и число рейтингов. При вставке нового отзыва мы сначала запрашиваем эти величины, затем вычисляем новое среднее, а затем производим обновление, указывая селектор:

```
{  '$set' => {:average_review => average, :ratings_total => total},
   '$inc' => {:total_reviews => 1}})
```

Но лишь эталонное тестирование системы на представительном наборе данных может показать, оправдан ли такой подход. Так или иначе, этот пример показывает, что в MongoDB часто есть несколько возможных способов решения задачи. Какой из них выбрать, зависит от требований приложения.

Иерархия категорий

Во многих СУБД не существует простого способа представить иерархию категорий. Это относится и к MongoDB, хотя возможности структурирования документа несколько упрощает ситуацию. Документ позволяет оптимизировать чтение, так как в каждой категории может храниться список предков. Единственная сложность – обеспечить актуальность этого списка. Рассмотрим на примере, как это делается.

Нам необходим общий способ обновить список предков любой категории. Вот одно из возможных решений:

```
def generate_ancestors(_id, parent_id)
   ancestor_list = []
   while parent = @categories.find_one(:_id => parent_id) do
      ancestor_list.unshift(parent)
      parent_id = parent['parent_id']
   end

   @categories.update({:_id => _id},
      {"$set" {:ancestors => ancestor_list}})
end
```

Здесь мы поднимаемся по иерархии категорий, последовательно запрашивая атрибут `parent_id` в каждом узле, пока не дойдем до корневого узла (в котором `parent_id` равно `nil`). По пути мы строим список предков в массиве `ancestor_list`. И в самом конце обновляем атрибут `ancestors` с помощью оператора `$set`.

Теперь, располагая базовым строительным блоком, рассмотрим процедуру вставки новой категории. Пусть имеется простая иерархия категорий, показанная на рис. 6.1.

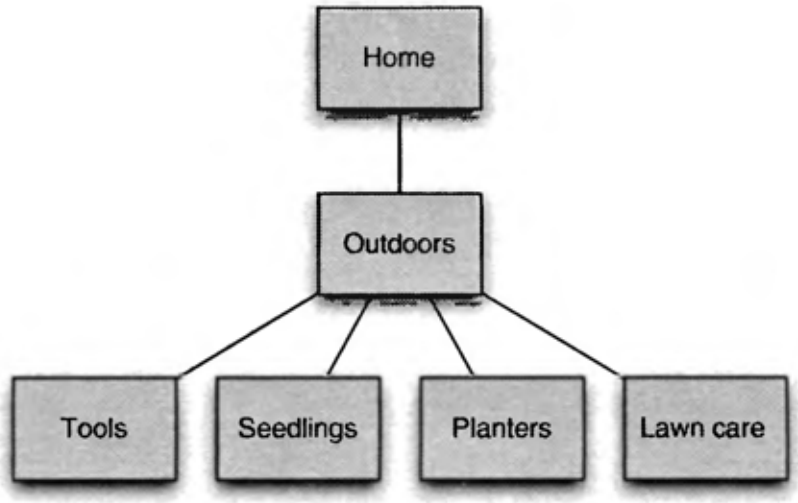


Рис. 6.1. Начальная иерархия категорий

Предположим, что требуется добавить новую категорию Gardening, расположив ее под категорией Home. Мы вставляем новый документ, а затем вызываем написанный ранее метод для порождения списка его предков:

```
category = {
   :parent_id => parent_id,
   :slug => "gardening",
   :name => "Gardening",
   :description => "All gardening implements, tools, seeds, and soil."
}
gardening_id = @categories.insert(category)
generate_ancestors(gardening_id, parent_id)
```

На рис. 6.2 показано обновленное дерево.

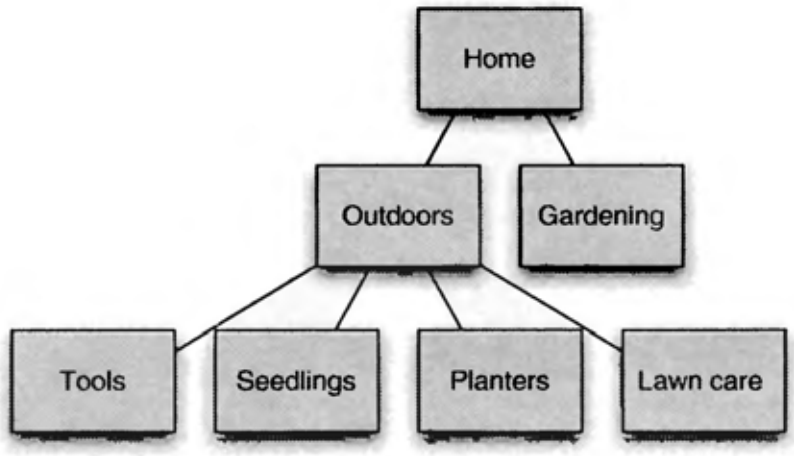


Рис. 6.2. После добавления категории Gardening

Это было несложно. Но что если требуется расположить категорию Outdoors под Gardening? Это труднее, так как необходимо изменить списки предков в нескольких категориях. Начать можно с изменения `parent_id` в категории Outdoors на значение `_id` категории Gardening. Это не очень трудно:

```
@categories.update({:_id => outdoors_id},
   {'$set' => {:parent_id => gardening_id}})
```

Поскольку мы изменили место категории Outdoors в дереве, во всех потомках этой категории списки предков теперь неверны. Это можно исправить, запросив все категории, у которых в списке предков есть Outdoors, и заново сгенерировав для них списки предков. Поскольку MongoDB умеет опрашивать массивы, то эта задача решается тривиально:

```
@categories.find({'ancestors.id' => outdoors_id}).each do |category|
   generate_ancestors(category['_id'], outdoors_id)
end
```

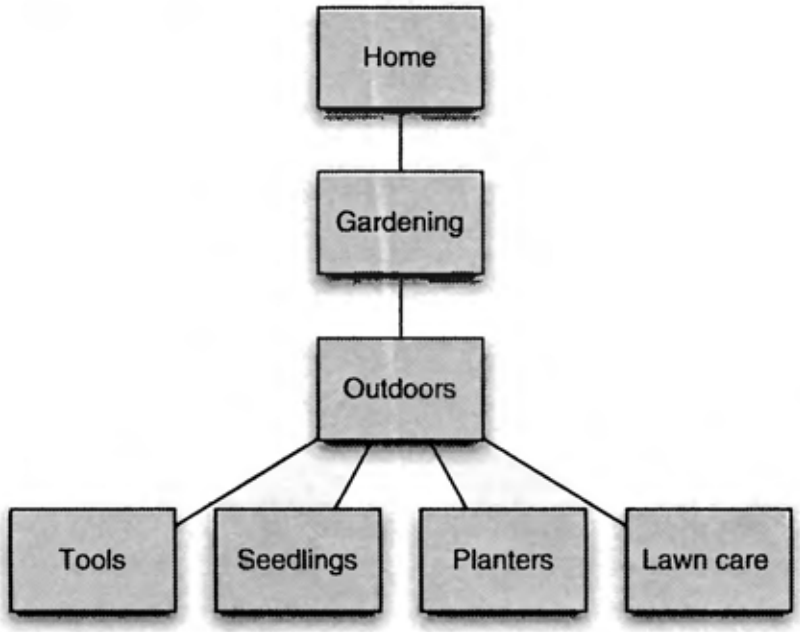


Рис. 6.3. Окончательный вид дерева категорий

Таким образом, мы сумели обновить атрибут категории `parent_id`, новая иерархия категорий показана на рис. 6.3.

Но что если требуется обновить название категории? Если изменить название Outdoors на The Great Outdoors, то придется также заменить строку Outdoors всюду, где она встречается в списках предков других категорий. Впору воскликнуть: «Вот к чему приводит денормализация!». Но, возможно, вас утешит тот факт, что такое обновление можно провести и без повторного вычисления списков предков. Вот как это делается:

```
doc = @categories.find_one({:_id => outdoors_id})
doc['name'] = "The Great Outdoors"
@categories.update({:_id => outdoors_id}, doc)

@categories.update({'ancestors.id' => outdoors_id},
   {'$set' => {'ancestors.$'=> doc}}, :multi => true)
```

Сначала мы находим документ Outdoors, локально изменяем его атрибут `name`, а затем производим обновление заменой. Далее обновленный документ Outdoors используется для замены всех его вхождений в различные списки предков. Для этого применяется позиционный оператор и множественное обновление. Что такое множественное обновление, понять нетрудно; напомним, что если задать параметр `:multi => true`, то обновлены будут все документы, отвечающие селектору. В данном случае, мы хотим обновить все категории, в списке предков которых есть категория Outdoors.

Позиционный оператор – вещь более тонкая. Дело в том, что мы не знаем, в каком месте списка предков находится категория Outdoors. Поэтому оператор обновления должен каким-то образом динамически находить позицию Outdoors в массиве категорий, хранящемся в каждом документе. Тут-то и приходит на помощь позиционный оператор. Этот оператор, обозначенный символом `$` в `ancestors.$`, подставляет индекс элемента массива, в котором хранится документ, отвечающий селектору. Поэтому обновление становится возможным.

Так как необходимость обновлять отдельные поддокументы, хранящиеся в массивах, возникает часто, вы должны иметь позиционный оператор в своем арсенале. Вообще говоря, описанная техника обновления иерархии категорий применима всегда, когда имеется массив поддокументов.

6.2.2. Отзывы

Не все отзывы одинаковы, потому-то приложение и позволяет пользователям голосовать за них. Сама идея элементарна – голосование показывает, насколько отзыв полезен. При моделировании отзыва мы включили в документ общее количество голосов, отданных за его полезность, а также список идентификаторов всех проголосовавших пользователей. Соответствующая часть документа выглядит так:

```
{  helpful_votes: 3,
   voter_ids: [ ObjectId("4c4b1476238d3b4dd5000041"),
   ObjectId("7a4f0376238d3b4dd5000003"),
   ObjectId("92c21476238d3b4dd5000032")
]}
```

Для сохранения информации о голосовании можно применить направленное обновление. Наша стратегия заключается в том, чтобы использовать оператор `$push` для добавления идентификатора проголосовавшего пользователя в список и оператор `$inc` для инкрементирования общего количества голосов – в одной и той же операции:

```
db.reviews.update({_id: ObjectId("4c4b1476238d3b4dd5000041")},
   {  $push: {voter_ids: ObjectId("4c4b1476238d3b4dd5000001")},
      $inc: {helpful_votes: 1}
   })
```

Это почти правильно. Но нужно еще сделать так, чтобы обновление происходило только в случае, когда данный пользователь еще не голосовал за данный отзыв. Поэтому модифицируем селектор, чтобы он отбирал только документы, в которых массив `voter_ids` не содер-

жит идентификатора, который мы собираемся добавить. Добиться этого можно с помощью оператора запроса $ne:

```
query_selector = {_id: ObjectId("4c4b1476238d3b4dd5000041"),
   voter_ids: {$ne: ObjectId("4c4b1476238d3b4dd5000001")}}
db.reviews.update(query_selector,
   {  $push: {voter_ids: ObjectId("4c4b1476238d3b4dd5000001")},
      $inc : {helpful_votes: 1}
   })
```

Это весьма содержательная демонстрация механизма обновления в MongoDB и его применения к документо-ориентированной схеме. В данном случае обновление информации о голосовании производится атомарно и эффективно. Атомарность гарантирует, что даже в среде с высокой степенью параллелизма ни один пользователь не сможет проголосовать более одного раза. А эффективность обеспечивается тем, что проверка на предыдущее голосование и обновление счетчика и списка проголосовавших производятся в одном и том же запросе к серверу.

Если вы решите применить эту технику к записи информации о голосовании, то следите за тем, чтобы любое обновление документа об отзыве также было направленным. Дело в том, что обновление заменой может привести к рассогласованию. Представьте, к примеру, что пользователь обновляет свой отзыв и что такое обновление производится заменой всего документа. Тогда вы сначала запрашиваете документ, который собираетесь обновить. Но между моментом, когда вы запросили документ, и моментом, когда вы его заменили, другой пользователь также мог проголосовать за этот отзыв. Такая последовательность событий изображена на рис. 6.4.

Понятно, что замена документа в момент Т3 затрет обновление, произошедшее в момент Т2. Этого можно избежать путем применения оптимистической блокировки, но лучше в данном случае применить направленное обновление.

6.2.3. Заказы

Обновление заказов может быть произведено с той же атомарностью и эффективностью, что и обновление отзывов. Конкретно я покажу, как реализовать функцию «Добавить в корзину» с помощью направленного обновления. Эта процедура состоит из двух шагов. Сначала конструируется документ с описанием товара, который будет сохранен в массиве позиций заказа. Затем выполняется операция направленного обновления, в которой указывается что это *обновление или*

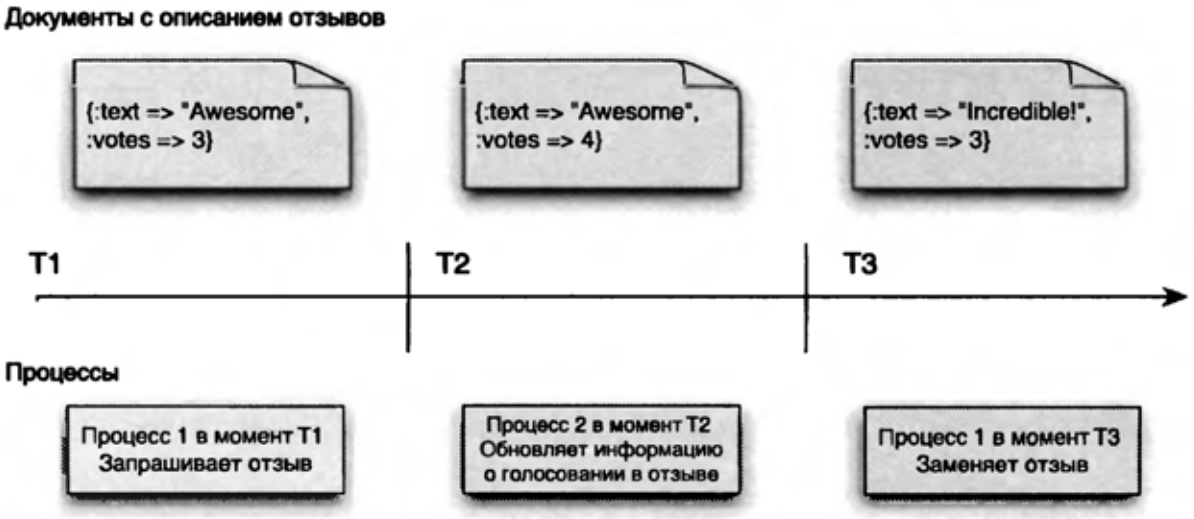


Рис. 6.4. При одновременном использовании направленного обновления и обновления заменой возможна потеря данных

вставка (upsert) – то есть в случае, если обновляемого документа еще не существует, вставляется новый документ (операции обновления или вставки детально описываются в следующем разделе). В результате мы естественно обрабатываем как первоначальное создание корзины, так и последующие добавления в нее[3].

Начнем с конструирования документа, добавляемого в корзину:

```
cart_item = {
   _id: ObjectId("4c4b1476238d3b4dd5003981"),
   slug: "wheel-barrow-9092",
   sku: "9092",
   name: "Extra Large Wheel Barrow",
   pricing: {
      retail: 589700,
      sale: 489700
   }
}
```

Скорее всего, при его построении вы опросите коллекцию `products`, а затем скопируете из полученного документа те поля, которые нужно сохранить в позиции заказа. Полей `_id`, `sku`, `slug`, `name` и `price` вполне достаточно[4]. Имея документ с описанием позиции корзины, мы можем вставить его в коллекцию заказов или обновить, если он уже там есть:

---

[3] Я употребляю термины *корзина* и *заказ* как синонимы, потому что в любом случае имеется в виду один тот же документ. Формально они различаются значением поля `state` (документ, в котором это поле равно `CART`, является корзиной).

[4] В реальном интернет-магазине надо будет проверить, что к моменту оформления заказа цена не изменилась.

```
selector = {user_id: ObjectId("4c4b1476238d3b4dd5000001"),
            state: 'CART',
            'line_items.id':
                  {'$ne': ObjectId("4c4b1476238d3b4dd5003981")}
           }

update = {'$push': {'line_items': cart_item}}

db.orders.update(selector, update, true, false)
```

Чтобы код было проще читать, я конструирую селектор запроса и документ, описывающий обновление, по отдельности. В последнем говорится, что документ с описанием позиций заказа следует поместить в массив позиций `line_items`. Как видно из селектора, обновление не произойдет, если указанной позиции еще нет в массиве. Разумеется, когда пользователь в первый раз выполняет функцию «Добавить в корзину», никакой корзины еще не существует. Именно поэтому мы и указываем операцию обновления или вставки. Таким образом, после первой вставки документ заказа будет выглядеть следующим образом:

```
{
user_id: ObjectId("4c4b1476238d3b4dd5000001"),
state: 'CART',
   line_items: [{
      _id: ObjectId("4c4b1476238d3b4dd5003981"),
      slug: "wheel-barrow-9092",
      sku: "9092",
      name: "Extra Large Wheel Barrow",
      pricing: {
         retail: 589700,
         sale: 489700
      }
   }]
}
```

Затем выполняется еще одно направленное обновление, чтобы привести в соответствие количество позиций и подитог заказа:

```
selector = {   user_id: ObjectId("4c4b1476238d3b4dd5000001"),
                        state: "CART",
                        'line_items.id': ObjectId("4c4b1476238
                        d3b4dd5003981")}

update = {  $inc:
                      {  'line_items.$.qty': 1,
                         sub_total: cart_item['pricing']['sale']
                      }
                }

db.orders.update(selector, update)
```

Обратите внимание, что для обновления подитога заказа и заказанного количества в одной позиции используется один и тот же оператор `$inc`. Последнее обновление стало возможно благодаря позиционному оператору (`$`), с которым мы познакомились в предыдущем подразделе. Основная причина, по которой второе обновление необходимо, заключается в том, чтобы обработать случай, когда пользователь нажимает кнопку «Добавить в корзину» для товара, который уже есть в корзине. В этом случае первое обновление не пройдет, но изменить количество и подитог все равно надо. Таким образом, если два раза нажать кнопку «Добавить в корзину» для тачки, то корзина будет выглядеть следующим образом:

```
{
'user_id': ObjectId("4c4b1476238d3b4dd5000001"),
'state' : 'CART',
'line_items': [{
      _id: ObjectId("4c4b1476238d3b4dd5003981"),
      qty: 2,
      slug: "wheel-barrow-9092",
      sku: "9092",

      name: "Extra Large Wheel Barrow",
      pricing: {
         retail: 589700,
         sale: 489700
      }
}],

subtotal: 979400
}
```

Теперь в корзине две тачки, и это отражено также в подитоге.

Есть еще несколько операций, которые нужно реализовать, чтобы функциональность корзины можно было считать законченной. Большую их часть, например удаление товара из корзины или полную очистку корзины, можно выполнить с помощью одного или нескольких направленных обновлений. Если пока это не очевидно, то следующий раздел, в котором описаны все операторы запроса, внесет ясность. Что касается собственно обработки заказа, то ее можно реализовать с помощью последовательности состояний документа, в каждом из которых применяются некоторые логические операции. Мы продемонстрируем это в следующем разделе, где я расскажу об атомарной обработке документа и команде `findAndModify`.

6.3. Атомарная обработка документа

Есть одна вещь, без которой вам никак не обойтись, – команда MongoDB `findAndModify`[5]. Она позволяет атомарно обновить документ и вернуть его в ходе одного цикла обмена данными с сервером. И это очень важно, принимая во внимание, что эта команда позволяет делать. Например, ее можно использовать для создания очередей задач и конечных автоматов. А на основе этих примитивов можно реализовать простую транзакционную семантику, что резко расширяет спектр приложений, которые можно разрабатывать на базе MongoDB. Располагая транзакционными возможностями, вы можете создать в MongoDB сайт интернет-магазина целиком – не только то, что связано с товарным наполнением, но также механизм оформления заказа и управления запасами.

Чтобы убедить вас в этом, я рассмотрю два примера команды `findAndModify` в действии. Сначала я покажу, как реализовать простые переходы состояний для корзины. А затем перейду к несколько более сложной задаче управления ограниченными запасами.

6.3.1. Переходы состояний заказа

У любого перехода состояний есть две части: запрос, проверяющий корректность начального состояния, и обновление, приводящее к изменению состояния. Мы пропустим несколько шагов оформления заказа и предположим, что пользователь готов нажать кнопку «Оплатить» и тем самым подтвердить покупку. Если вы собираетесь синхронно авторизовать кредитную карту пользователя на стороне приложения, то должны обеспечить следующее:

1. Авторизуется именно та сумма, которую пользователь видит на экране.
2. Во время авторизации содержимое корзины не изменяется.
3. В случае ошибки в ходе авторизации корзина возвращается в предыдущее состояние.

---

[5] Название этой команды зависит от окружения. В оболочке применяется ВерблюжьяНотация – `db.orders.findAndModify`, а в Ruby используются подчерки – `find_and_modify`. Чтобы запутать ситуацию еще сильнее, сам сервер распознает эту команду по имени `findandmodify`. Если вам когда-нибудь придется посылать ее серверу напрямую, то пользуйтесь именно последней формой.

4. Если кредитная карта успешно авторизована, то информация об оплате заносится в заказ, и заказ переходит в состояние SHIPMENT PENDING (ожидает доставки).

На первом шаге мы переводим заказ в состояние PRE-AUTHORIZE (перед авторизацией). С помощью команды findAndModify мы находим текущий объект заказа и убеждаемся, что он находится в состоянии CART:

```
db.orders.findAndModify({
   query: {user_id: ObjectId("4c4b1476238d3b4dd5000001"),
           state: "CART" },

   update: {"$set": {"state": "PRE-AUTHORIZE"},
            new: true}
})
```

Если всё хорошо, то findAndModify вернет объект заказа с измененным состоянием[6]. Раз заказ находится в состоянии PRE-AUTHORIZE, пользователь не сможет изменять содержимое корзины, так как в любой операции обновления проверяется, что состояние заказа равно CART. Переведя заказ в состояние «перед авторизацией», мы можем пересчитать различные итоги для возвращенного объекта. Зная итоги, выполним еще раз команду findAndModify, которая переведет документ в состояние AUTHORIZING (производится авторизация) только в том случае, когда новые итоги совпадают с ранее вычисленными. Вот как выглядит эта команда:

```
db.orders.findAndModify({
   query: {user_id: ObjectId("4c4b1476238d3b4dd5000001"),
           total: 99000,
           state: "PRE-AUTHORIZE" },

   update: {"$set": {"state": "AUTHORIZING"}}
})
```

Если вторая команда findAndModify завершается с ошибкой, то следует вернуть заказ в состояние CART и уведомить пользователя о новых итогах. Если же всё хорошо, то мы знаем, что будем авторизовывать именно ту сумму, которую видит пользователь. Теперь можно обращаться к API авторизации, то есть приложение отправляет запрос на авторизацию кредитной карты данного пользователя. Если авторизация не пройдет, то мы зафиксируем ошибку и, как и раньше, вернем заказ в состояние CART.

[6] По умолчанию команда findAndModify возвращает документ в том виде, в котором он был до обновления. Чтобы вернуть модифицированный документ, следует задать параметр {new: true}, как в этом примере.

Если же авторизация завершится успешно, то мы запишем сведения о ней в заказ и переведем его в следующее состояние. Ниже показано, как реализовать это в одной команде findAndModify. Здесь мы используем документ, представляющий подтверждение авторизации, который присоединяется к заказу:

```
auth_doc = {   ts: new Date(),
                          cc: 3432003948293040,
                          id: 29238382910293844839493 48,
                          gateway: "Authorize.net"}

db.orders.findAndModify({
   query: {user_id: ObjectId("4c4b1476238d3b4dd5000001"),
           state: "AUTHORIZING" },

update: {"$set":
           {"state": "PRE-SHIPPING"},
            "authorization": auth}
})
```

Важно знать о тех средствах MongoDB, которые делают возможной такую транзакционную процедуру. Это способность атомарно модифицировать любой документ. Это гарантия согласованного чтения на одном соединении. И, наконец, это сама структура документа, которая позволяет выполнять операции в контексте одного документа, когда MongoDB может гарантировать атомарность. В нашем случае поддержка сложной структуры позволяет хранить в одном документе позиции заказа, товары, цены и информацию о пользователе. Следовательно, для оформления покупки мы можем работать только с одним документом.

По идее, это должно вас впечатлить. Но, быть может, вы, как и я, задаетесь вопросом, можно ли в MongoDB реализовать транзакционное поведение при работе с *несколькими объектами*. Ответ – да, но с осторожностью. Как это делается, я продемонстрирую еще на одном важном элементе интернет-магазина – управлении запасами.

6.3.2. Управление запасами

Не всякому сайту интернет-магазина необходимо строгое управление запасами. Как правило, запасы стандартных товаров можно пополнить достаточно быстро и не задерживать оформление заказа даже в том случае, когда в данный момент нужного количества нет на складе. В таких случаях управление запасами сводится просто к управлению ожиданиями – если на складе не хватает товара, подправить оценку времени доставки.

Совсем другое дело – продажа товаров, единственных в своем роде. Предположим, вы торгуете билетами на концерт с фиксированными местами или предметами рукоделия. Тут никакой замены быть не может; пользователю нужна гарантия, что он получит именно тот товар, который выбрал. Ниже я опишу одно из возможных решений этой задачи в MongoDB. А заодно продемонстрирую дополнительные возможности команды `findAndModify` и продуманное использование модели документа. Кроме того, я покажу, как реализовать транзакционную семантику при работе с несколькими документами.

Подход к моделированию запасов проще всего понять, представив себе реальный магазин. Находясь в магазине для садоводов, вы видите и можете пощупать физический инвентарь: десятки лопат, грабель, секаторов выложены на витринах. Если вы положите лопату себе в корзину, то для других покупателей станет одной лопатой меньше. Следовательно, никакие два покупателя не могут одновременно положить в корзину одну и ту же лопату. Тот же принцип можно применить и к моделированию запасов. Для каждого физического товара на складе заводится документ в коллекции, описывающей запасы. Если на складе 10 лопат, то в базе данных будет 10 документов о лопатах. Каждый элемент на складе связывается с товаром по полю `sku` (артикул) и может находиться в одном из четырех состояний: `AVAILABLE` (0), `IN_CART` (1), `PRE_ORDER` (2), `PURCHASED` (3).

Следующий метод добавляет три лопаты, трое грабель и три набора секаторов на склад:

```
3.times do
   @inventory.insert({:sku => 'shovel', :state => AVAILABLE})
   @inventory.insert({:sku => 'rake', :state => AVAILABLE})
   @inventory.insert({:sku => 'clippers', :state => AVAILABLE})
end
```

Для управления запасами мы заведем специальный класс-кладовщик. Сначала посмотрим, как он используется, а потом сорвем покровы и обнажим реализацию.

Кладовщик может добавлять в корзину произвольные наборы товаров. В коде ниже мы создаем новый объект заказа и новый кладовщик. Потом мы просим кладовщика добавить в заказ три лопаты и один набор секаторов, передавая методу `add_to_cart` идентификатор заказа и два документа, описывающих товары (вместе с количеством):

```
@order_id = @orders.insert({:username => 'kbanker', :item_ids => []})
@fetcher = InventoryFetcher.new(:orders => @orders,
```

```
                                    :inventory => @inventory)

@fetcher.add_to_cart(@order_id,
                {:sku => "shovel", :qty => 3},
                {:sku => "clippers", :qty => 1})

order = @orders.find_one({"_id" => @order_id})
puts "\nHere's the order:"
p order
```

Метод `add_to_cart` возбудит исключение, если не сможет добавить в корзину все запрошенные товары. Если же всё пройдет без ошибок, то заказ будет выглядеть так:

```
{  "_id" => BSON::ObjectId('4cdf3668238d3b6e3200000a'),
   "username"=>"kbanker",
   "item_ids" => [BSON::ObjectId('4cdf3668238d3b6e32000001'),
                  BSON::ObjectId('4cdf3668238d3b6e32000004'),
                  BSON::ObjectId('4cdf3668238d3b6e32000007'),
                  BSON::ObjectId('4cdf3668238d3b6e32000009')],
}
```

Идентификатор `_id` физического элемента складского хранения сохраняется в документе о заказе. Опросить эти элементы можно следующим образом:

```
puts "\nHere's each item:"
order['item_ids'].each do |item_id|
   item = @inventory.find_one({"_id" => item_id})
   p item
end
```

Нетрудно заметить, что каждый элемент находится в состоянии 1, то есть `IN_CART`. Отметим также, что вместе с каждым элементом хранится время последнего изменения состояния. Впоследствии эту временную метку можно будет использовать, чтобы вернуть на склад элементы, находящиеся в корзине слишком долго. Например, можно дать пользователю 15 минут на оформление заказа с момента добавления товара в корзину:

```
{  "_id" => BSON::ObjectId('4cdf3668238d3b6e32000001'),
   "sku"=>"shovel", "state"=>1, "ts"=>"Sun Nov 14 01:07:52 UTC 2010"}

{  "_id"=>BSON::ObjectId('4cdf3668238d3b6e32000004'),
   "sku"=>"shovel", "state"=>1, "ts"=>"Sun Nov 14 01:07:52 UTC 2010"}

{  "_id"=>BSON::ObjectId('4cdf3668238d3b6e32000007'),
   "sku"=>"shovel", "state"=>1, "ts"=>"Sun Nov 14 01:07:52 UTC 2010"}

{  "_id"=>BSON::ObjectId('4cdf3668238d3b6e32000009'),
   "sku"=>"clippers", "state"=>1, "ts"=>"Sun Nov 14 01:07:52 UTC 2010"
```

Если такой API класса `InventoryFetcher` кажется вам осмысленным, то у вас должны появиться догадки о том, как будет реализовано управление запасами. Неудивительно, что в основе реализации лежит команда `findAndModify`. Полный код класса `InventoryFetcher` вместе с тестами включен в состав исходного кода, прилагаемого к этой книге. Мы не будем изучать его построчно, а сосредоточимся лишь на трех основных методах.

Прежде всего, получив список элементов, которые нужно добавить в корзину, кладовщик пытается перевести каждый из состояния `AVAILABLE` в состояния `IN_CART`. Если в какой-то точке возникнет ошибка (элемент не удалось добавить в корзину), то вся операция откатывается. Взгляните на код метода `add_to_cart`, который мы вызывали:

```
def add_to_cart(order_id, *items)
   item_selectors = []
   items.each do |item|
      item[:qty].times do
         item_selectors << {:sku => item[:sku]}
      end
   end

   transition_state(order_id, item_selectors, :from => AVAILABLE,
         :to => IN_CART)
end
```

Этот метод делает не так уж много. Он получает описания всех добавляемых в корзину элементов и разворачивает количества, так чтобы для каждого физического элемента существовало по одному селектору. Например, документ, требующий добавить в корзину две лопаты:

```
{:sku => "shovel", :qty => 2}
```

преобразуется в такой:

```
[{:sku => "shovel"}, {:sku => "shovel"}]
```

Для каждого добавляемого в корзину элемента необходим отдельный селектор запроса. Далее метод передает массив селекторов другому методу, `transition_state`. В приведенном выше коде говорится, что нужно перейти из состояния `AVAILABLE` в состояние `IN_CART`:

```
def transition_state(order_id, selectors, opts={})
   items_transitioned = []

   begin
      for selector in selectors do
```

```
            query = selector.merge(:state => opts[:from])

            physical_item = @inventory.find_and_modify(:query => query,
               :update => {'$set' => {:state => opts[:to], :ts =>
                                      Time.now.utc}})

            if physical_item.nil?
               raise InventoryFetchFailure
            end

            items_transitioned << physical_item['_id']

            @orders.update({:_id => order_id},
                           {"$push" => {:item_ids => physical_item['_id']}})
         end

      rescue Mongo::OperationFailure, InventoryFetchFailure
         rollback(order_id, items_transitioned, opts[:from], opts[:to])
         raise InventoryFetchFailure, "Failed to add #{selector[:sku]}"
      end

      items_transitioned.size
   end
```

Чтобы можно было перейти в другое состояние, в каждый селек тор записывается дополнительное условие `{:state => AVAILABLE}` после чего селектор передается функции `findAndModify`, которая если находит соответствие, устанавливает временную метку и изме няет состояние элемента. Затем метод сохраняет список элементов переведенных в новое состояние, и обновляет заказ, записывая в нег идентификатор только что добавленного элемента.

Если команда `findAndModify` завершается с ошибкой и возвраща ет `nil`, то мы возбуждаем исключение `InventoryFetchFailure`. Если же причиной ошибки являются сбои в сети, то мы перехватываем ис ключение `Mongo::OperationFailure`. В любом случае обработчи исключения откатывает все уже переведенные в новое состояние эле менты, после чего возбуждает исключение `InventoryFetchFailure` включая в него артикул элемента, который не удалось добавить. За тем, на уровне приложения, это исключение можно перехватить и со общить об ошибке пользователю.

Осталось рассмотреть только код отката:

```
def rollback(order_id, item_ids, old_state, new_state)
   @orders.update({"_id" => order_id},
                  {"$pullAll" => {:item_ids => item_ids}})
   item_ids.each do |id|
      @inventory.find_and_modify(
         :query => {"_id" => id, :state => new_state},
```

```
            :update => {"$set" => {:state => old_state, :ts => Time.now.utc}}
        )
    end
end
```

Мы используем оператор `$pullAll`, чтобы удалить все идентификаторы, добавленные в массив `item_ids`, хранящийся в заказе. Затем обходим список идентификаторов элементов и переводим каждый в прежнее состояние.

Метод `transition_state` можно использовать как основу для других методов, изменяющих состояния элементов. Нетрудно будет включить его в систему изменения состояний заказа, разработанную в предыдущем подразделе. Но это я оставляю в качестве упражнения для читателя.

Можно задать законный вопрос: «Достаточно ли надежна эта система для промышленной эксплуатации?» Ответить на него, не зная конкретных деталей, нелегко, но можно с уверенностью утверждать, что MongoDB располагает достаточными средствами для реализации решения, нуждающегося в транзакционном поведении. Согласен, никто не будет строить банковскую систему на основе MongoDB. Но если требуется некое подобие транзакционной семантики, то имеет смысл рассмотреть MongoDB как потенциального кандидата, особенно если вы хотите реализовать приложение целиком на базе одной СУБД.

6.4. Технические детали: обновление и удаление в MongoDB

Чтобы по-настоящему разобраться в операциях обновления в MongoDB, необходимо целостное представление о документной модели и языке запросов MongoDB. Примеры из предыдущих разделов окажут в этом неоценимую помощь. Но сейчас, как и во всех разделах «Технические детали» в этой книге, мы попробуем докопаться до сути дела. Для этого я вкратце перечислю все элементы интерфейса обновления в MongoDB, а также сделаю несколько замечаний о производительности. Краткости ради, примеры в последующих подразделах написаны на JavaScript.

6.4.1. Типы и параметры операций обновления

MongoDB поддерживает два типы обновлений: направленное и заменой. В первом случае используются операторы обновления, один или

несколько; во втором – указывается новый документ, который полностью заменяет документ, определяемый селектором запроса.

Замечание о синтаксисе: обновление и запрос

Начинающие пользователи MongoDB иногда плохо понимают разницу между синтаксисом обновления и запроса. Операция направленного обновления всегда начинается оператором обновления, и этот оператор почти всегда обозначается глаголом. Возьмем, к примеру, оператор `$addToSet`:

```
db.products.update({}, {$addToSet: {tags: 'green'}})
```

Если же добавить в эту команду `update` селектор запроса, то семантически он будет играть роль прилагательного и располагаться после имени поля, по которому производится запрос:

```
db.products.update({'price' => {$lte => 10}},
   {$addToSet: {tags: 'cheap'}})
```

По сути дела, операторы обновления префиксные, тогда как операторы запроса – обычно инфиксные.

Отметим, что обновление завершится ошибкой, если описывающий его документ неоднозначен. Ниже мы употребили оператор обновления `$addToSet` в сочетании с семантикой замены, `{name: "Pitchfork"}`:

```
db.products.update({}, {name: "Pitchfork", $addToSet: {tags: 'cheap'}})
```

Если вы намеревались изменить поле `name` в документе, то надо было использовать оператор `$set`:

```
db.products.update({},
   {$set: {name: "Pitchfork"}, $addToSet: {tags: 'cheap'}})
```

Множественное обновление

По умолчанию операция обновления изменяет только первый документ, сопоставившийся с селектором запроса. Чтобы обновить все сопоставившиеся документы, необходимо явно указать режим множественного обновления. В оболочке для этого нужно передать `true` в качестве четвертого аргумента метода `update`. Ниже показано, как добавить тег `cheap` во все документы в коллекции `products`:

```
db.products.update({}, {$addToSet: {tags: 'cheap'}}, false, true)
```

В драйвере для Ruby (и в большинстве других драйверов) множественное обновление можно выразить более наглядно:

```
@products.update({}, {'$addToSet' => {'tags' => 'cheap'}}, :multi => true)
```

Обновление или вставка

Часто бывает необходимо вставить элемент, если его еще нет, и обновить, если он уже существует. Этот фокус, обычно непростой

можно проделать с помощью имеющейся в MongoDB операции обновления или вставки (upsert). Если селектор запроса находит документ, то производится его обновление. Если же такого документа нет, то вставляется новый. Атрибуты нового документа получаются логическим объединением селектора запроса и документа, описывающего направленное обновление[7]. Вот простой пример обновления или вставки из оболочки:

```
db.products.update({slug: 'hammer'}, {$addToSet: {tags: 'cheap'}}, true)
```

А это его эквивалент на Ruby:

```
@products.update({'slug' => 'hammer'},
   {'$addToSet' => {'tags' => 'cheap'}}, :upsert => true)
```

Как и следовало ожидать, операция обновления или вставки способна обновить (или вставить) только один документ. Вы убедитесь в ее чрезвычайной полезности, когда возникнет необходимость в атомарном обновлении или когда вы заранее не знаете, существует ли подлежащий обновлению документ. Практический пример приведен в разделе 6.2.3, где описывается добавление товаров в корзину.

6.4.2. Операторы обновления

MongoDB поддерживает разнообразные операторы обновления. Ниже приведены краткие примеры их использования.

Стандартные операторы обновления

Это операторы общего характера, все они работают почти с любыми типами данных.

$inc

Оператор `$inc` служит прежде всего для инкремента или декремента числового значения:

```
db.products.update({slug: "shovel"}, {$inc: {review_count: 1}})
db.users.update({username: "moe"}, {$inc: {password_retires: -1}})
```

Но он позволяет также прибавить или вычесть произвольное число:

```
db.readings.update({_id: 324}, {$inc: {temp: 2.7435}})
```

Оператор `$inc` столь же эффективен, сколь и удобен. Поскольку при его выполнении размер документа изменяется редко, то обычно

[7] Отметим, что операция обновления или вставки не работает в режиме обновления заменой.

обновление производится по месту и, стало быть, затрагивает только указанное значение[8].

Как продемонстрировано в коде добавления товаров в корзину оператор `$inc` работает и с операцией вставки или обновления. В частности, предыдущий пример можно переписать так:

```
db.readings.update({_id: 324}, {$inc: {temp: 2.7435}}, true)
```

Если не существует отсчета с `_id`, равным 324, то будет создан новый документ с таким `_id`, в котором полю `temp` будет присвоено значение, равное 2.7435.

$set и $unset

Если требуется задать значение некоторого ключа в документе то следует использовать оператор `$set`. Значение может иметь любой допустимый тип BSON. Это означает, что все приведенные ниже команды правильны:

```
db.readings.update({_id: 324}, {$set: {temp: 97.6}})
db.readings.update({_id: 325}, {$set: {temp: {f: 212, c: 100} })
db.readings.update({_id: 326}, {$set: {temps: [97.6, 98.4, 99.1]}})
```

Если ключ, которому присваивается значение, уже существует, то его значение будет перезаписано, в противном случае будет создан новый ключ.

Оператор `$unset` удаляет указанный ключ из документа. Ниже показано, как удалить ключ `temp`:

```
db.readings.update({_id: 324}, {$unset: {temp: 1})
```

Оператор `$unset` применим также к вложенным документам и к массивам. В обоих случаях внутренний объект адресуется с помощью точечной нотации. Если в коллекции есть такие два документа:

```
{_id: 325, 'temp': {f: 212, c: 100}}
{_id: 326, temps: [97.6, 98.4, 99.1]}
```

то для удаления отсчетов по шкале Фаренгейта из первого документа и обнуления нулевого элемента во втором документе нужно поступить следующим образом:

```
db.readings.update({_id: 325},
   {$unset: {'temp.f': 1}})

db.readings.update({_id: 236},
   {$pop: {temps: -1})
```

---

[8] Исключением из этого правила является ситуация, когда изменяется числовой тип. Если в результате выполнения `$inc` 32-разрядное число преобразуется в 64-разрядное, то перезаписывается весь BSON-документ.

Точечную нотацию для доступа к поддокументам и элементам массива можно использовать и в операторе `$set`.

$rename

Для изменения имени ключа служит оператор `$rename`:

```
db.readings.update({_id: 324}, {$rename: {'temp': 'temperature'}})
```

Переименовывать можно также целый поддокумент:

```
db.readings.update({_id: 325}, {$rename: {'temp.f': 'temp.farenheit'}})
```

Использование $unset с массивами

Отметим, что применение оператора `$unset` к отдельным элементам массива может работать не так, как вы ожидаете. Оператор не удаляет элемент, а просто записывает в соответствующую позицию массива значение `null`. Чтобы удалить элемент из массива, пользуйтесь операторами `$pull` и `$pop`.

```
db.readings.update({_id: 325}, {$unset: {'temp.f': 1}})
db.readings.update({_id: 326}, {$unset: {'temp.0': 1}})
```

Операторы обновления массива

Сейчас вам уже, вероятно, очевидно, что массивы играют в документной модели MongoDB важнейшую роль. Естественно, имеется ряд операторов обновления, применяемых исключительно к массивам.

$push и $pushAll

Для добавления значений в конец массива пользуйтесь операторами `$push` и `$pushAll`. Оператор `$push` добавляет единственное значение, а `$pushAll` – список значений. Например, совсем нетрудно добавить новый тег в описание товара «лопата» (shovel):

```
db.products.update({slug: 'shovel'}, {$push: {'tags': 'tools'}})
```

Но и добавить за одну операцию несколько тегов ничуть не сложнее:

```
db.products.update({slug: 'shovel'},
   {$pushAll: {'tags': ['tools', 'dirt', 'garden']}})
```

Отметим, что в массив можно добавлять любые значения, а не только скаляры. Пример был приведен в предыдущем разделе, где мы добавляли товар в массив позиций, помещенных в корзину.

$addToSet и $each

Оператор `$addToSet` также добавляет значение в конец массива, но лишь в том случае, если такого значения еще нет. Таким образом,

если у лопаты уже имеется тег tool (инструмент), то следующая команда обновления не изменит документ:

```
db.products.update({slug: 'shovel'}, {$addToSet: {'tags': 'tools'}})
```

Если требуется в одной операции добавить несколько уникальных значений, то следует использовать $addToSet в сочетании с оператором $each:

```
db.products.update({slug: 'shovel'},
   {$addToSet: {'tags': {$each: ['tools', 'dirt', 'steel']}}})
```

Добавляются только те из перечисленных в $each значений, которые отсутствуют в массиве tags.

$pop

Самый простой способ удалить элемент из массива – воспользоваться оператором $pop. Если добавить элемент в конец массива оператором $push, то следующий за ним $pop удалит только что добавленный элемент. Хотя оператор $pop часто употребляется вместе с $push, его можно использовать и самостоятельно. Если массив tags содержит значения ['tools', 'dirt', 'garden', 'steel'], то показанный ниже оператор $pop удалит тег steel:

```
db.products.update({slug: 'shovel'}, {$pop: {'tags': 1}})
```

Синтаксически оператор $pop похож на $unset: {$pop: {'elementToRemove': 1}}. Но в отличие от $unset, значением второго параметра может быть -1 – это означает, что нужно удалить первый элемент массива. Вот как можно удалить из массива тег tools:

```
db.products.update({slug: 'shovel'}, {$pop: {'tags': -1}})
```

Есть одна неприятность: невозможность вернуть значение удаленное оператором $pop. Таким образом, вопреки своему имени $pop работает не совсем так, как одноименная операция со стеком Помните об этом.

$pull и $pullAll

Оператор $pull похож на $pop, но несколько сложнее. Он позволяет точно указать удаляемый элемент – по значению, а не по номеру позиции. Всё в том же примере для удаления тега dirt необязательно знать, в какой позиции массива он находится; достаточно передать сам тег оператору $pull:

```
db.products.update({slug: 'shovel'}, {$pull {'tags': 'dirt'}})
```

Оператор $pullAll работает аналогично $pushAll, то есть позволяет передать список удаляемых значений. Чтобы удалить теги

dirt и garden, можно использовать $pullAll следующим образом:

```
db.products.update({slug: 'shovel'}, {$pullAll {'tags': ['dirt', 'garden']}})
```

Позиционное обновление

Часто данные в MongoDB моделируются в виде массива поддокументов, однако до появления позиционного оператора манипулировать такими поддокументами было нелегко. Позиционный оператор позволяет обновлять поддокумент в массиве. Для определения нужного поддокумента в селекторе запроса применяется точечная нотация. Без примера разобраться в этом трудно, поэтому предположим, что имеется документ с описанием заказа, в котором есть такая часть:

```
{  _id: new ObjectId("6a5b1476238d3b4dd5000048"),
   line_items: [
      {  _id: ObjectId("4c4b1476238d3b4dd5003981"),
         sku: "9092",
         name: "Extra Large Wheel Barrow",
         quantity: 1,
         pricing: {
            retail: 5897,
            sale: 4897,
         }
      },
      {  _id: ObjectId("4c4b1476238d3b4dd5003981"),
         sku: "10027",
         name: "Rubberized Work Glove, Black",
         quantity: 2,
         pricing: {
            retail: 1499,
            sale: 1299,
         }
      }
   ]
}
```

Мы хотим изменить количество во втором документе – том, для которого SKU равно 10027, – задав новое значение 5. Проблема в том, что мы не знаем, в какой именно позиции массива line_items этот документ находится. Мы даже не знаем, существует ли он вообще. Однако простой селектор и документ, описывающий обновление с использованием позиционного оператора, решают обе проблемы:

```
query = {_id: ObjectId("4c4b1476238d3b4dd5003981"),
         'line_items.sku': "10027"}
update = {$set: {'line_items.$.quantity': 5}}

db.orders.update(query, update)
```

Позиционный оператор – это символ $ в строке 'line_items.$.quantity'. Если найдется документ, отвечающий селектору запроса, то сервер подставит вместо позиционного оператора индекс поддокумента, в котором SKU равно 10027, поэтому обновление будет применено к корректному поддокументу. Таким образом, в моделях, содержащих поддокументы, позиционный оператор весьма полезен для выполнения нетривиальных обновлений.

6.4.3. Команда findAndModify

Выше мы уже привели немало содержательных примеров использования команды findAndModify, так что осталось лишь формально перечислить ее параметры. Обязательными являются только параметры query и update либо remove.

- query – селектор запроса. По умолчанию {}.
- update – документ, описывающий обновление. По умолчанию {}.
- remove – булевская величина; если равна true, то команда удаляет объект и возвращает его в качестве своего значения. По умолчанию false.
- new – булевская величина; если равна true, то команда возвращает модифицированный документ в том виде, в котором он окажется после обновления. По умолчанию false.
- sort – документ, определяющий направление сортировки. Поскольку команда findAndModify изменяет только один документ за раз, то этот параметр может быть полезен для уточнения того, какой из отобранных документов обрабатывать. Например, можно задать критерий сортировки {created_at: -1}, чтобы обрабатывать документ, созданный последним.
- fields – этот параметр определяет, какие поля возвращать, что особенно полезно в случае больших документов. Поля задаются так же, как в любом запросе. Примеры см. в разделе о полях в главе 5.
- upsert – булевская величина; если равна true, то команда findAndModify трактуется как вставка или обновление. Если искомого документа не существует, он создается. Если вы хотите еще и вернуть вновь созданный документ, то необходимо задать параметр {new: true}.

6.4.4. Операции удаления

Вам будет приятно узнать, что с удалением документов никаких сложностей не возникает. Можно удалить коллекцию целиком или передать методу `remove` селектор, чтобы удалить только некоторые документы. Следующая команда удалит все отзывы:

```
db.reviews.remove({})
```

Но гораздо чаще приходится удалять отзывы одного конкретного пользователя:

```
db.reviews.remove({user_id: ObjectId('4c4b1476238d3b4dd5000001')})
```

Поэтому команда `remove` принимает дополнительный селектор запроса, позволяющий точно указать, какие документы удалить. С точки зрения API добавить больше нечего. Но есть кое-какие вопросы, касающиеся параллелизма и атомарности этих операций. Я отвечу на них в следующем разделе.

6.4.5. Параллелизм, атомарность и изолированность

Важно понимать, как в MongoDB устроен параллелизм. В версии MongoDB 2.0 стратегия блокировки довольно грубая – всем экземпляром `mongod` управляет единственная глобальная блокировка типа «один писатель, много читателей». Это означает, что в любой момент времени в базе данных может выполняться только одна операция записи или произвольное число операций чтения (но не то и другое одновременно). Звучит это гораздо хуже, чем есть на самом деле, потому что алгоритмы работы с этой блокировкой тщательно оптимизированы для обеспечения максимального параллелизма. Одна из оптимизаций состоит в том, что сервер хранит внутреннюю структуру данных, показывающую, какие документы находятся в памяти. Когда поступает запрос на чтение или запись документов, не находящихся в памяти, СУБД отдает процессор другим операциям на время, необходимое для загрузки документов в память.

Вторая оптимизация – уступка блокировок записи. Проблема в том, что если какая-то операция записи долго не завершается, то все остальные операции чтения и записи должны ждать. Блокировку захватывает любая операция вставки, обновления или удаления. Вставка обычно завершается быстро. Но обновление или удаление, затрагивающее много документов, а то и всю коллекцию, может занимать много времени. В текущей версии эта проблема решается за счет того

что длительная операция периодически уступает процессор другим читателям и писателям, то есть приостанавливается, освобождает блокировку, а через какое-то время возобновляет исполнение[9].

Однако при обновлении и удалении документов уступка процессора может оказаться палкой о двух концах. Легко представить ситуацию, когда требуется обновить или удалить все документы, а только потом приступать к другим операциям. Для таких случаев предусмотрен специальный оператор `$atomic`, который запрещает уступку блокировки. Достаточно добавить его в селектор запроса:

```
db.reviews.remove({user_id: ObjectId('4c4b1476238d3b4dd5000001')
   {$atomic: true}})
```

То же самое относится и к множественному обновлению. Следующая команда обновляет несколько документов в полной изоляции от других операций:

```
db.reviews.update({$atomic: true}, {$set: {rating: 0}}, false, true)
```

Здесь мы присваиваем всем рейтингам значение 0. Поскольку операция выполняется изолированно, то она никогда не уступает блокировку, и, следовательно, на всем ее протяжении система остается согласованной[10].

6.4.6. Замечания о производительности обновления

Опыт показывает, что наличие умозрительного представления о том как обновление отражается на документе, находящемся на диске, помогает проектировать системы с более высокой производительностью. Первое, что необходимо понимать, – это до какой степени обновление можно назвать выполняемым «по месту». В идеале обновление должно затрагивать как можно меньшую часть хранимого BSON-документа, так как это обеспечивает максимальную эффективность. Но не всегда так бывает. Ниже я объясню, что этому мешает.

Есть три основных вида обновления документа на диске. Первый и самый эффективный – когда изменяется только одно значение, и размер всего BSON-документа остается прежним. Чаще всего так

---

[9] Впрочем, вся процедура уступки и возобновления происходит в течение нескольких миллисекунд, то есть речь не идет о длительном прерывании.

[10] Отметим, что если во время операции, выполняемой в режиме `$atomic`, возникает ошибка, то неявный откат не производится. Часть документов будет обновлена, часть останется в исходном состоянии.

происходит при выполнении оператора `$inc`. Он всего лишь инкрементирует целое число, и, значит, размер занимаемого им на диске места не изменяется. Число, представленное типом `int`, всегда занимает на диске четыре байта; длинные целые и числа с двойной точностью занимают по восемь байтов. При изменении значения числа дополнительное место не требуется, поэтому перезаписать на диске нужно только занимаемую числом область.

Второй вид обновления приводит к изменению размера или структуры документа. BSON-документ представлен массивом байтов, размер которого хранится в первых четырех байтах. Поэтому, если вы примените к документу оператор `$push`, то произойдут две вещи: увеличится размер и изменится структура. Следовательно, придется перезаписать весь документ на диске. Не то чтобы это было чудовищно неэффективно, но помнить об этом надо. Если в одной операции присутствует несколько операторов обновления, то документ будет перезаписан по одному разу для каждого оператора. Это тоже обычно не проблема, особенно если запись происходит в памяти. Но если документ очень велик, скажем порядка 4 МБ, и вы вставляете в массивы, являющиеся его частью, значения оператором `$push`, то серверу придется проделать много работы[11].

Последний вид обновления является следствием перезаписи документа. Если размер увеличивается настолько, что документ больше не помещается в выделенное ему на диске место, то его необходимо не только перезаписать, но и переместить на новое место. Если операция перемещения производится часто, то накладные расходы будут велики. MongoDB пытается сгладить эту проблему, динамически подстраивая коэффициент заполнения для каждой коллекции в отдельности. Это означает, что если в данной коллекции производилось много операций обновления с перемещением документов, то внутренний коэффициент заполнения будет увеличен. Смысл его в том, что размер каждого документа умножается на коэффициент, чтобы зарезервировать на диске место сверх строго необходимого для хранения данных. При такой стратегии количество перемещений в будущем может уменьшиться.

Чтобы узнать коэффициент заполнения для коллекции, выполните команду `stats`:

```
db.tweets.stats()
{
```

[11] Без слов понятно, что если вы намереваетесь часто выполнять обновления, то лучше проектировать модель так, чтобы документы были небольшими.

```
    "ns" : "twitter.tweets",
    "count" : 53641,
    "size" : 85794884,
    "avgObjSize" : 1599.4273783113663,
    "storageSize" : 100375552,
    "numExtents" : 12,
    "nindexes" : 3,
    "lastExtentSize" : 21368832,
    "paddingFactor" : 1.2,
    "flags" : 0,
    "totalIndexSize" : 7946240,
    "indexSizes" : {
    "_id_" : 2236416,
    "user.friends_count_1" : 1564672,
    "user.screen_name_1_user.created_at_-1" : 4145152
},
"ok" : 1 }
```

Для этой коллекции «твитов» коэффициент заполнения равен 1.2, то есть при вставке документа длиной 100 байтов MongoDB зарезервирует на диске 120 байтов. По умолчанию коэффициент заполнения равен 1, то есть дополнительное место не выделяется.

И еще одно предупреждение. Все приведенные выше соображения особенно справедливы для систем, в которых размер данных превышает объем оперативной памяти или ожидается высокая частота операций записи. Поэтому если вы собираетесь разрабатывать систему аналитики для сайта с высоким уровнем трафика, отнеситесь к этой информации со всей серьезностью.

6.5. Резюме

Это была очень насыщенная глава. Поначалу разнообразие видов обновления может повергнуть в панику, однако предоставляемая ими мощь должна вас успокоить. Язык обновления в MongoDB по своей развитости не уступает языку запросов. Простой документ обновить так же легко, как сложный, содержащий вложенные поддокументы. При необходимости отдельные документы можно обновлять атомарно, а в сочетании с командой `findAndModify` это позволяет строить транзакционные последовательности действий.

Если, прочитав эту главу, вы почувствовали, что готовы применить изложенную в ней информацию к своим задачам, то можете считать, что встали на путь превращения в гуру MongoDB.

Часть 3.

MongoDB — ПОСТИЖЕНИЕ МАСТЕРСТВА

Прочитав первые две части, вы должны неплохо представлять себе, как MongoDB выглядит с точки зрения разработчика. Теперь пора сменить роль. В этой, последней, части мы рассмотрим MongoDB с точки зрения администратора базы данных, то есть поведаем всё, что нужно знать о производительности, развертывании, отказоустойчивости и масштабируемости.

Чтобы добиться от MongoDB максимальной производительности, необходимо эффективно проектировать запросы и строить подходящие индексы. Этому посвящена глава 7. Вы узнаете, почему индексы так важны и каким образом оптимизатор запросов решает, как их использовать. Вы также научитесь работать с такими полезными инструментами, как объяснитель плана выполнения запроса и профилировщик.

Глава 8 посвящена репликации. В основном в ней рассказывается, как работают наборы реплик и как их следует развертывать для достижения высокой доступности и автоматического перехода на резервный ресурс в случае отказа. Кроме того, вы узнаете, как с помощью репликации масштабировать операции чтения и обеспечить долговечность записи.

Горизонтальная масштабируемость — заветная цель всех современных СУБД. MongoDB решает ее путем *сегментирования* данных (sharding). В главе 9 описана теория и практика сегментирования,

показано, как использовать эту технику, как учитывать ее при проектировании схемы и как производить развертывание.

В последней главе описываются тонкости развертывания и администрирования. Мы дадим некоторые рекомендации, относящиеся к оборудованию и операционной системе. А затем покажем, как осуществляется резервное копирование, мониторинг и поиск неполадок в кластерах MongoDB.

ГЛАВА 7. Индексирование и оптимизация запросов

В этой главе:

- Основы теории индексирования.
- Управление индексами.
- Оптимизация запросов.

Важность индексов невозможно переоценить. Имея подходящие индексы, MongoDB может эффективно использовать оборудование и быстро обслуживать запросы. Плохие индексы приводят к противоположному результату: медленному выполнению запросов и недоиспользованию оборудования. Поэтому всякий, кто хочет работать с MongoDB эффективно, должен понимать принципы индексирования.

Однако для многих разработчиков индексы – этой тайна, покрытая мраком. Так быть не должно. После прочтения этой главы у вас в голове сложится модель, позволяющая правильно рассуждать об индексах. Чтобы достичь этой цели, мы начнем со скромного мысленного эксперимента. А затем рассмотрим базовые понятия индексирования и дадим обзор B-деревьев – структуры данных, лежащей в основе индексов в MongoDB.

Далее мы перейдем к практическому использованию индексов. Мы обсудим уникальные, разреженные и многоключевые индексы и дадим советы по их администрированию. После этого мы углубимся в оптимизацию запросов, опишем, как работает команда `explain()` и научимся дружить с оптимизатором.

7.1. Теория индексирования

Будем продвигаться постепенно, начав с аналогии и закончив изложением некоторых существенных деталей реализации в MongoDB. Попутно я определю ряд важных терминов и приведу соответствующие примеры. Если вы незнакомы с составными индексами, виртуальной памятью и индексными структурами данных, то этот раздел окажется для вас весьма поучительным.

7.1.1. Мысленный эксперимент

Чтобы разобраться в индексировании, необходима какая-то мысленная картина. Представьте себе кулинарную книгу. И не какую-нибудь, а толстенный сборник, страниц эдак на 5000, в котором есть самые вкусные рецепты на все случаи жизни, из любой кухни и на любой сезон, с указанием самых лучших ингредиентов, которые можно раздобыть дома. Это будет всем книгам книга. Назовем ее «Кухней от А до Я».

Но у этой лучшей в мире кулинарной книги есть два мелких недостатка. Во-первых, рецепты расположены в случайном порядке. Так, на странице 3475 приведен рецепт тушеной утки по-австралийски, а на странице 2 – салата «Жаркая Ямайка».

Но с этим мы бы как-нибудь справились, если бы не вторая беда: в «Кухне от А до Я» нет алфавитного указателя.

Отсюда первый вопрос: если нет указателя, то как найти рецепт картошки с розмарином? Единственный выход – листать книгу с самого начала, пока не дойдете до искомого. Если рецепт находится на 3973-ей странице, то именно столько страниц и придется просмотреть. В худшем случае, когда нужный рецепт напечатан на последней странице, вам предстоит пролистать всю книгу.

Это безумие. Нет, надо строить указатель.

Можно придумать разные способы поиска рецепта, но, наверное, начать проще всего с названия. Если составить список всех рецептов, расположив их по алфавиту и выписав рядом номер страницы, то получится, указатель, построенный названию рецепта. Вот несколько возможных строк:

- смородиновый мусс;
- стерлядь по-царски;
- суфле из тибетского яка.

Зная название рецепта (или хотя бы его начальные буквы), вы сможете быстро найти его в указателе. Если никаких других способов поиска рецепта не предполагается, то задача решена.

Но это маловероятно, так как полезно бывает искать рецепты, в которые, например, входят ингредиенты, имеющиеся у вас в кладовке. Или, скажем, по виду кухни. Тогда потребуются дополнительные указатели, которые отныне будем называть индексами.

Отсюда второй вопрос. Если имеется только индекс по названиям рецептов, то как найти все рецепты блюд из курицы? Раз подходящего индекса нет, то снова придется листать всю книгу, все 5000 страниц. Это относится и к поиску по ингредиентам или по виду кухни. Стало быть, надо построить еще один индекс – на этот раз по ингредиентам. В нем будут в алфавитном порядке перечислены все ингредиенты и для каждого указаны номера страниц, где встречаются содержащие этот ингредиент рецепты. Простейший индекс по ингредиентам мог бы выглядеть так:

- кабачок: 2, 47, 88, 89, 90, 275...
- кленовый сироп: 1001, 1050, 2000, 2133...
- корица: 7, 9, 80, 81, 82, 83, 84...
- кэшью: 3, 20, 42, 88, 103, 1215...

Вы о таком индексе мечтали? Он вообще полезен или нет?

Этот индекс годится, если вам нужен только список рецептов, содержащих данный ингредиент. Но если требуется *еще* какая-то информация о рецепте, то все равно придется заняться перелистыванием – зная номера страниц, на которых упоминается кабачок, вы должны будете открыть каждую, чтобы узнать, как рецепт называется и к какой кухне относится. Это, конечно, лучше, чем листать всю книгу, но можно пойти еще дальше.

Представьте, к примеру, что несколько месяцев назад вы случайно наткнулись в «Кухне от А до Я» на замечательный рецепт приготовления курочки, вот только забыли, как он называется. А теперь у вас есть два индекса: по названию и по ингредиентам. Можете вы придумать, как эти индексы объединить, чтобы найти вожделенный рецепт курочки?

На самом деле, это невозможно. Если вы начнете с индекса по названиям, но самого названия не помните, то поиск в этом индексе окажется ничем не лучше перелистывания всей книги. Если же начать с индекса по ингредиентам, то у вас будет список номеров возможных страниц, но эти номера никак не увязаны с индексом по названию.

Поэтому в данном случае можно воспользоваться только одним индексом, причем индекс по ингредиентам выглядит перспективнее.

Один индекс на каждый запрос. Пользователи нередко полагают, что для выполнения запроса, в котором участвуют два поля, можно будет воспользоваться двумя разными индексами. Такой алгоритм существует: найти в каждом индексе номера страниц, соответствующих поисковому термину, а затем взять объединение этих страниц и найти в нем рецепты, содержащие оба термина. На многих страницах не окажется ничего подходящего, но все же общее количество просмотренных страниц удастся сократить. В некоторых СУБД этот алгоритм применяется, но не в MongoDB. А даже если бы он и был реализован, то все равно поиск по составному индексу всегда оказался бы эффективнее. Поэтому запомните, что для выполнения любого запроса используется не более одного индекса и, если вы хотите опрашивать несколько полей, что стройте по ним составной индекс.

Так что же делать? К счастью, решение проблемы потерявшегося рецепта существует, и связано оно с использованием составных индексов.

Оба созданных выше индекса были построены по одному ключу, то есть по одному полю из каждого рецепта. А теперь мы построим для «Кухни от А до Я» индекс по двум ключам. Индексы, построенные по нескольким ключам, называются *составными*.

В данном случае составной индекс будет построен по ингредиентам и названию рецепта – *именно в таком порядке*. Назовем этот индекс `ingredient-name`. На рис. 7.1 изображена часть этого индекса.

Cashews
- **Cashew Marinade**
  - 1,215
- **Chicken with Cashews**
  - 88
- **Rosemary-Roasted Cashews**
  - 103

Cauliflower
- **Bacon Cauliflower Salad**
  - 875
- **Lemon-baked Cauliflower**
  - 89
- **Spicy Cauliflower Cheese Soup**
  - 47

Currants
- **Creamed Scones with Currants**
  - 2,000
- **Fettuccini with Glazed Duck**
  - 2,133
- **Saffron Rice with Currants**
  - 1,050

Рис. 7.1. Составной индекс для кулинарной книги

Для человека ценность этого индекса очевидна. Теперь можно сначала найти ингредиент, а потом содержащий его рецепт, причем достаточно вспомнить хотя бы начальные буквы названия. Для компьютера этот индекс также полезен, потому что позволяет не просматривать все рецепты, в которые входит указанный ингредиент. А уж в случае кулинарной книги, где приведены сотни (а то и тысячи) рецептов блюд из курицы, такой составной индекс просто бесценен. Понимаете почему?

Отметим одну вещь: в составных индексах порядок ключей имеет значение. Подумайте, как выглядел бы индекс `name-ingredient` с другим порядком ключей. Можно ли его использовать вместо рассмотренного выше?

Безусловно, нет. В этом случае, если вы знаете название рецепта, то поиск ограничен только этим рецептом, по сути всего одной страницей книги. Если бы вы попытались с помощью этого индекса найти рецепт кэшью в маринаде с ингредиентом «банан», то индекс сказал бы, что такого рецепта не существует. Но нас-то интересует другой сценарий – известен ингредиент, а не название рецепта.

Итак, сейчас над кулинарной книгой построено три индекса: по названиям рецептов (`recipe`), по ингредиентам (`ingredient`) и составной индекс (`ingredient-name`). Но тогда индекс `ingredient` по одному ключу можно спокойно выкинуть. Почему? Да потому что для поиска по ингредиенту можно воспользоваться индексом `ingredient-name`. То есть, если вы знаете ингредиент, то можете извлечь из составного индекса номера всех страниц, где этот ингредиент упоминается. Взгляните еще раз на рисунок и поймете, почему это так.

В этом разделе я хотел предложить распространенную метафору для тех читателей, которым необходимо составить в уме модель индексов. Эта метафора позволяет сформулировать несколько простых эвристических правил:

1. Индексы значительно сокращают объем работы, необходимой для выборки документов. Без подходящих индексов единственный способ выполнить запрос заключается в том, чтобы последовательно просмотреть все документы и для каждого проверить, удовлетворяются ли условия запроса. Часто эту процедуру называют полным сканированием коллекции.
2. При выполнении запроса можно использовать только один индекс[1]. Если в запросе участвует несколько ключей (например, ингредиент и название рецепта), то для его выполнения лучше подходит составной индекс по этим ключам.
3. Индекс по ингредиентам `ingredients` можно и нужно исключить, если уже существует составной индекс `ingredient-name`. Вообще, если имеется составной индекс `a-b`, то индекс по ключу `a` избыточен[2].

[1] Исключение составляют запросы с оператором `$or`. Но вообще говоря в MongoDB использовать сразу несколько индексов невозможно и даже нежелательно.

[2] Однако из этого правила есть исключения. Если `b` – индекс по нескольким ключам, то имеют смысл оба индекса `a-b` и `a`.

4. Порядок ключей в составном индексе имеет значение.

Помните, что аналогия с кулинарной книгой все же ограничена. Это всего лишь модель для понимания индексов, но она не вполне точно описывает работу реальных индексов в MongoDB. В следующем разделе мы дополним сформулированные выше эвристические правила и рассмотрим индексирование в MongoDB более подробно.

7.1.2. Основные понятия индексирования

В описанном выше мысленном эксперименте мы намекнули на целый ряд важных понятий индексирования. Здесь и далее мы раскроем эти идеи.

Индексы с одним ключом

В этом случае в каждой записи индекса хранится ровно одно значение из проиндексированного документа. Хороший пример – индекс по полю `_id`. Поскольку это поле проиндексировано, то идентификатор каждого документа хранится также в индексе, что позволяет быстро искать документ по значению `_id`.

Составные индексы

В текущей версии MongoDB умеет использовать только один индекс при обработке каждого запроса[3]. Однако зачастую необходимо предъявлять запросы, содержащие несколько атрибутов, и хотелось бы, чтобы они выполнялись максимально эффективно. Представьте, например, что вы построили два индекса над коллекцией *products* из примера интернет-магазина: по полю `manufacturer` и по полю `price`. Тем самым вы создали две совершенно различные структуры данных, упорядоченные, как показано на рис. 7.2.

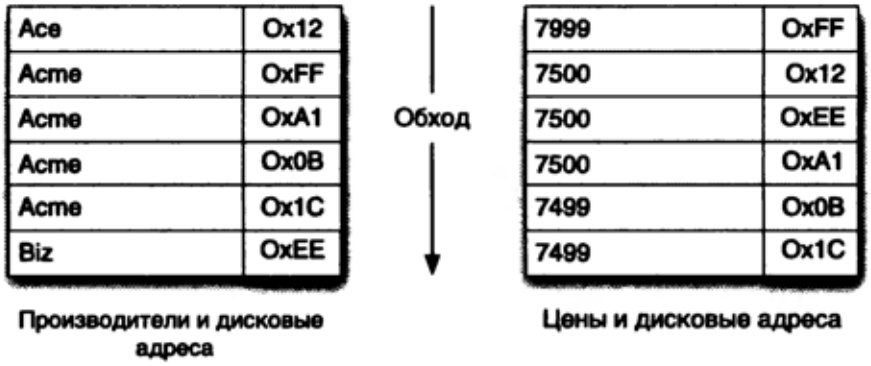


Рис. 7.2. Обход индекса с одним ключом

[3] Из этого правила все же есть редкие исключения. Например, при выполнении запросов с оператором `$or` для каждой части запроса могут использоваться разные индексы. Тем не менее, для одной части используется только один индекс.

А теперь рассмотрим такой запрос:

```
db.products.find({'details.manufacturer': 'Acme',
   'pricing.sale': {$lt: 7500}})
```

Здесь мы хотим найти все товары компании Acme с ценой ниже 75,00 долларов. Если этот запрос выполняется, когда есть только индексы с одним ключом – по полям `manufacturer` и `price`, – то использован будет только один. Оптимизатор выберет тот, что наиболее эффективен, но идеального результата все равно не получится. Чтобы применить к обработке запроса оба индекса, надо было бы независимо обойти обе структуры, получить из каждого множество дисковых адресов подходящих документов и построить объединение этих множеств. В настоящее время MongoDB не поддерживает такую стратегию, отчасти потому что составной индекс гораздо эффективнее.

Составным называется индекс, в каждой записи которого хранится более одного ключа. *Составной индекс* по полям `manufacturer` и `price` упорядочен, как показано на рис. 7.3.

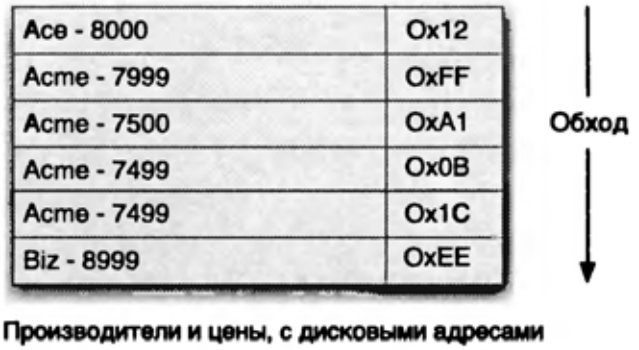


Рис. 7.3. Обход составного индекса

Теперь, чтобы выполнить приведенный выше запрос, оптимизатор должен всего лишь найти в индексе первую запись, для которой производитель – Acme, а цена равна $75,00. Для выборки остальных результатов достаточно последовательно просматривать записи индекса, пока не встретится запись, в которой значение поля `manufacturer` отлично от *Acme*.

Необходимо отметить два момента, касающихся использования этого индекса при обработке запроса. Во-первых, порядок ключей имеет значение. Если построить составной индекс, в котором первым ключом является цена, а вторым – производитель, то запрос выполнялся бы гораздо медленнее. Не понимаете, почему? Взгляните на порядок записей в таком индексе (рис. 7.4).

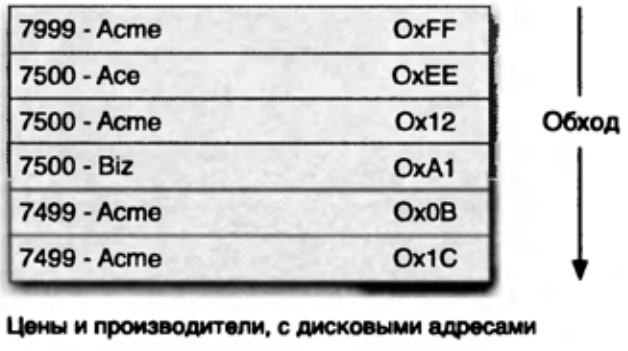


Рис. 7.4. Обход составного индекса с измененным порядком ключей

Ключи необходимо сравнивать в порядке их следования в индексе. К сожалению, этот индекс не позволяет легко обойти все товары, произведенные Acme. Поэтому единственный способ выполнить запрос – просмотреть *все* товары с ценой ниже 75,00 долларов и выбрать из них те, что изготовлены Acme. А теперь представьте, что коллекция содержит миллион товаров, причем все они дешевле 100,00 долларов и равномерно распределены по цене. При таких условиях для выполнения запроса пришлось бы просмотреть 750 000 записей индекса. Тогда как при использовании первоначального составного индекса, в котором производитель предшествует цене, просматривается ровно столько записей, сколько будет возвращено. А все потому, что отыскав в индексе запись (Acme – 7500), нам останется только последовательно просмотреть записи с тем же производителем.

Таким образом, порядок ключей в составном индексе действительно имеет значение. Если это понятно, то следует еще уяснить себе, почему мы предпочли первый порядок, а не второй. Наверное, это очевидно из рисунков, но можно взглянуть на проблему и иначе. Посмотрите еще раз на запрос: термы в двух его частях описывают различные условия. При сравнении с производителем нас интересует точное равенство, а при сравнении с ценой – значения, большие 7500. В общем случае для выполнения запроса, в котором одна часть определяет точное равенство, а другая – диапазон, следует предпочесть составной индекс, в котором ключ, соответствующий диапазону, идет вторым. Мы еще вернемся к этой мысли в разделе, посвященном оптимизации запросов.

Эффективность индексов

Хотя индексы очень важны с точки зрения скорости выполнения запросов, с обслуживанием каждого индекса связаны определенные накладные расходы. Откуда они берутся, понятно. Всякий раз как в коллекцию добавляется новый документ, его необходимо включить также во все построенные над этой коллекцией индексы. Стало быть, если над коллекцией построено 10 индексов, значит, при любой вставке потребуется модифицировать 10 разных структур. И это относится вообще к любой операции записи, будь то удаление или обновление полей документа, по которым построен индекс.

Для приложений, ориентированных главным образом на чтение, затраты на обслуживание индексов почти всегда оправданы. Просто имейте в виде, что индексы – удовольствие не бесплатное, поэтому выбирать их следует с оглядкой. Иными словами, следите за тем,

чтобы все индексы действительно использовались, а от избыточных избавляйтесь. Узнать, какие индексы лишние, позволит профилирование отправляемых приложением запросов; как это делается, я объясню ниже.

Но есть и еще один момент. Даже если все необходимые индексы имеются, они необязательно приведут к ускорению обработки запросов. Так происходит, когда индексы и рабочий набор данных не помещаются в оперативную память.

В главе 1 упоминалось, что MongoDB просит операционную систему спроецировать все файлы данных на память с помощью системного вызова `mmap()`. Начиная с этого момента, любые файлы данных, к которым относятся документы, коллекции и индексы, ОС загружает в память и выгружает из нее *страницами* по 4 КБ[4]. При запросе данных, находящихся в некоторой странице, ОС должна удостовериться, что эта страница присутствует в памяти. Если это не так, то возбуждается исключение *отсутствия страницы* (страничный отказ), в ответ на которое диспетчер памяти загружает страницу с диска.

Если физической памяти достаточно, то все файлы данных рано или поздно будут в нее загружены. Изменения, произведенные в памяти – например, операцией записи, – асинхронно сбрасываются на диск операционной системой, но сама операция записи завершается быстро, так как имеет дело только с оперативной памятью. Если все данные помещаются в память, то мы имеем идеальную ситуацию, поскольку число обращений к диску минимально. В противном случае то и дело будут возникать страничные отказы. Следовательно, операционной системе придется часто обращаться к диску, что замедляет операции чтения и записи. В худшем случае, когда объем данных значительно больше объема оперативной памяти, может случиться, что любая операция чтения или записи приводит к *загрузке и последующей выгрузке* страницы. Это явление называется *пробуксовкой*; если до этого доходит, то производительность резко падает.

К счастью, такой ситуации сравнительно просто избежать. Как минимум, нужно сделать так, чтобы в память помещались все индексы. Именно поэтому так важно не создавать лишних индексов. При наличии таковых для их обслуживания понадобится больше памяти. По той же причине всякий индекс должен содержать лишь те ключи, что необходимы: иногда составной индекс по трем ключам бывает полезен, но помните, что он занимает больше места, чем простой индекс с одним ключом.

---

[4] 4 КБ – стандартный, но не универсальный размер страницы.

В идеале в память должны помещаться индексы *и* рабочий набор данных. Но оценить, сколько памяти требуется данной системе, не всегда легко. Суммарный размер индексов позволяет узнать команда `stats`. Но вот определить размер рабочего набора не так просто, поскольку он зависит от приложения. *Рабочим набором* называется подмножество данных, к которому часто адресуются запросы – как на чтение, так и на обновление. Пусть, например, у приложения имеется миллион пользователей. Если активна только половина, то размер рабочего набора для коллекции пользователей составляет половину от общего размера хранящихся в ней данных. Если же активны все пользователи, то рабочий набор совпадает со всей коллекцией.

В главе 10 мы вернемся к понятию рабочего набора и рассмотрим способы диагностики проблем с производительностью, вызванных оборудованием. А пока просто имейте в виду, что с каждым новым индексом связаны дополнительные накладные расходы, и следите за соотношением между суммарным размером индексов и рабочего набора и общим объемом оперативной памяти. Это позволит обеспечивать приемлемую производительность с увеличением количества данных.

7.1.3. B-деревья

Выше отмечалось, что в MongoDB индексы представлены *B-деревьями*. B-деревья вездесущи (см. http://mng.bz/wQfG), они используются в СУБД для хранения данных и индексов по крайней мере с конца 1970-х годов[5]. Если вам доводилось работать с другими СУБД, то вы, скорее всего, знакомы с различными последствиями использования B-деревьев. И это хорошо, потому что многое из того, что вам известно об индексировании, применимо и к MongoDB. Но даже если вы ничего не знаете о B-деревьях, тоже не страшно; в этом разделе будут изложены наиболее важные сведения.

У B-деревьев есть два главных свойства, которые делают их идеальной структурой для организации индексов в базах данных. Во-первых, они обеспечивают эффективное выполнение самых разных запросов, в том числе поиск по совпадению, по диапазону, с сортировкой, по совпадению префиксов и с применением только самих индексов. Во-вторых, B-деревья остаются сбалансированными после вставки и удаления ключей.

---

[5] В MongoDB B-деревья применяются только для организации индексов; коллекции хранятся в виде двусвязных списков.

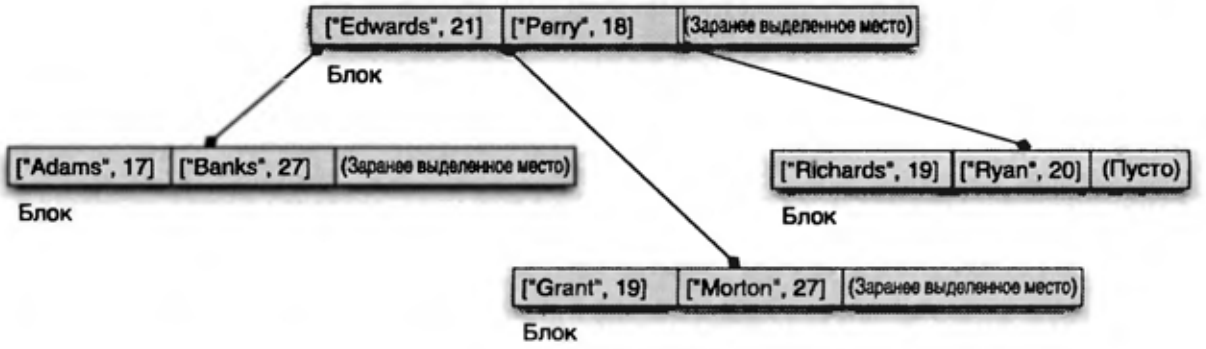


Рис. 7.5. Структура простого B-дерева

Мы рассмотрим простое представление B-дерева, а затем обсудим некоторые принципы, о которых следует помнить. Итак, представьте, что имеется коллекция пользователей и над ней построен составной индекс по фамилии и возрасту[6]. На рис. 7.5 изображено абстрактное представление B-дерева для такого индекса.

Вы, конечно, догадались, что B-дерево – это древовидная структура. В каждом узле дерева может храниться несколько ключей. Так, в корневом узле дерева, изображенного на рисунке, два ключа, и каждый из них – это BSON-объект, представляющий индексированное значение из коллекции `users`. Поэтому, прочитав корневой узел, вы найдете две записи с ключами – о пользователе Edwards 21 года и о пользователе Perry 18 лет. В каждой записи хранятся также два указателя: на файл данных, в котором хранится сам документ, и на дочерний узел, в котором хранятся ключи, меньшие наименьшего ключа в данном узле.

Отметим еще, что в каждом узле имеется нераспределенная область. В MongoDB размер вновь создаваемых узлов B-дерева составляет 8192 байта, то есть на практике в одном узле может храниться несколько сотен ключей. Сколько именно, зависит от среднего размера ключа; в нашем примере он равен примерно 30 байтам. Максимальный размер ключа в MongoDB версии 2.0 составляет 1024 байта. Добавим к средней длине ключа накладные расходы на каждый ключ (18 байтов) и учтем накладные расходы на сам узел (40 байтов) – в итоге получится около 170 ключей в одном узле[7].

Это существенно, поскольку пользователям часто интересно, почему индекс занимает именно столько места. Теперь вы знаете, что размер узла составляет 8 КБ, и можете оценить, сколько в него поместится ключей. При этом учтите, что обычно узлы B-деревьев по умолчанию заполняются на 60% – это осознанное решение.

---

[6] Индекс по фамилии и возрасту – вещь экзотическая, но для иллюстрации идей вполне подходит.

[7] $(8192 - 40)/(30 + 18) = 169.8$

Если все вышеизложенное понятно, то теперь, помимо поверхностной умозрительной картины B-деревьев, вы представляете, каким образом в них выделяется место и как они обслуживаются. Это еще одно подтверждение тезиса о том, что индексы не бесплатны. Подходите к их выбору с умом.

7.2. Индексирование на практике

Теперь оставим теорию и уточним наши представления об индексировании в MongoDB. А затем перейдем к тонкостям администрирования индексов.

7.2.1. Типы индексов

Базовая структура всех индексов в MongoDB одинакова, но тем не менее они могут обладать различными свойствами. Так, часто используются уникальные, разреженные и многоключевые индексы. Ниже все они описываются более подробно[8].

Уникальные индексы

Для создания уникального индекса следует задать параметр `unique`:

```
db.users.ensureIndex({username: 1}, {unique: true})
```

В уникальном индексе не может быть двух одинаковых ключей. Так, при попытке вставить в коллекцию `users` документ с уже существующим именем пользователя будет возбуждено следующее исключение:

```
E11000 duplicate key error index:
gardening.users.$username_1 dup key: { : "kbanker" }
```

Если используется какой-нибудь драйвер, то это исключение будет перехвачено только, если вставка производится в безопасном режиме драйвера. Подробности см. в главе 3.

Если коллекция нуждается в уникальном индексе, то лучше всего построить его до того, как будут добавлены какие-нибудь данные. В этом случае уникальность гарантируется с самого начала. Если же вы попробуете построить уникальный индекс над коллекцией, в ко-

[8] MongoDB поддерживает также пространственные индексы, но они настолько специфичны, что я расскажу о них отдельно, в приложении E.

торой уже есть данные, то рискуете получить ошибку в ситуации, когда в коллекции имеются одинаковые ключи, – тогда индекс не будет создан.

Если все-таки возникает необходимость построить уникальный индекс над непустой коллекцией, то вариантов два. Во-первых, можно раз за разом пытаться построить индекс, последовательно удаляя документы с повторяющимися ключами, о которых говорится в сообщение об ошибке. Если данные не являются критически важными, то можно поручить эту работу СУБД, задав параметр `dropDups`. Например, если в коллекции `users` уже есть данные и вы не боитесь, что будут удалены документы с повторяющимися ключами, то можете построить индекс такой командой:

```
db.users.ensureIndex({username: 1}, {unique: true, dropDups: true})
```

Отметим, что заранее сказать, какие именно документы с ключами-дубликатами удалит сервер, невозможно, так что применяйте эту возможность с осторожностью.

Разреженные индексы

По умолчанию индексы плотные. Это означает, что каждому документу в индексированной коллекции соответствует запись в индексе, даже если в документе нет индексируемого ключа. Вспомните, к примеру, коллекцию `products` из модели данных интернет-магазина и представьте, что строится индекс по атрибуту товара `category_ids`. Предположим теперь, что имеются товары, не отнесенные ни к какой категории. Несмотря на это, в индексе `category_ids` для них все равно будет присутствовать пустая запись. Найти все такие пустые записи позволяет следующий запрос:

```
db.products.find({category_ids: null})
```

При поиске товаров, не отнесенных ни к какой категории, оптимизатор запросов все же сможет воспользоваться индексом `category_ids`.

Однако есть два случая, когда плотный индекс нежелателен. Первый – когда требуется уникальный индекс по полю, которое в некоторых документах может отсутствовать. Например, индекс по полю товара `sku` определенно должен быть уникальным. Но допустим, что по какой-то причине товары вводятся в систему до того, как им приписан артикул. Тогда если индекс по `sku` уникален, то вставка первого товара закончится успешно, а всех последующих – ошибкой, потому что в индексе уже существует запись с пустым ключом `sku`. В такой ситуации плотный индекс не годится, а необходим *разреженный*.

В разреженном индексе представлены только документы, для которых индексируемый ключ имеет какое-то значение. Для создания такого индекса нужно задать параметр `{sparse: true}`. Например, чтобы создать уникальный разреженный индекс по полю `sku`, следует написать:

```
db.products.ensureIndex({sku: 1}, {unique: true, sparse: true})
```

Существует еще один случай, когда желательно иметь разреженный индекс: если в коллекции имеется много документов, не содержащих индексируемого ключа. Предположим, к примеру, что на вашем сайте разрешены анонимные отзывы. В этом случае в половине отзывов поле `user_id` может отсутствовать, и, если это поле индексировано, то половина записей в индексе окажется пустой. Это неэффективно по двум причинам. Во-первых, увеличивается размер индекса, а, во-вторых, при добавлении и удалении документов с пустым полем `user_id` индекс придется обновлять.

Если вы ожидаете, что запросы на поиск анонимных отзывов будут предъявляться редко (или вообще никогда), то по полю `user_id` имеет смысл построить разреженный индекс:

```
db.reviews.ensureIndex({user_id: 1}, {sparse: true})
```

Теперь индексируются только отзывы, оставленные пользователем с известным `user_id`.

Многоключевые индексы

В предыдущих главах нам встречались примеры индексирования полей со значениями-массивами[9]. Это возможно благодаря так называемым *многоключевым индексам* (multikey index), в которых допускается несколько записей, указывающих на один и тот же документ. Зачем это нужно, легко понять на простом примере. Пусть имеется документ с несколькими тегами:

```
{  name: "Wheelbarrow",
   tags: ["tools", "gardening", "soil"]
}
```

Если создать индекс по полю `tags`, то в нем будут присутствовать все значения, перечисленные в массиве тегов. Следовательно, для поиска документа можно указать любой из его тегов. Именно в этом и состоит идея многоключевого индекса: несколько ключей (записей), указывающих на один документ.

---

[9] Вспомните, например, об идентификаторах категорий.

Многоключевые индексы поддерживаются в MongoDB без каких-либо специальных указаний. Если индексируемое поле содержит массив, то в индекс попадают все хранящиеся в этом массиве значения.

Осмысленное применение многоключевых индексов – важная составная часть проектирования схемы в MongoDB. Это должно быть очевидно из примеров в главах 4 – 6; дополнительные примеры будут приведены в разделе о паттернах проектирования в приложении B.

7.2.2. Администрирование индексов

Что касается администрирования индексов в MongoDB, то ваших прежних знаний может не хватить. В этом разделе мы подробно рассмотрим создание и удаление индексов, а также ряд вопросов, относящихся к сжатию и резервному копированию.

Создание и удаление индексов

Вы уже приобрели опыт построения нескольких индексов, так что никаких тайн в синтаксисе команды их создания не осталось. Нужно просто вызвать вспомогательный метод создания – в оболочке или в объемлющем языке – передав ему документ с описанием индекса, который будет сохранен в специальной системной коллекции `system.indexes`.

Хотя, как правило, для создания индекса проще использовать вспомогательный метод, можно вставить спецификацию индекса и вручную (именно это и делает вспомогательный метод). Нужно только указать в нем минимальный набор ключей: `ns`, `key` и `name`. Ключ `ns` описывает пространство имен, ключ `key` – одно или несколько индексируемых полей, а ключ `name` – имя самого индекса. Здесь же можно задать дополнительные параметры, например `sparse`. Вот как создается разреженный индекс над коллекцией `users`:

```
spec = {ns: "green.users", key: {'addresses.zip': 1}, name: 'zip'}
db.system.indexes.insert(spec, true)
```

Если при вставке не возникнет ошибок, то индекс будет создан, и его можно запросить у коллекции `system.indexes`:

```
db.system.indexes.find()
{  "_id" : ObjectId("4d2205c4051f853d46447e95"), "ns" : "green.users",
   "key" : { "addresses.zip" : 1 }, "name" : "zip", "v" : 1 }
```

При работе с версией MongoDB 2.0 или более поздней вы увидите дополнительный ключ `v`. В нем хранится номер версии, который

позволяет распознать изменения внутреннего формата в последующих версиях, но для разработчиков приложений он не представляет особого интереса.

Вы можете решить, что для удаления индекса достаточно удалить описывающий его документ из коллекции `system.indexes`, однако эта операция запрещена. Вместо этого следует пользоваться встроенной командой `deleteIndexes`. Как и в случае создания, существуют вспомогательные методы для удаления индексов, но при желании можно выполнить эту команду самостоятельно. В качестве аргумента ей передается документ, содержащий имя коллекции и либо имя индекса, либо *, если нужно удалить все построенные над ней индексы. Чтобы вручную удалить только что созданный индекс, выполните следующую команду:

```
use green
db.runCommand({deleteIndexes: "users", index: "zip"})
```

Как правило, для создания и удаления индексов применяются вспомогательные методы, встроенные в оболочку:

```
use green
db.users.ensureIndex({zip: 1})
```

Для получения спецификаций индексов служит метод `getIndexSpecs()`:

```
> db.users.getIndexSpecs()
[
   {
      "v" : 1,
      "key" : {
         "_id" : 1
      },
      "ns" : "green.users",
      "name" : "_id_"
   },
   {
      "v" : 1,
      "key" : {
         "zip" : 1
      },
      "ns" : "green.users",
      "name" : "zip_1"
   }
]
```

Наконец, удалить индекс можно с помощью метода `dropIndex()`. Отметим, что ему необходимо указать то имя индекса, которое было задано в спецификации:

```
use green
db.users.dropIndex("zip_1")
```

Это всё, что можно сказать о создании и удалении индексов. Если вы хотите знать, чего ожидать от созданного индекса, читайте дальше.

Построение индексов

Обычно индексы объявляются перед запуском приложения в эксплуатацию. Это позволяет заполнять индекс постепенно, по мере вставки данных. Но существует два случая, когда создание индекса имеет смысл отложить. Первый – импорт большого объема данных до начала эксплуатации. Например, если вы переписываете приложение для MongoDB и хотите поместить в базу данных информацию о пользователях из прежнего хранилища. Можно, конечно, создать индекс и заранее, но если сделать это после импорта данных, то вы гарантированно получите в начале идеально сбалансированный и сжатый индекс. К тому же и время его построения уменьшится.

Второй (более очевидный) случай создания индекса над непустым набором данных – оптимизация новых запросов.

Но какова бы ни была причина, этот процесс отнюдь не всегда приятный. Если набор данных велик, то построение индекса может занимать несколько часов, а то и дней. Следить за ходом процесса можно по журналам MongoDB. Возьмем для примера набор данных, с которым будем работать в следующем разделе. Сначала объявим необходимый индекс:

```
db.values.ensureIndex({open: 1, close: 1})
```

Будьте осторожны при объявлении индексов. Поскольку объявить индекс так просто, то можно случайно запустить его построение. Если набор данных велик, то индекс будет строиться долго. А в производственной системе это может обернуться кошмаром, потому что простого способа прервать процесс построения нет. Если с вами такое случится, то, возможно, придется перейти на резервный узел – если, он конечно, имеется. Но самый разумный совет – считать построение индекса своего рода миграцией базы данных и никогда не объявлять их в приложении автоматически.

Построение индекса происходит в два этапа. На первом индексируемые значения сортируются. Вставка отсортированного набора в B-дерево производится гораздо эффективнее. Следить за процессом сортировки позволяет отношение числа уже отсортированных документов к их общему числу:

```
[conn1] building new index on { open: 1.0, close: 1.0 } for stocks.values
1000000/4308303 23%
2000000/4308303 46%
3000000/4308303 69%
4000000/4308303 92%
Tue Jan 4 09:59:13 [conn1] external sort used : 5 files in 55 secs
```

На втором шаге отсортированные значения вставляются в индекс. Ход процесса индицируется точно так же, и по его завершении отображается общее время построения – как время, затраченное на вставку в `system.indexes`:

```
1200300/4308303 27%
2227900/4308303 51%
2837100/4308303 65%
3278100/4308303 76%
3783300/4308303 87%
4075500/4308303 94%
Tue Jan 4 10:00:16 [conn1] done building bottom layer, going to commit
Tue Jan 4 10:00:16 [conn1] done for 4308303 records 118.942secs
Tue Jan 4 10:00:16 [conn1] insert stocks.system.indexes 118942ms
```

Помимо изучения журнала MongoDB, проследить за ходом построения индекса позволяет команда оболочки `currentOp()`[10]:

```
> db.currentOp()
{
   "inprog" : [
      {
         "opid" : 58,
         "active" : true,
         "lockType" : "write",
         "waitingForLock" : false,
         "secs_running" : 55,
         "op" : "insert",
         "ns" : "stocks.system.indexes",
         "query" : {
         },
         "client" : "127.0.0.1:53421",
         "desc" : "conn",
         "msg" : "index: (1/3) external sort 3999999/4308303 92%"
      }
   ]
}
```

Последнее поле, `msg`, касается процедуры построения индекса. Обратите также внимание на поле `lockType`, которое говорит, что на

[10] Если построение индекса запущено из оболочки MongoDB, то необходимо будет открыть другой экземпляр оболочки и запустить в нем команду `currentOp`. Дополнительные сведения об этой команде см. в главе 10.

время построения индекса захватывается блокировка записи. Это означает, что больше никто не может ни читать, ни записывать в базу данных. Понятно, что в производственной системе это крайне нежелательно, потому-то построение больших индексов так досаждает. Ниже мы рассмотрим два подхода к решению этой проблемы.

Индексирование в фоновом режиме

Если система работает в производственном режиме и доступ к базе данных нельзя приостанавливать даже на время, то можно строить индекс в фоновом режиме. Хотя при этом блокировка записи все равно захватывается, но периодически задача будет уступать ее другим читателям и писателям. Если приложение сильно нагружает сервер MongoDB, то фоновое индексирование приведет к снижению производительности, но в некоторых случаях это приемлемо. Например, если вы знаете, что индекс можно построить в период минимального трафика, то фоновое индексирование – неплохой выход из положения.

Для фонового построения индекса задайте при его объявлении параметр `{background: true}`. Например, показанный выше индекс в фоновом режиме строится так:

```
db.values.ensureIndex({open: 1, close: 1}, {background: true})
```

Автономное индексирование

Если набор данных настолько велик, что его индексирование занимает много часов, то придется прибегнуть к альтернативному плану. Обычно это сводится к переводу одного из узлов-реплик в автономный режим, построению на нем индекса и последующей синхронизации этого узла с главным. Когда синхронизация завершится, этот узел можно сделать главным, а другой вывести в автономный режим и построить индекс на нем. Такая тактика предполагает, что журнал репликации достаточно велик и не переполнится за время, необходимое для индексирования. В следующей главе мы будем подробно говорить о репликации и рассмотрим, как спланировать подобную миграцию.

Резервное копирование

Раз на построение индексов положено столько сил, то неплохо было бы сделать их резервную копию. К сожалению, не все методы резервного копирования учитывают индексы. Так, утилиты `mongodump` и `mongorestore`, несмотря на многообещающие названия, сохраняют

только коллекции и объявления индексов. Это означает, что после выполнения `mongorestore` все индексы, объявленные для восстановленных коллекций, будут построены заново. И, если набор данных велик, то время их построения может оказаться неприемлемым.

Следовательно, если вы решите включить индексы в резервную копию, то должны будете включить туда же и сами файлы данных MongoDB. Подробнее об этом и о резервном копировании вообще см. главу 10.

Сжатие

Если приложение интенсивно обновляет данные или выполняет много операций удаления, то индекс фрагментируется. B-деревья до некоторой степени способны к «зарастанию», но полностью компенсировать массивное удаление получается не всегда. Основной признак фрагментированного индекса – размер, значительно превышающий ожидаемый. Из-за фрагментации индекс может потреблять больше памяти, чем необходимо. В таких случаях следует подумать о том, чтобы перестроить один или несколько индексов. Для этого нужно удалить и заново создать отдельные индексы или выполнить команду `reIndex`, которая перестроит все индексы над указанной коллекцией:

```
db.values.reIndex();
```

Но дважды подумайте, прежде чем использовать эту команду: она захватывает блокировку на все время перестроения, делая базу данных временно недоступной. Переиндексирование лучше выполнять в автономном режиме на узле-реплике, как описано выше. Отметим, что команда `compact`, рассматриваемая в главе 10, также перестраивает индексы над коллекцией, для которой запущена.

7.3. Оптимизация запросов

Оптимизацией называется процедура выявления медленных запросов, выяснения причин медленного выполнения и принятия мер для его ускорения. В этом разделе мы рассмотрим все шаги оптимизации, так что в конце вы будете знать, как подходить к отладке проблематичных запросов в своей базе данных MongoDB. Но я сразу хочу предупредить, что описанные ниже приемы не позволят решить *любую* проблему, связанную с производительностью. Причин для медленного выполнения запроса очень много. Из наиболее типичных можно назвать неправильно спроектированное приложение, неудачную

модель данных и недостаточно мощное оборудование. Для их устранения иногда требуется потратить немало времени. Я буду говорить только о том, как оптимизировать запросы путем их реструктурирования и построения полезных индексов. Кроме того, я подскажу, в каком направлении двигаться, если этого оказывается недостаточно.

7.3.1. Выявление медленных запросов

Если вам кажется, что приложение для MongoDB часто «задумывается», то самое время заняться профилированием запросов. Любая строгая методика проектирования приложения должна включать аудит запросов, а учитывая, насколько просто это делается в MongoDB, никаких извинений пренебрежению аудитом быть не может. Хотя требования бывают разные, разумно предположить, что в большинстве приложений выполнение запроса не должно занимать больше 100 миллисекунд. В модуль протоколирования MongoDB это предположение встроено, поскольку он выводит предупреждения обо всех операциях, в том числе запросах, которые выполнялись более 100 мс. Поэтому журналы – первое место, где следует искать медленные запросы.

Маловероятно, что запросы к встречавшимся нам до сих пор наборам данных будут выполняться дольше 100 мс. Поэтому в примерах ниже мы воспользуемся набором данных о ежедневных сводках биржи NASDAQ. Если вы хотите проработать эти примеры, то необходимо загрузить их на свой компьютер. Для этого скачайте архив со страницы http://mng.bz/ii49. Затем распакуйте файл во временный каталог. На экране будут напечатаны следующие сообщения:

```
$ unzip stocks.zip
Archive: stocks.zip
creating: dump/stocks/
inflating: dump/stocks/system.indexes.bson
inflating: dump/stocks/values.bson
```

Теперь загрузите данные в базу командой

```
$ mongorestore -d stocks -c values dump/stocks
```

Этот набор данных достаточно велик, и с ним легко работать. В наборе представлены документы для некоторых котируемых на NASDAQ акций, и в каждом документе хранятся три цены – максимальная, минимальная и закрытия – а также общая сумма торгов за каждый день в течение 25 лет, начиная с 1983 года. При таком количестве и размере документов нетрудно получить предупреждения о

длительных запросах. Попробуйте запросить дату первого появления на бирже акций Google:

```
db.values.find({"stock_symbol": "GOOG"}).sort({date: -1}).limit(1)
```

Как вы убедитесь, этот запрос выполняется довольно долго. А заглянув в журнал MongoDB, вы обнаружите ожидаемое предупреждение:

```
Thu Nov 16 09:40:26 [conn1] query stocks.values
           ntoreturn:1 scanAndOrder reslen:210 nscanned:4308303
           { query: { stock_symbol: "GOOG" }, orderby: { date: -1.0 } }
           nreturned:1 4011ms
```

Информации много, и мы рассмотрим ее при обсуждении команды `explain()`. А пока обратите внимание на самые важные аспекты: запрос касался коллекции `stocks.values`; в селекторе запроса указано поле `stock_symbol`; производилась сортировка и, самое главное, общее время выполнения запроса составило целых 4 секунды (4011 мс).

Такие предупреждения нельзя оставлять без внимания. Они настолько значимы, что стоит время от времени выуживать их из журналов MongoDB. Это несложно сделать с помощью утилиты `grep`:

```
grep -E '([0-9])+ms' mongod.log
```

Если порог в 100 мс кажется вам слишком высоким, то можете снизить его, запустив сервер `mongod` с параметром `--slowms`. Если вы считаете, что «долго» – это больше 50 мс, то задайте такое значение параметра: `--slowms 50`.

Разумеется, поиск с помощью `grep` – не слишком систематический подход. Журналы MongoDB можно использовать для выявления медленных запросов, но это грубая процедура, которую лучше применять как своего рода контроль в промежуточных или рабочих системах. Чтобы выявить медленные запросы до того, как они стали серьезной проблемой, необходим более точный инструмент. Как раз таким инструментом является встроенный в MongoDB профилировщик запросов.

Работа с профилировщиком

Для выявления медленных запросов можно воспользоваться встроенным профилировщиком. По умолчанию профилирование отключено, поэтому для начала необходимо его включить. В оболочке MongoDB введите такие команды:

```
use stocks
db.setProfilingLevel(2)
```

Сначала нужно выбрать базу данных, которую вы собираетесь профилировать; режим профилирования всегда задается для конкретной базы. Затем устанавливаем уровень профилирования 2. При этом выдается самая подробная информация – в журнал записываются сведения обо всех операциях чтения и записи. Существуют еще несколько режимов. Чтобы протоколировать только медленные (занимающие более 100 мс) операции, задайте уровень профилирования 1. Чтобы вообще отключить профилирование, задайте уровень 0. А чтобы протоколировать лишь операции, занимающие более определенного числа миллисекунд, укажите это число в качестве второго аргумента:

```
use stocks
db.setProfilingLevel(1, 50)
```

Включив профилировщик, можно приступать к выполнению запросов. Попробуем найти самую высокую цену закрытия в нашем наборе данных:

```
db.values.find({}).sort({close: -1}).limit(1)
```

Результаты профилирования сохраняются в специальной ограниченной коллекции `system.profile`. Напомним, что размер ограниченных коллекций фиксирован, и данные в них записываются циклически, то есть когда размер коллекции достигает максимума, новые документы начинают записываться на место старых. Для коллекции `system.profile` выделяется 128 КБ, поэтому данные профилирования потребляют не слишком много ресурсов.

Опрашивать `system.profile` можно так же, как любую другую ограниченную коллекцию. Например, вот как найти все запросы, выполнявшиеся дольше 150 мс:

```
db.system.profile.find({millis: {$gt: 150}})
```

А поскольку для ограниченных коллекций поддерживается естественный порядок вставки, то можно воспользоваться оператором `$natural`, чтобы вывести последние добавленные результаты:

```
db.system.profile.find().sort({$natural: -1}).limit(5)
```

Результирующий набор для приведенного выше запроса будет выглядеть примерно так:

```
{ "ts" : ISODate("2011-09-22T22:42:38.332Z"),
```

```
"op" : "query", "ns" : "stocks.values",
"query" : { "query" : { }, "orderby" : { "close" : -1 } },
"ntoreturn" : 1, "nscanned" : 4308303, "scanAndOrder" : true,
"nreturned" : 1, "responseLength" : 194, "millis" : 14576,
"client" : "127.0.0.1", "user" : "" }
```

Как видим, запрос обходится весьма дорого: выполнялся почти 15 секунд! Помимо общего времени выполнения, вы получаете ту же информацию, что присутствует в журнале медленных запросов MongoDB, и этого достаточно, чтобы приступить к более углубленному исследованию проблемы, о чем мы поговорим в следующем разделе.

Но предварительно скажем несколько слов о стратегии профилирования. Лучше всего начать с установки грубых параметров, а потом постепенно снижать порог. Сначала удостоверьтесь в отсутствии запросов, выполняющихся дольше 100 мс, потом понизьте планку до 75 мс и т. д. При включенном профилировщике вы должны подвергнуть приложение дотошному испытанию. Как минимум, необходимо, чтобы были выполнены все встречающиеся в нем операции чтения и записи. Но этого мало – нужно, чтобы все эти операции выполнялись в реальных условиях, когда объемы данных, нагрузка и характеристики оборудования подобны тем, что существуют в производственной среде.

Профилировщик – вещь полезная, но чтобы получить от него максимум возможного, нужно действовать методично. Лучше обнаружить медленные запросы на этапе разработки, а не в режиме промышленной эксплуатации, когда исправление обойдется гораздо дороже.

7.3.2. Исследование медленных запросов

Профилировщик MongoDB позволяет легко находить медленные запросы. Но вот для того чтобы понять, *почему* эти запросы медленные, придется провести целое детективное расследование. Как уже было сказано, у этого явления много причин. Если вам повезет, то для ускорения запроса, возможно, придется всего лишь добавить индекс. В более трудных случаях необходимо реорганизовывать индексы, изменять структуру модели данных или модернизировать оборудование. Но сначала всегда нужно искать самое простое решение, этим мы и займемся ниже.

В простейшем случае корень зла – отсутствие индексов, непригодные индексы или плохо составленные запросы. Чтобы узнать точ-

но, можно выполнить подозрительные запросы в режиме *объяснения*. Посмотрим, как это делается.

Команда EXPLAIN()

Команда MongoDB `explain` выводит подробную информацию о пути выполнения запроса[11]. Давайте посмотрим, что можно узнать о последнем запросе, приведенном в предыдущем разделе. Чтобы запустить `explain` из оболочки, достаточно добавить вызов метода `explain()`:

```
db.values.find({}).sort({close: -1}).limit(1).explain()
{
   "cursor" : "BasicCursor",
   "nscanned" : 4308303,
   "nscannedObjects" : 4308303,
   "n" : 1,
   "scanAndOrder" : true,
   "millis" : 14576,
   "nYields" : 0,
   "nChunkSkips" : 0,
   "indexBounds" : { }
}
```

Поле `millis` говорит, что запрос выполнялся более 14 секунд, и понятно почему. Взгляните на значение `nscanned`: оно показывает, что серверу пришлось просмотреть 4 308 303 документа. А теперь выполним команду `count` для коллекции *values*:

```
db.values.count()
4308303
```

Как видим, это общее число документов в коллекции. Таким образом, мы просканировали коллекцию целиком. Если вы ожидали, что запрос должен будет просмотреть каждый документ, то всё нормально. Но в данном-то случае возвращен всего один документ, как показывает значение `n`, так что мы имеем очевидную проблему. Вообще говоря, значения `n` и `nscanned` должны быть максимально близки, чего при полном сканировании коллекции почти никогда не бывает. Поле `cursor` говорит, что использовался курсор типа `BasicCursor`, и это лишь подтверждает, что сканировалась сама коллекция, а не индекс.

Еще одно объяснение причины медленного выполнения мы находим в поле `scanAndOrder`. Этот признак появляется, когда оптими-

[11] В главе 2 я уже кратко упоминал о команде `explain`. Здесь же приводится ее исчерпывающее объяснение.

затор запроса не может воспользоваться индексом для возврата отсортированного результирующего набора. Поэтому в данном случае сервер не только вынужден был просмотреть всю коллекцию, но еще и отсортировать ее заново.

Низкая производительность – вещь неприемлемая, но здесь исправить положение просто. Нужно лишь построить индекс по полю `close`. Сделайте это и выполните запрос снова[12]:

```
db.values.ensureIndex({close: 1})
db.values.find({}).sort({close: -1}).limit(1).explain()
{
   "cursor" : "BtreeCursor close_1 reverse",
   "nscanned" : 1,
   "nscannedObjects" : 1,
   "n" : 1,
   "millis" : 0,
   "nYields" : 0,
   "nChunkSkips" : 0,
   "indexBounds" : {
      "close" : [
         [
            {
               "$maxElement" : 1
            },
            {
               "$minElement" : 1
            }
         ]
      ]
   }
}
```

Почувствуйте разницу! Теперь запрос завершается меньше, чем за одну миллисекунду. Поле `cursor` показывает, что был использован курсор типа `BtreeCursor` над индексом *close_1* и что индекс обходился в обратном порядке. В поле `indexBounds` присутствуют два специальных значения: `$maxElement` и `$minElement`. Они показывают, что запрос покрывает весь индекс. Таким образом, оптимизатор решил, что нужно начать с правого края B-дерева, дойти до максимального ключа, а затем повернуть обратно. Поскольку мы просили вернуть всего одно значение, то запрос завершился сразу после обнаружения максимального элемента. А так как записи в индексе упорядочены, то больше не нужно дополнительно сортировать коллекцию, о чем свидетельствует отсутствие признака `scanAndOrder`.

Если включить в селектор запроса ключ, по которому построен индекс, то распечатка будет несколько иной. Взгляните на объясне-

[12] Отметим, что построение индекса может занять несколько минут.

ние запроса, в котором выбираются цены закрытия, большие 500:

```
> db.values.find({close: {$gt: 500}}).explain()
{
   "cursor" : "BtreeCursor close_1",
   "nscanned" : 309,
   "nscannedObjects" : 309,
   "n" : 309,
   "millis" : 5,
   "nYields" : 0,
   "nChunkSkips" : 0,
   "indexBounds" : {
      "close" : [
         [
            500,
            1.7976931348623157e+308
         ]
      ]
   }
}
```

Мы снова просматриваем ровно столько документов, сколько возвращаем (`n` и `nscanned` одинаковы), – идеальная ситуация. Однако обратите внимание на то, что границы индекса теперь другие. Вместо `$maxElement` и `$minElement` указаны истинные значения. Нижняя граница равна 500, а верхняя – по сути дела, бесконечность. Эти значения всегда принадлежат тому же семейству типов данных, которое указано в запросе. Так как в данном случае запрашивается число, то и границы индекса числовые. Если бы в запросе был указан диапазон строк, то и границы были бы строками[13].

Прежде чем читать дальше, попробуйте выполнить `explain()` для нескольких запросов по собственному усмотрению, обращая внимание на разницу между `n` и `nscanned`.

Оптимизатор запросов в MongoDB и команда HINT()

Оптимизатор запросов – это программный компонент, который решает, какой индекс эффективнее всего использовать для выполнения запроса и есть ли такой индекс вообще. Для выбора идеального индекса оптимизатор руководствуется довольно простыми правилами:

1. Избегать `scanAndOrder`. Если в запросе указана сортировка, попытаться выполнить ее с помощью индекса.

[13] Если это непонятно, вспомните, что индекс может содержать ключи разных типов. Но результат содержит только значения того типа, который указан в запросе.

2. Удовлетворять ограничения на поля с помощью индексов, то есть попытаться использовать индексы для полей, указанных в селекторе запроса.
3. Если в запросе указан диапазон или сортировка, то выбрать индекс, последний ключ которого можно будет использовать для удовлетворения этого условия.

Если существует единственный индекс, отвечающий всем этим пожеланиям, то он и будет использован. Если таких индексов несколько, то оптимальным признается любой из них. Отсюда вывод: если вы можете построить оптимальные индексы для своих запросов, то существенно облегчите работу оптимизатору. Всеми силами стремитесь к этому.

Рассмотрим запрос, для которого есть идеальный индекс. Вернемся к набору данных о биржевых тикерах. Пусть требуется найти для акций Google все цены закрытия, большие 200:

```
db.values.find({stock_symbol: "GOOG", close: {$gt: 200}})
```

Оптимальный для этого запроса индекс должен содержать оба ключа, причем ключ `close` должен быть последним, чтобы ускорить поиск по диапазону:

```
db.values.ensureIndex({stock_symbol: 1, close: 1})
```

Выполнив запрос, вы увидите, что используются оба ключа, и границы индекса соответствуют ожиданиям:

```
db.values.find({stock_symbol: "GOOG", close: {$gt: 200}}).
explain()
{
   "cursor" : "BtreeCursor stock_symbol_1_close_1",
   "nscanned" : 730,
   "nscannedObjects" : 730,
   "n" : 730,
   "millis" : 1,
   "nYields" : 0,
   "nChunkSkips" : 0,
   "isMultiKey" : false,
   "indexOnly" : false,
   "indexBounds" : {
      "stock_symbol" : [
         [
            "GOOG",
            "GOOG"
         ]
      ],
      "close" : [
```

```
            [
               200,
               1.7976931348623157e+308
            ]
         ]
      }
}
>
```

Команда `explain` показывает, что план выполнения оптимален: значения `n` и `nscanned` одинаковы. А теперь рассмотрим случай, когда не существует индекса, идеально отвечающего запросу. Пусть, например, индекса `{stock_symbol: 1, close: 1}` нет, но зато есть два отдельных индекса по каждому полю. Вызвав метод `getIndexKeys()` для получения списка индексов, вы увидите следующее:

```
db.values.getIndexKeys()
[ { "_id" : 1 }, { "close" : 1 }, { "stock_symbol" : 1 } ]
```

Поскольку в запросе указаны оба ключа `stock_symbol` и `close`, то очевидного кандидата на роль оптимального индекса нет. Тут-то и приходит на помощь оптимизатор со своей эвристикой, которая проще, чем можно было бы предположить. Она основана исключительно на значении `nscanned`. Иными словами, оптимизатор выбирает тот индекс, для которого число просмотренных индексных записей будет наименьшим. При первом выполнении запроса оптимизатор создает план выполнения для каждого индекса, который мог бы эффективно удовлетворить этот запрос. Затем оптимизатор прогоняет все планы параллельно[14]. Тот план, для которого получится наименьшее значение `nscanned`, признается победителем. Затем оптимизатор останавливает прогон планов, требующих больше времени, и запоминает победителя для использования в будущем.

Посмотреть, как это выглядит на практике, можно запустив свой запрос в режиме `explain()`. Сначала удалите составной индекс `{stock_symbol: 1, close: 1}` и постройте отдельные индексы по каждому ключу:

```
db.values.dropIndex("stock_symbol_1_close_1")
db.values.ensureIndex({stock_symbol: 1})
db.values.ensureIndex({close: 1})
```

Затем передайте методу `explain()` аргумент `true`, чтобы включить список планов, опробуемых оптимизатором. Результат приведен в листинге 7.1.

---

[14] Строго говоря, выполнение планов чередуется.

Листинг 7.1. Просмотр плана выполнения в режиме explain(true)

```
db.values.find({stock_symbol: "GOOG", close: {$gt: 200}}).explain(true)
{
   "cursor" : "BtreeCursor stock_symbol_1",
   "nscanned" : 894,
   "nscannedObjects" : 894,
   "n" : 730,
   "millis" : 8,
   "nYields" : 0,
   "nChunkSkips" : 0,
   "isMultiKey" : false,
   "indexOnly" : false,
   "indexBounds" : {
      "stock_symbol" : [
         [
            "GOOG",
            "GOOG"
         ]
      ]
   },
   "allPlans" : [
      {
         "cursor" : "BtreeCursor close_1",
         "indexBounds" : {
            "close" : [
               [
                  100,
                  1.7976931348623157e+308
               ]
            ]
         }
      },
      {
         "cursor" : "BtreeCursor stock_symbol_1",
         "indexBounds" : {
            "stock_symbol" : [
               [
                  "GOOG",
                  "GOOG"
               ]
            ]
         }
      },
      {
         "cursor" : "BasicCursor",
         "indexBounds" : {
         }
      }
   ]
}
```

Сразу видно, что оптимизатор выбрал для выполнения запроса индекс `{stock_symbol: 1}`. Ниже приведен ключ `allPlans`, указывающий на список, содержащий два альтернативных плана выполнения: с индексом `{close: 1}` и с полным сканированием коллекции (курсор типа `BasicCursor`).

Понятно, почему оптимизатор отверг сканирование, но почему не годится индекс `{close: 1}`, не столь очевидно. Для ответа на этот вопрос можно воспользоваться методом `hint()`, который заставляет оптимизатор выбрать конкретный индекс:

```
query = {stock_symbol: "GOOG", close: {$gt: 100}}
db.values.find(query).hint({close: 1}).explain()
{
   "cursor" : "BtreeCursor close_1",
   "nscanned" : 5299,
   "n" : 730,
   "millis" : 36,
   "indexBounds" : {
      "close" : [
         [
            200,
            1.7976931348623157e+308
         ]
      ]
   }
}
```

Как видите, в этом случае `nscanned` равно 5299, то есть гораздо больше, чем 894 записи, просматриваемые при использовании предыдущего индекса. И на времени выполнения запроса это отражается весьма заметно.

Осталось только понять, как оптимизатор кэширует выбранный план выполнения и сколько времени план хранится в кэше. В конце концов, не будет же оптимизатор параллельно прогонять и сравнивать планы при каждом запросе.

После того как подходящий план найден, запоминаются образец запроса, значение `nscanned` и спецификация выбранного индекса. Для запроса, с которым мы работали, сохраняемая структура выглядит так:

```
{  pattern: {stock_symbol: 'equality', close: 'bound',
   index: {stock_symbol: 1},
   nscanned: 894 }
```

В образце запроса запоминаются способы сравнения для каждого ключа. В данном случае для ключа `stock_symbol` использовалось сравнение на равенство (equality), а для ключа `close` – сравнение с

диапазоном (bound)[15]. Если новый запрос соответствует образцу, то будет использован данный индекс.

Но план не должен оставаться в кэше навечно. Оптимизатор автоматически выталкивает план из кэша при наступлении любого из следующих событий:

- произведено 100 операций записи в коллекцию;
- изменился состав индексов над коллекцией (построены новые или удалены существующие);
- запрос, выполняемый по кэшированному плану, производит гораздо больше работы, чем ожидалось. Здесь под словами «много работы» понимается, что истинное значение `nscanned` превосходит хранящееся в кэше более чем в 10 раз.

В последнем случае оптимизатор сразу же начинает пробовать другие планы запроса в надежде, что альтернативный индекс окажется более эффективным.

7.3.3. Образцы запросов

В этом разделе я представлю несколько типичных образцов запросов и используемых для них индексов.

Индексы с одним ключом

Для этого примера вспомним индекс над коллекцией цен на акции, построенный по ценам закрытия: `{close: 1}`. Этот индекс можно использовать в следующих случаях.

Точное равенство

Например, для поиска всех документов, в которых цена закрытия равна 100:

```
db.values.find({close: 100})
```

Сортировка

Сортировка по индексированному полю, например:

```
db.values.find({}).sort({close: 1})
```

В случае сортировки без селектора запроса, наверное, следует указать ограничение на число записей, если только вы не собираетесь обойти всю коллекцию.

[15] Если вам интересно, сообщу, что сохраняется один их трех видов диапазона: upper (верхняя граница), lower (нижняя граница) и upper-and-lower (верхняя и нижняя граница). Кроме того, в образце запроса сохраняется спецификация сортировки.

Запросы по диапазону

Это запросы с критерием принадлежности поля некоторому диапазону с сортировкой по этому полю или без нее. Например, найти все цены закрытия, большие или равные 100:

```
db.values.find({close: {$gte: 100})
```

Если еще добавить по ключу сортировку, то оптимизатор тем не менее сможет воспользоваться тем же индексом.

```
db.values.find({close: {$gte: 100}).sort({close: 1})
```

Составные индексы

Составные индексы немного сложнее, но по существу аналогичны индексам с одним ключом. Главное, что нужно запомнить: составной индекс способен эффективно обслуживать только один диапазон или сортировку в одном запросе. Возьмем, к примеру, индекс по тройному ключу всё над тем же набором данных об акциях: `{close: 1, open: 1, date: 1}`. Ниже описано несколько возможных сценариев.

Точное равенство

Точное равенство для первого ключа, для первого и второго или для всех трех – при любой другой комбинации индекс будет бесполезен:

```
db.values.find({close: 1})
db.values.find({close: 1, open: 1})
db.values.find({close: 1, open: 1, date: "1985-01-08"})
```

Попадание в диапазон

Точное равенство с нулем или более «левых» ключей и попадание следующего по порядку ключа в указанный диапазон или сортировка по этому ключу. Так, все показанные ниже запросы могут быть удовлетворены вышеупомянутым индексом по тройному ключу:

```
db.values.find({}).sort({close: 1})
db.values.find({close: {$gt: 1}})
db.values.find({close: 100}).sort({open: 1})
db.values.find({close: 100, open: {$gt: 1}})
db.values.find({close: 1, open: 1.01, date: {$gt: "2005-01-01"}})
db.values.find({close: 1, open: 1.01}).sort({date: 1})
```

Покрывающие индексы

Если вы никогда не слышали о *покрывающих индексах*, то имейте в виду, что этот термин не вполне удачен. Покрывающий индекс не является каким-то особым видом индекса, а лишь предполагает

определенное использование индекса. В частности, можно сказать, что индекс покрывает запрос, если все необходимые запросу данные содержатся в самом индексе. Запрос, удовлетворяемый покрывающим индексом, иногда так и называют – *«запрос к индексу»* (index-only query), поскольку для его выполнения обращаться к самим индексированным документам вообще не нужно. Это может повысить производительность.

В MongoDB использовать покрывающий индекс просто. Нужно лишь указать в запросе подмножество полей, содержащихся в индексе, и исключить поле `_id` (так как оно, скорее всего, не является частью индекса). Вот пример использования построенного ранее составного индекса по трем ключам:

```
db.values.find({open: 1}, {open: 1, close: 1, date: 1, _id: 0})
```

Если выполнить для этого запроса команду `explain()`, то результат будет включать поле `indexOnly`, равное `true`. Это означает, что для обслуживания запроса был использован только индекс, без обращения к самой коллекции данных.

Оптимизация запросов неизбежно зависит от приложения, но хочется надеяться, что изложенные выше идеи и приемы помогут вам настроить собственные запросы для получения максимальной производительности. Эмпирические подходы всегда полезны. Возьмите за правило профилировать и анализировать запросы с помощью `explain`. Заодно заглянете в потайные закоулки оптимизатора и научитесь формулировать запросы наиболее эффективно.

7.4. Резюме

Эта глава оказалась объемной, так как индексирование – весьма обширная тема. Если вы не всё поняли, ничего страшного. По крайней мере, вы теперь кое-что знаете о том, как анализировать индексы и избегать медленных запросов, и имеете базу для того, чтобы продолжить изучение. Ввиду сложности, присущей индексированию и оптимизации запросов, самым лучшим учителем будет старый добрый эксперимент.

ГЛАВА 8.
Репликация

В этой главе:

- Основные идеи репликации.
- Администрирование набора реплик и отработка отказов.
- Подключение к набору реплик, гарантии записи, масштабирование чтения и тегирование.

Репликация – одна из центральных тем в большинстве СУБД по одной простой причине: отказы неизбежны. Если вы хотите, чтобы данные производственной системы были доступны даже после отказа, то должны размещать их на нескольких машинах. Репликация страхует от сбоев, обеспечивая высокую доступность и аварийное восстановление.

Я начну эту главу с общего введения в репликацию и обсуждения нескольких наиболее типичных сценариев. Затем я перейду к детальному изучению наборов реплик в MongoDB. И в конце опишу, как подключаться к реплицированным кластерам MongoDB с помощью драйверов, как использовать гарантии записи и как балансировать операции чтения между несколькими репликами.

8.1. Обзор репликации

Под репликацией понимается размещение и обслуживание серверов базы данных на нескольких машинах. В MongoDB реализованы два варианта репликации: *главный-подчиненный* и *наборы реплик*. В обоих случаях единственный первичный узел выполняет все операции записи, после чего вторичные узлы считывают описания этих операций и асинхронно применяют их у себя.

В обоих вариантах репликации применяется один и тот же механизм, но наборы реплик дополнительно обеспечивают автоматическую отработку отказа: если первичный узел по какой-то причине выходит из строя, то один из вторичных узлов автоматически принимает на себя его функции, если это возможно. Наборы реплик обладают и другими достоинствами, например, упрощенное восстановление и возможность более сложных топологий развертывания. Поэтому не так уж много причин использовать простую репликацию типа главный-подчиненный[1]. Таким образом, рекомендуемая стратегия репликации в производственных системах – наборы реплик, и именно их я буду в основном объяснять и иллюстрировать на примерах в этой главе, а о репликации типа главный-подчиненный скажу лишь несколько общих слов.

8.1.1. Почему так важна репликация

Любая база данных уязвима перед лицом отказов инфраструктуры, в которой работает. Репликация в какой-то мере является страховкой от таких отказов. О каких отказах может идти речь? Вот несколько возможных сценариев.

- Пропадает сетевое соединение между приложением и базой данных.
- Вследствие запланированного отключения сервер временно выводится из оперативного доступа. Любая организация, эксплуатирующая серверы, периодически вынуждена останавливать их для профилактического обслуживания, и не всегда получается предсказать это событие заранее. Простая перезагрузка компьютера делает базу данных недоступной на несколько минут. Но что произойдет по завершении перезагрузки? Бывает, что после установки нового оборудования или программного обеспечения операционная система перестает загружаться.
- Отключения питания тоже возможны. Хотя большинство современных центров обработки данных оснащены резервными источниками питания, нет никакой гарантии от ошибок пользователей внутри самого ЦОД или от длительного сбоя системы электроснабжения. А в результате сервер базы данных останавливается.

[1] Единственный случай, когда следует выбирать репликацию главный-подчиненный, – это наличие более 11 подчиненных узлов, потому что в наборе реплик MongoDB может быть не больше 12 членов.

- Сбой диска на сервере базы данных. Обычно среднее время наработки на отказ составляет всего несколько лет, так что диски выходят из строя чаще, чем вы думаете[2].

В MongoDB репликация не только защищает от внешних сбоев, но и чрезвычайно важна для обеспечения долговечности данных. Если сервер запущен без журналирования, то не гарантируется, что файлы данных MongoDB останутся неповрежденными после нештатного останова. Если журнал выключен, то обязательно следует реплицировать данные, чтобы иметь исправную копию в случае некорректного останова одного узла.

Разумеется, репликация желательна и тогда, когда журналирование включено. Ведь вам же нужна высокая доступность и быстрая отработка отказа, правда? Журналирование ускоряет восстановление, так как для возврата отказавших узлов в нормальный режим работы достаточно воспроизвести журнал. Это куда быстрее, чем синхронизация с существующей репликой или копирование файлов данных из реплики вручную.

В общем, с журналированием или без, репликация в MongoDB значительно повышает общую надежность базы данных и потому настоятельно рекомендуется.

8.1.2. Сценарии репликации

Вы удивитесь, какую гибкость обеспечивает репликация базы данных. Среди прочего, она позволяет организовать резервирование, отработку отказа, обслуживание и балансирование нагрузки. Ниже мы вкратце рассмотрим все эти сценарии.

Основное назначение репликации – обеспечить резервирование. По существу это означает, что должна поддерживаться синхронизация между первичным узлом и его репликами. Реплики могут находиться в том же ЦОД, что и первичный узел, или быть территориально разнесенными для пущей отказоустойчивости. Поскольку репликация производится асинхронно, то сетевые задержки и удаленность вторичных узлов никак не сказываются на производительности первичного узла. Тот факт, что реплицированные узлы могут отставать от первичного на заданное количество секунд, можно считать еще одной формой резервирования. Это страховка на тот случай, когда пользо-

[2] Детальный анализ отказов стандартных жестких дисков приведен в опубликованной Google статье «Failure Trends in a Large Disk Drive Population» (http://research.google.com/archive/disk_failures.pdf).

ватель случайно удалил коллекцию или приложение каким-то образом повредило базу данных. Обычно такие операции реплицируются немедленно, но задержка репликации дает администратору возможность вовремя отреагировать и, быть может, спасти данные.

Важно отметить, что хотя реплики и обеспечивают резервирование, они не могут служить заменой резервному копированию. Резервная копия – это снимок базы данных на определенный момент в прошлом, тогда как реплика всегда актуальна. Бывают ситуации, когда набор данных настолько велик, что резервное копирование трудноосуществимо, но в общем случае снимать резервные копии – благоразумное решение, рекомендуемое даже при включенной репликации.

Еще один сценарий применения репликации – отработка отказов. Вы, безусловно, хотите, чтобы система была доступна большую часть времени, но добиться этого можно только при наличии резервных узлов и средств переключения на них в случае аварии. К счастью, наборы реплик в MongoDB зачастую обеспечивают автоматическое переключение.

Помимо резервирования и отработки отказов, репликация упрощает обслуживание, обычно за счет того, что длительные операции выполняются не на первичном узле. Например, общепринятой практикой является снятие резервной копии на вторичном узле, чтобы разгрузить первичный и избежать простоя. Другой пример – построение больших индексов. Так как это дорогостоящая операция, то часто имеет смысл выполнить ее сначала на вторичном узле, затем поменять первичный и вторичный узел ролями, после чего построить его еще раз уже на новом вторичном узле.

Наконец, репликация позволяет балансировать операции чтения между репликами. В тех случаях, когда приложения преимущественно читают данные, это самый простой способ масштабировать MongoDB. Но несмотря на все обещания, масштабирование чтения за счет применения вторичных узлов ничего не даст, если выполняется любое из следующих условий:

- Имеющееся оборудование не справляется с нагрузкой. В предыдущей главе я уже приводил соответствующий пример. Если размер рабочего набора данных намного превышает объем оперативной памяти, то случайное распределение операций чтения между вторичными узлами не избавит от интенсивного доступа к диску и, следовательно, замедлению обработки запросов.

- Отношение числа операций записи к числу операций чтения превышает 50 %. Эта величина выбрана более-менее произвольно, но в качестве отправной точки вполне разумна. Проблема в том, что каждая операция записи на первичном узле должна быть рано или поздно повторена на всех вторичных. Поэтому перенаправление операций чтения на вторичные узлы, которые уже заняты обработкой записи, иногда замедляет репликацию и может не дать повышения пропускной способности чтения.
- Приложению требуется согласованное чтение. Репликация на вторичные узлы производится асинхронно, поэтому нет гарантии, что будут отражены результаты последних операций записи. В патологических случаях вторичные узлы могут отставать от первичного на несколько часов.

Таким образом, репликация позволяет балансировать чтение, но только в частных случаях. Если наблюдается одно из вышеперечисленных условий, то для масштабирования нужно применять другую стратегию, например сегментирование, покупку дополнительного оборудования или сочетание того и другого.

8.2. Наборы реплик

Наборы реплик – это усовершенствованная технология репликации типа главный-подчиненный, рекомендуемая для MongoDB. Начнем с конфигурирования простого набора реплик. Затем я опишу, как в действительности работает репликация, поскольку эти знания неоценимы для диагностики проблем в реальной системе. И в конце мы обсудим более сложные аспекты конфигурации, отработку отказов, восстановление и оптимальные способы развертывания.

8.2.1. Настройка

Минимальная рекомендуемая конфигурация набора реплик включает три узла. Два из них – полноценные экземпляры `mongod`. Каждый может выступать в роли первичного узла набора реплик, и на обоих могут храниться полные копии данных. Третий узел играет роль *арбитра*; он не реплицирует данные, а выполняет функции нейтрального наблюдателя. Как следует из названия, этот узел занимается арбитражем: в случае отказа арбитр помогает выбрать новый первичный узел. Пример набор реплик изображен на рис. 8.1.

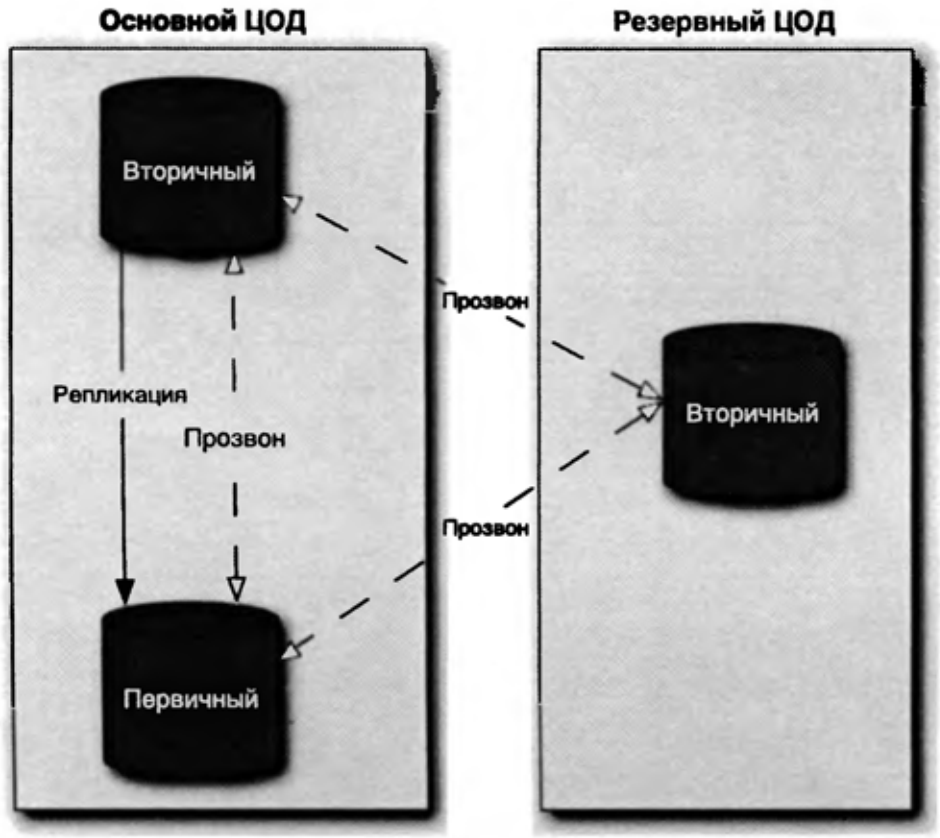


Рис. 8.1. Простой набор реплик, состоящий из первичного узла, вторичного узла и арбитра

Начнем с создания каталога данных для каждого члена набора реплик:

```
mkdir /data/node1
mkdir /data/node2
mkdir /data/arbiter
```

Затем запустим для каждого члена набора отдельный экземпляр mongod. Так как все процессы работают на одной машине, то лучше всего запускать разные mongod в разных консольных окнах:

```
mongod --replSet myapp --dbpath /data/node1 --port 40000
mongod --replSet myapp --dbpath /data/node2 --port 40001
mongod --replSet myapp --dbpath /data/arbiter --port 40002
```

Заглянув в журнал mongod, вы сразу же увидите сообщения о том, что не найдена конфигурация. Это нормально:

```
[startReplSets] replSet can't get local.system.replset
config from self or any seed (EMPTYCONFIG)
[startReplSets] replSet info you may need to run replSetInitiate
```

Далее необходимо сконфигурировать набор реплик. Для этого сначала подключитесь к одному из запущенных экземпляров mongod,

но не к арбитру. В нашем примере все три процесса mongod запущены локально, поэтому подключаться нужно по локальному имени, в данном случае – arete.

После подключения выполните команду rs.initiate():

```
> rs.initiate()
{
   "info2" : "no configuration explicitly specified -- making one",
   "me" : "arete:40000",
   "info" : "Config now saved locally. Should come online in about a minute
      .",
   "ok" : 1
}
```

Примерно через минуту вы получите набор реплик с одним членом. Чтобы добавить два других, нужно выполнить команду rs.add():

```
> rs.add("localhost:40001")
{ "ok" : 1 }
> rs.add("arete.local:40002", {arbiterOnly: true})
{ "ok" : 1 }
```

Для второго узла мы задали параметр arbiterOnly, то есть сконфигурировали его для работы в качестве арбитра. Через минуткудругую все члены должны перейти в рабочий режим. Чтобы получить краткие сведения о состоянии набора реплик, выполните команду db.isMaster():

```
> db.isMaster()
{
   "setName" : "myapp",
   "ismaster" : false,
   "secondary" : true,
   "hosts" : [
      "arete:40001",
      "arete:40000"
   ],
   "arbiters" : [
      "arete:40002"
   ],
   "primary" : "arete:40000",
   "maxBsonObjectSize" : 16777216,
   "ok" : 1
}
```

Более подробную информацию о системе дает метод rs.status(). Он выводит сведения о каждом узле. Ниже приведена полная распечатка состояния:

```
> rs.status()
{
```

```
    "set" : "myall",
    "date" : ISODate("2011-09-27T22:09:04Z"),
    "myState" : 1,
    "members" : [
        {
            "_id" : 0,
            "name" : "arete:40000",
            "health" : 1,
            "state" : 1,
            "stateStr" : "PRIMARY",
            "optime" : {
                "t" : 1317161329000,
                "i" : 1
            },
            "optimeDate" : ISODate("2011-09-27T22:08:49Z"),
            "self" : true
        },
        {
            "_id" : 1,
            "name" : "arete:40001",
            "health" : 1,
            "state" : 2,
            "stateStr" : "SECONDARY",
            "uptime" : 59,
            "optime" : {
                "t" : 1317161329000,
                "i" : 1
            },
            "optimeDate" : ISODate("2011-09-27T22:08:49Z"),
            "lastHeartbeat" : ISODate("2011-09-27T22:09:03Z"),
            "pingMs" : 0
        },
        {
            "_id" : 2,
            "name" : "arete:40002",
            "health" : 1,
            "state" : 7,
            "stateStr" : "ARBITER",
            "uptime" : 5,
            "optime" : {
                "t" : 0,
                "i" : 0
            },
            "optimeDate" : ISODate("1970-01-01T00:00:00Z"),
            "lastHeartbeat" : ISODate("2011-09-27T22:09:03Z"),
            "pingMs" : 0
        }
    ],
    "ok" : 1
}
```

Если ваша база данных MongoDB не слишком велика, то набор реплик должен перейти в рабочий режим в течение 30 секунд. На протяжении этого времени поле `stateStr` в каждом узле будет сначала равно `RECOVERING`, а потом – `PRIMARY`, `SECONDARY` или `ARBITER`.

А теперь хотелось бы не слепо доверять показанному состоянию, а эмпирически убедиться, что репликация действительно работает. Так давайте подключимся к первичному узлу из оболочки и вставим какой-нибудь документ:

```
$ mongo arete:40000
> use bookstore
switched to db bookstore
> db.books.insert({title: "Oliver Twist"})
> show dbs
admin (empty)
bookstore 0.203125GB
local 0.203125GB
```

Данные должны реплицироваться практически мгновенно. В другом консольном окне откройте еще один экземпляр оболочки и подключитесь в нем к вторичному узлу. Запросите только что вставленный документ; он уже должен появиться:

```
$ mongo arete:40001
> show dbs
admin (empty)
bookstore 0.203125GB
local 0.203125GB
> use bookstore switched to db bookstore
> db.books.find()
{ "_id" : ObjectId("4d42ebf28e3c0c32c06bdf20"), "title" : "Oliver Twist" }
```

Если все произошло, как показано, значит, набор реплик успешно сконфигурирован.

Конечно, смотреть на репликацию в действии приятно, но интереснее, как поведет себя автоматическая отработка отказа. Проверим. Имитировать отказ сети проблематично, поэтому пойдем по более простому пути – «прибьем» узел. Можно остановить вторичный узел, но тогда мы всего лишь прекратим репликацию, а остальные узлы сохранят текущее состояние. Если же вы хотите наблюдать, как изменяется состояние системы, то нужно остановить первичный узел. Воспользуйтесь стандартной комбинацией клавиш Ctrl+C или командой `kill -2`. Можно также подключиться к первичному узлу и выполнить в оболочке команду `db.shutdownServer()`.

После того как первичный узел остановлен, вторичный обнаруживает отсутствие периодических тактовых сигналов от него. Тогда

вторичный узел выбирает себя на роль первичного. Такой выбор возможен, потому что большинство исходных узлов (арбитр и исходный вторичный) могут прозванивать друг друга. Вот выдержка из журнала вторичного узла:

```
[ReplSetHealthPollTask] replSet info arete:40000 is down (or slow to respond)
Mon Jan 31 22:56:22 [rs Manager] replSet info electSelf 1
Mon Jan 31 22:56:22 [rs Manager] replSet PRIMARY
```

Если сейчас подключиться к новому первичному узлу и проверить состояние набора реплик, то будет видно, что старый первичный узел недоступен:

```
> rs.status()
{
   "_id" : 0,
   "name" : "arete:40000",
   "health" : 1,
   "state" : 6,
   "stateStr" : "(not reachable/healthy)",
   "uptime" : 0,
   "optime" : {
      "t" : 1296510078000,
      "i" : 1
   },
   "optimeDate" : ISODate("2011-01-31T21:43:18Z"),
   "lastHeartbeat" : ISODate("2011-02-01T03:29:30Z"),
   "errmsg": "socket exception"
}
```

После отработки отказа в наборе реплик осталось всего два узла. Поскольку на арбитре никакие данные не хранятся, приложение продолжит работать при условии, что обращается только к первичному узлу[3]. Однако репликация не происходит, поэтому больше возможности для автоматической отработки отказа нет. Необходимо восстановить старый первичный узел. В предположении, что он был остановлен штатно, его можно вновь запустить, и он автоматически присоединится к набору реплик в качестве вторичного узла. Попробуйте проделать это.

Это упрощенное описание набора реплик. Конечно, есть и неприглядные детали, куда же без них? В следующих двух разделах мы увидим, как на самом деле работают наборы реплик, и рассмотрим развертывание, более сложные конфигурации и способы разрешения

[3] Иногда приложение опрашивает и вторичные узлы, чтобы масштабировать операции чтения. В таком случае описанный выше отказ приведет к ошибкам чтения. Поэтому важно проектировать приложение с учетом возможных отказов. Я еще скажу об этом в конце главы.

запутанных ситуаций, которые иногда возникают в производственных системах.

8.2.2. Как работает репликация

Набор реплик основан на двух механизмах: *журнал операций* (oplog) и *тактовый сигнал* (heartbeat). Журнал операций делает репликацию возможной, а с помощью тактовых сигналов ведется мониторинг состояния и активируется процедура отработки отказа. Ниже мы рассмотрим принципы работы обоих механизмов. Это научит вас понимать и прогнозировать поведение набора реплик, особенно в аварийных ситуациях.

Всё о журнале операций

Журнал операций – это душа репликации в MongoDB. Он представляет собой ограниченную коллекцию, находящуюся в базе данных `local` на каждом узле; в эту коллекцию записываются все изменения данных. Всякий раз, как клиент что-то записывает на первичный узел, в журнал операций на этом узле автоматически заносится информация, достаточная для воспроизведения операции записи. После репликации на вторичный узел эта информация окажется в его журнале операций. Каждая запись в журнале операций снабжена временной меткой в формате BSON, и вторичные узлы используют эту метку, чтобы понять, какая запись была применена последней[4].

Чтобы лучше понять, как всё это работает, посмотрим на реальный журнал операций и хранящуюся в нем информацию. Подключитесь в оболочке к первичному узлу, запущенному в предыдущем разделе, и сделайте текущей базу данных `local`:

```
> use local
switched to db local
```

В базе данных `local` хранятся метаданные, описывающие набор реплик, и журнал операций. Естественно, сама эта база не реплицируется, то есть вполне отвечает своему названию – локальная база данных, своя на каждом узле. В ней вы найдете коллекцию `oplog.rs`, в которой и хранится журнал операций. Кроме нее, есть еще несколько системных коллекций. Вот их полный перечень:

```
> show collections
me
```

[4] Временная метка в формате BSON – это уникальный идентификатор, состоящий из количества секунд с начала «эпохи» и монотонно растущего счетчика.

```
oplog.rs
replset.minvalid
slaves
system.indexes
system.replset
```

В `replset.minvalid` хранится информация о начальной синхронизации данного члена набора реплик, а в `system.replset` – документ с описанием конфигурации набора реплик. Коллекции `me` и `slaves` нужны для реализации механизма гарантирования записи, описанного в конце главы, а `system.indexes` – стандартный контейнер для спецификаций индексов.

Прежде всего, сосредоточим внимание на журнале операций. Запросим из него запись, соответствующую документу `book`, который вы добавили в предыдущем разделе. Для этого введите показанный ниже запрос. Результирующий документ состоит из четырех полей:

```
> db.oplog.rs.findOne({op: "i"})
{  "ts" : { "t" : 1296864947000, "i" : 1 }, "op" : "i", "ns" :
   "bookstores.books", "o" : { "_id" : ObjectId("4d4c96b1ec5855af3675d7a1"),
   "title" : "Oliver Twist" }
}
```

В первом поле, `ts`, хранится временная метка записи в формате BSON. Обратите внимание – оболочка отображает временную метку в виде поддокумента с двумя полями: `t`, число секунд с начала «эпохи», и `i`, счетчик. На основании этого вы могли бы прийти к выводу, что эту запись можно запросить таким образом:

```
db.oplog.rs.findOne({ts: {t: 1296864947000, i: 1}})
```

Но на самом деле этот запрос вернет `null`. Чтобы запросить временную метку, необходимо явно сконструировать соответствующий объект. В любом драйвере имеются специальные конструкторы для создания временных меток в формате BSON, и JavaScript – не исключение. Вот как это делается:

```
db.oplog.rs.findOne({ts: new Timestamp(1296864947000, 1)})
```

Второе поле в записи журнала операций, `op`, определяет код операции. Из него вторичный узел понимает, какую операцию представляет эта запись. В данном случае значение `i` обозначает вставку. После `op` идет поле `ns`, описывающее пространство имен (базу данных и коллекцию) и `o`, которое в случае вставки содержит копию вставленного документа.

При изучении журнала операций можно заметить, что операции, затрагивающие несколько документов, разбиваются на составные час-

ти. Так, в случае множественного обновления и массового удаления для каждого документа в журнале создается отдельная запись. Пусть, например, в коллекцию добавлено еще несколько книг Диккенса:

```
> use bookstore
db.books.insert({title: "A Tale of Two Cities"})
db.books.insert({title: "Great Expectations"})
```

Имея четыре книги, мы можем выполнить команду множественного обновления, которая заменяет имя автора:

```
db.books.update({}, {$set: {author: "Dickens"}}, false, true)
```

А что при этом появится в журнале операций?

```
> use local
> db.oplog.$main.find({op: "u"})
{  "ts" : { "t" : 1296944149000, "i" : 1 }, "op" : "u",
   "ns" : "bookstore.books",
   "o2" : { "_id" : ObjectId("4d4dcb89ec5855af365d4283") },
   "o" : { "$set" : { "author" : "Dickens" } } }

{  "ts" : { "t" : 1296944149000, "i" : 2 }, "op" : "u",
   "ns" : "bookstore.books",
   "o2" : { "_id" : ObjectId("4d4dcb8eec5855af365d4284") },
   "o" : { "$set" : { "author" : "Dickens" } } }

{  "ts" : { "t" : 1296944149000, "i" : 3 }, "op" : "u",
   "ns" : "bookstore.books",
   "o2" : { "_id" : ObjectId("4d4dcbb6ec5855af365d4285") },
   "o" : { "$set" : { "author" : "Dickens" } } }
```

Как видите, каждому обновленному документу соответствует своя запись. Такая нормализация является частью общей стратегии, цель которой – гарантировать, что на вторичных узлах окажутся те же данные, что на первичном. Для этого каждая операция должна быть идемпотентной – результат не должен зависеть от того, сколько раз она применена. Прочие многодокументные операции, в частности удаление, демонстрируют такое же поведение. Попробуйте выполнить несколько разных операций и посмотрите, как они отражаются в журнале.

Чтобы получить основную информацию о текущем состоянии журнала операций, выполните из оболочки метод `db.getReplicationInfo()`:

```
> db.getReplicationInfo()
{
   "logSizeMB" : 50074.10546875,
   "usedMB" : 302.123,
```

```
    "timeDiff" : 294,
    "timeDiffHours" : 0.08,
    "tFirst" : "Thu Jun 16 2011 21:21:55 GMT-0400 (EDT)",
    "tLast" : "Thu Jun 16 2011 21:26:49 GMT-0400 (EDT)",
    "now" : "Thu Jun 16 2011 21:27:28 GMT-0400 (EDT)"
}
```

Здесь мы видим временные метки первой и последней записи в журнале. Их можно найти вручную, указав модификатор сортировки `$natural`. Например, запрос `db.oplog.rs.find().sort({$natural: -1}).limit(1)` выбирает самую позднюю запись.

Осталось разобраться еще в одном вопросе: как вторичные узлы отслеживают, до какого места в журнале операций они уже дошли. Ответ простой – вторичные узлы также ведут журнал операций. Это существенное преимущество по сравнению с репликацией типа главный-подчиненный, поэтому подробнее рассмотрим, какие соображения за ним стоят.

Представьте, что вы инициируете операцию записи на первичном узле набора реплик. Что произойдет? Сначала эта операция производится и добавляется в журнал операций на первичном узле. Тем временем все вторичные узлы ведут собственные журналы операций, реплицирующие журнал первичного узла. Когда вторичный узел будет готов обновить себя, он делает три вещи. Во-первых, смотрит на временную метку последней записи в своем журнале. Во-вторых, запрашивает у журнала первичного узла все записи с более поздними метками. И, наконец, добавляет эти записи в свой журнал и применяет их к своей базе данных[5]. Это означает, что в случае отработки отказа у вторичного узла, ставшего первичным, будет журнал операций, который смогут реплицировать другие вторичные узлы. Именно этот механизм и обеспечивает восстановление набора реплик.

Вторичные узлы применяют технику «протяженного опроса» (long polling), чтобы незамедлительно получать новые записи из журнала операций первичного узла. Поэтому, как правило, вторичные узлы почти не отстают от первичного. Если же отставание всё же случается, например из-за топологии сети или обслуживания самого вторичного узла, то для оценки его величины можно воспользоваться временной меткой последней записи в журнале операций вторичного узла.

---

[5] Если включен режим журналирования, то документы записываются в основные файлы данных и в журнал операций одновременно, в рамках атомарной транзакции.

Остановленная репликация

Репликация полностью останавливается, если вторичный узел не может понять, до какого момента он синхронизирован с журналом операций первичного узла. Если такое происходит, то в журнале вторичного узла появляется такое сообщение об ошибке:

```
repl: replication data too stale, halting
Fri Jan 28 14:19:27 [replsecondary] caught SyncException
```

Напомним, что журнал операций – это ограниченная коллекция. Следовательно, записи в ней рано или поздно затираются новыми. Если вторичный узел не может найти в журнале операций первичного узла точку, до которой он синхронизировался, то невозможно гарантировать, что данные на вторичном узле являются точной копией данных на первичном. Единственный способ исправить эту ситуацию – провести полную ресинхронизацию с данными первичного узла, поэтому следует всеми силами ее избегать. Для этого требуется следить за отставанием вторичного узла и выбирать размер журнала операций, соответствующий нагрузке на вашу систему. Подробные сведения о мониторинге приведены в главе 10. А сейчас мы рассмотрим вопрос о выборе размера журнала операций.

Выбор размера журнала операций для репликации

Журнал операций – ограниченная коллекция, поэтому после создания изменить его размер невозможно (по крайней мере, в версии MongoDB 2.0)[6]. Поэтому так важно тщательно подходить к вопросу о выборе размера.

Размер, подразумеваемый по умолчанию, зависит от нескольких факторов. В 32-разрядных системах он равен 50 МБ, а в 64-разрядных – 5 % свободного дискового пространства, но не менее 1 ГБ[7]. Во многих системах 5 % свободного места – более чем достаточно. Можно сказать, что если бы журнал такого размера был полностью перезаписан 20 раз, то диск оказался бы заполненным.

С другой стороны, размер по умолчанию идеален не для всех приложений. Если вы заранее знаете, что число операций записи будет велико, то перед развертыванием следует провести эмпирическое тес-

[6] В будущем планируется ввести возможность увеличения размера ограниченной коллекции. См. https://jira.mongodb.org/browse/SERVER-1864.

[7] В системе OS X размер журнала операций составляет 192 МБ. Это решение принято потому, что предполагается, что Mac'и применяются для разработки, а не для эксплуатации.

тирование. Настройте репликацию и начните писать на первичный узел в том темпе, который ожидается в производственной системе. «Бомбить» сервер следует не меньше часа. По завершении испытания подключитесь к любому члену набору реплик и получите информацию о текущем состоянии репликации:

```
db.getReplicationInfo()
```

Зная, сколько места в журнале операций заполняется за час, вы сможете оценить потребный размер журнала. Пожалуй, имеет смысл заложиться на восьмичасовой простой вторичного узла. Ваша цель – не доводить до полной ресинхронизации любого узла, а увеличение журнала операций даст вам время на восстановление в случае сетевых сбоев и других неожиданностей.

Чтобы изменить подразумеваемый размер журнала операций, нужно при первом запуске каждого узла-члена командой `mongod` задать параметр `--oplogSize`. Значение измеряется в мегабайтах. Следовательно, для запуска `mongod` с журналом операций размером 1 ГБ необходимо выполнить команду:

```
mongod --replSet myapp --oplogSize 1024
```

Тактовые сигналы и отработка отказа

Механизм тактовых сигналов обеспечивает отработку отказа и выбор нового первичного узла. По умолчанию каждый член набора реплик посылает тактовые сигналы всем остальным членам (прозванивает их) раз в две секунды. Это позволяет системе в целом отслеживать собственное состояние. Команда `rs.status()` показывает временные метки последнего тактового сигнала от каждого узла и его состояние в поле `health` (`1` означает, что узел работает, `0` – что он не отвечает).

Пока все узлы отвечают, набор реплик радостно трудится. Но если какой-то узел перестает отвечать, необходимо принять меры. Набор реплик стремится обеспечить наличие ровно одного первичного узла в любой момент времени. Однако это возможно лишь в случае, когда большинство узлов видимы. Вспомните о наборе реплик, созданном в предыдущем разделе. После остановки вторичного узла еще остается большинство узлов, поэтому набор реплик не изменяет свое состояние, а просто ждет, пока вторичный узел вернется. Если остановить первичный узел, то большинство по-прежнему существует, но уже без первичного узла. Поэтому на роль первичного автоматически назначается вторичный узел. Если вторичных узлов несколько, то выбирается узел с самым «свежим» состоянием.

Но возможны и другие сценарии. Представьте, что остановлены одновременно вторичный узел и арбитр. Первичный узел остался, но большинства уже нет – из трех первоначальных узлов работает только один. В таком случае в журнале первичного узла появится сообщение:

```
Tue Feb 1 11:26:38 [rs Manager] replSet can't see a majority of the set,
relinquishing primary
Tue Feb 1 11:26:38 [rs Manager] replSet relinquishing primary state
Tue Feb 1 11:26:38 [rs Manager] replSet SECONDARY
```

В отсутствие большинства первичный узел добровольно принимает роль вторичного. На первый взгляд, странно, но подумайте, что могло бы случиться, если бы этому узлу было разрешено оставаться первичным. Возможно, что из-за каких-то сетевых проблем перестают приходить тактовые сигналы, но остальные узлы при этом работают. Если арбитр и вторичный узел видят друг друга, то согласно правилу большинства, оставшийся вторичный узел должен стать первичным. Но если исходный первичный узел не понизил себя в ранге, то мы оказываемся в недопустимой ситуации: набор реплик содержит два первичных узла. Если при этом приложение продолжает работать, то будет записывать и читать данные с двух разных первичных узлов – верный путь к несогласованности и странному поведению. Поэтому, не видя большинства, первичный узел обязан сложить с себя полномочия.

Фиксация и откат

И последнее, что нужно знать о наборах репликации, – это понятие *фиксации*. Вы можете писать на первичный узел весь день напролет, но операции записи не будут считаться зафиксированными, пока они не реплицированы на большинство узлов. Что здесь понимается под словом *зафиксированы*? Проще всего объяснить на примере. Пусть имеется набор реплик, сконфигурированный в предыдущем разделе. Предположим, мы отправили первичному узлу серию операций записи, которые по какой-то причине не реплицированы на вторичный узел (из-за нарушения связности сети, останова вторичного узла для снятия резервной копии, отставания вторичного узла и так далее). Предположим также, что вторичный узел неожиданно оказался выбран на роль первичного. Теперь вы пишете на новый первичный узел, но старый первичный рано или поздно «оживет» и попытается реплицировать данные с нового первичного узла. Проблема в том, что на старом первичном узле имеется ряд операций записи, которых нет

в журнале операций нового первичного узла. В такой ситуации инициируется откат.

При откате все операции записи, которые не были реплицированы на большинство узлов, отменяются, то есть удаляются из журнала операций вторичного узла и из коллекции, которой адресованы. Если вторичный узел произвел операцию удаления, то он найдет удаленный документ в другой реплике и восстановит его. То же самое относится к удалению целых коллекций и обновлению документов.

Отмененные операции записи хранятся в подкаталоге `rollback` каталога данных на соответствующем узле. Для каждой коллекции, в которой имеются откаченные операции записи, создается отдельный BSON-файл, имя которого включает время отката. Если потребуется восстановить откаченные документы, то можно исследовать эти файлы с помощью утилиты `bsondump` и произвести восстановление вручную, например воспользовавшись `mongorestore`.

Если у вас когда-нибудь возникнет необходимость в восстановлении откаченных данных, то вы поймете, что этой ситуации следует всячески избегать, и, к счастью, это до некоторой степени возможно. Если приложение может смириться с небольшой задержкой записи, то можно воспользоваться описанной ниже техникой гарантирования записи, чтобы обеспечить репликацию данных на большинство узлов при каждой записи (или, быть может, после нескольких последовательных операций записи). Умелое применение гарантий записи и мониторинга задержки репликации поможет вам справиться или даже вообще не допустить проблем с откатом.

В этом разделе вы, наверное, узнали о внутренних механизмах репликации больше, чем ожидали, но эти знания небесполезны. Понимание того, как работает репликация, очень поможет при диагностике многих ошибок в производственных системах.

8.2.3. Администрирование

Хотя многие возможности наборов реплик автоматизированы, у них есть ряд нетривиальных конфигурационных параметров. Далее я подробно рассмотрю их. Чтобы вы не запутались, я также скажу, какие параметры можно без опаски игнорировать.

Детали конфигурирования

В этом разделе описаны параметры запуска `mongod`, относящиеся к наборам реплик, а также структура документа, содержащего конфигурацию набора реплик.

Параметры репликации

Выше вы научились инициализировать набор реплик с помощью методов оболочки `rs.initiate()` и `rs.add()`. Эти методы удобны, но скрывают некоторые конфигурационные параметры набора реплик. Здесь мы увидим, как инициализировать и изменять конфигурацию набора реплик с помощью конфигурационного документа.

В конфигурационном документе задается конфигурация набора реплик. Чтобы создать этот документ, сначала добавьте в ключ `_id` значение, совпадающее с именем, которое было задано в параметре `--replSet`:

```
> config = {_id: "myapp", members: []}
{ "_id" : "myapp", "members" : [ ] }
```

Члены набора реплик (`members`) можно определить в конфигурационном документе следующим образом:

```
config.members.push({_id: 0, host: 'arete:40000'})
config.members.push({_id: 1, host: 'arete:40001'})
config.members.push({_id: 2, host: 'arete:40002', arbiterOnly:
true})
```

Сейчас конфигурационный документ должен выглядеть так:

```
> config
{
   "_id" : "myapp",
   "members" : [
      {
         "_id" : 0,
         "host" : "arete:40000"
      },
      {
         "_id" : 1,
         "host" : "arete:40001"
      },
      {
         "_id" : 2,
         "host" : "arete:40002",
         "arbiterOnly" : true
      }
   ]
}
```

Затем можно передать этот документ в качестве первого аргумента методу `rs.initiate()`, чтобы инициализировать набор реплик.

Технически документ состоит из поля `_id`, содержащего имя набора реплик, массива описаний от 3 до 12 членов (`members`) и необя-

зательного поддокумента, определяющего некоторые глобальные настройки. В приведенном выше примере заданы минимально необходимые конфигурационные параметры, а также необязательная настройка `arbiterOnly`.

Необходимо, чтобы поле `_id` в документе совпадало с именем набора реплик. Команда инициализации проверяет, что при запуске каждого узла-члена был указан параметр `--replSet` именно с таким значением. В элементе массива `members`, соответствующего каждому члену набора реплик, поле `_id` должно содержать последовательные целые числа, начиная с 0. Также в каждом элементе должно быть поле `host`, содержащее имя сервера и необязательный номер порта.

Набор реплик инициализируется с помощью метода `rs.initiate()`. Это простая обертка вокруг команды `replSetInitiate`. Следовательно, можно было бы запустить набор реплик так:

```
db.runCommand({replSetInitiate: config});
```

Здесь `config` – переменная, в которой находится конфигурационный документ. После инициализации на каждом узле-члене будет храниться копия этого конфигурационного документа – в коллекции `system.replset` в базе данных `local`. Опросив эту коллекцию, вы обнаружите, что у документа имеется номер версии. При последующих изменениях конфигурации набора реплик этот номер необходимо увеличивать на 1.

Для изменения конфигурации набора реплик имеется специальная команда `replSetReconfig`, которая принимает новый конфигурационный документ. В новом документе можно задать добавление или удаление членов набора, а также модификации локальных и глобальных параметров. Процедура модификации конфигурационного документа, увеличения номера версии и передачи его команде `replSetReconfig` утомительна, поэтому в оболочке предусмотрено несколько вспомогательных методов для упрощения работы. Их перечень можно получить, вызвав в оболочке `rs.help()`. С одним таким вспомогательным методом, `rs.add()`, вы уже знакомы.

Имейте в виду, что если изменение конфигурации набора реплик влечет за собой выбор нового первичного узла, то все соединения с клиентами закрываются. Это делается для того, чтобы клиенты не пытались отправить вторичному узлу команды записи в режиме «выстрелил и забыл».

Если вам интересно, как набор реплик конфигурируется на уровне драйвера, можете заглянуть в реализацию метода `rs.add()`. Вве-

дите строку `rs.add` (имя метода без скобок) в ответ на приглашение оболочки, и вы узнаете, как этот метод работает.

Параметры в конфигурационном документе

До сих пор мы ограничивались простейшим конфигурационным документом набора реплик. Но вообще-то в них можно задавать и другие параметры как на уровне отдельных членов набора, так и на уровне набора реплик в целом. Начнем с параметров членов. С параметрами `_id`, `host` и `arbiterOnly` вы уже знакомы. Ниже они и все остальные параметры описаны во всех подробностях.

- `_id` (*обязательный*) – уникальное монотонно возрастающее целое число, представляющее идентификатор члена набора. Значения начинаются с 0 и должны увеличиваться на 1 при добавлении каждого нового члена.
- `host` (*обязательный*) – строка, содержащая имя сервера данного узла и необязательный номер порта. Если порт указан, то должен отделяться от имени сервера двоеточием (например, `arete:30000`). Если порт не указан, то по умолчанию подразумевается 27017.
- `arbiterOnly` – булевское значение, `true` или `false`, показывающее, является ли данный член арбитром. На узлах-арбитрах хранятся только конфигурационные данные. Они принимают участие в выборе первичного узла, но не в самой репликации.
- `priority` – целое число от 0 до 1000, определяющее вероятность того, что этот узел будет выбран на роль первичного. На этапе инициализации и отработки отказа набор реплик пытается выбрать на роль первичного узел с наивысшим приоритетом при условии, что данные на нем актуальны.
  В некоторых случаях требуется, чтобы узел никогда не становился первичным (например, если это узел аварийного восстановления, находящийся во вторичном ЦОД). Тогда задайте для него приоритет 0. Узлы с нулевым приоритетом помечаются как пассивные в результатах команды `isMaster()` и никогда не выбираются на роль первичного узла.
- `votes` – по умолчанию у каждого члена набора реплик имеется всего один голос. Параметр `votes` позволяет увеличить число голосов у данного члена.
  Этим параметром следует пользоваться с большой осторожностью. Прежде всего, трудно предсказать, как поведет себя

набор реплик при отработке отказа, если у его членов разное число голосов. К тому же, в подавляющем большинстве производственных систем одного голоса на каждый член вполне достаточно. Поэтому если вы собираетесь изменять число голосов, то тщательно обдумайте свое решение и постарайтесь промоделировать различные сценарии отказа.

- `hidden` – булевское значение. Если равно `true`, то этот член не будет показываться в результатах команды `isMaster`. Поскольку драйверы MongoDB используют эту команду для получения информации о топологии набора реплик, то сокрытие члена не позволяет драйверам автоматически обращаться к нему. Этот параметр может использоваться в сочетании с параметром `buildIndexes` и обязан использоваться в сочетании с `slaveDelay`.
- `buildIndexes` – булевское значение, по умолчанию равно `true`. Определяет, будет ли этот член строить индексы. Задавать значение `false` следует только для членов, которые никогда не станут первичными узлами (то есть с приоритетом 0).

  Параметр предназначен для узлов, используемых исключительно для резервного копирования. Если требуется включать в резервную копию индексы, то не задавайте этот параметр.
- `slaveDelay` – на сколько секунд этот вторичный узел должен отставать от первичного. Параметр можно задавать только для узлов, которые никогда не станут первичными. Поэтому, задавая значение `slaveDelay`, большее 0, не забудьте установить приоритет в 0.

  Отставание репликации может служить страховкой от некоторых видов ошибок пользователей. Например, если для вторичного узла задано отставание на 30 минут и администратор случайно удалил базу данных, то у него будет полчаса на исправление последствий до того, как они распространятся по всей системе.
- `tags` – документ, содержащий произвольное множество пар ключ-значение. Обычно используется для идентификации положения члена в конкретном ЦОД или серверной стойке. Теги применяются для задания уровня детальности гарантий записи и параметров чтения. Подробно они обсуждаются в разделе 8.4.9.

Это все параметры отдельных членов набора реплик. Существуют также два глобальных конфигурационных параметра, располагаемых в разделе `settings`. В конфигурационном документе они выглядят следующим образом:

```
{
   settings: {
      getLastErrorDefaults: {w: 1},
      getLastErrorModes: {
         multiDC: { dc: 2 }
      }
   }
}
```

- `getLastErrorDefaults` – документ, описывающий, какие аргументы следует подставлять по умолчанию, когда клиент вызывает команду `getLastError` без аргументов. Этот параметр следует применять с осторожностью, так как имеется также возможность задавать глобальные умолчания для `getLastError` на уровне драйвера, и может сложиться ситуация, когда разработчик приложения вызывает `getLastError`, не сознавая, что администратор определил умолчание на сервере.
  Дополнительные сведения о команде `getLastError` см в разделе 3.2.3, посвященном гарантиям записи. В двух словах – чтобы описать режим, в котором все операции записи реплицируются по крайней мере на два члена с таймаутом 500 мс, нужно задать этот параметр следующим образом: `settings: { getLastErrorDefaults: {w: 2, wtimeout: 500} }`.
- `getLastErrorModes` – документ, в котором описываются дополнительные режимы для команды `getLastError`. Эта функция зависит от тегирования набора реплик и подробно описывается в разделе 8.4.4.

Состояние набора реплик

Посмотреть состояние набора реплик и его членов позволяет команда `replSetGetStatus`. Для ее вызова из оболочки воспользуйтесь вспомогательным методом `rs.status()`. В возвращенном документе отражены работающие члены и их состояния, время с момента запуска и временные метки журналов операций. Полный список возможных состояний членов приведен в табл. 8.1.

Набор реплик можно считать стабильным и работоспособным, если все узлы находятся в состояниях 1, 2 или 7, и ровно один узел

является первичным. Команду `replSetGetStatus` можно вызывать из внешнего скрипта для мониторинга общего состояния, отставания репликации и времени с момента запуска; именно такой подход рекомендуется применять в производственных системах[8].

Таблица 8.1. Состояния наборов реплик

Состояние	В виде строки	Примечания
0	`STARTUP`	Набор реплик инициализируется – ведется прозвон членов и обмен конфигурационной информацией.
1	`PRIMARY`	Это первичный узел. Набор реплик всегда должен содержать *не более* одного первичного узла.
2	`SECONDARY`	Это вторичный узел, находящийся в режиме чтения. Он может стать первичным в результате отработки отказа тогда и только тогда, когда его приоритет больше 0 и он не помечен как скрытый.
3	`RECOVERING`	Этот узел недоступен ни для чтения, ни для записи. Обычно это состояние наблюдается после отработки отказа или добавления нового узла. Во время восстановления часто производится синхронизация файлов данных; проверить это можно, заглянув в журналы на восстанавливаемом узле.
4	`FATAL`	Сетевое соединение есть, но узел не отвечает на прозвон. Обычно это означает фатальную ошибку на машине, где работает узел.
5	`STARTUP2`	Идет начальная синхронизация файлов данных.
6	`UNKNOWN`	Сетевое соединение еще не установлено.
7	`ARBITER`	Этот узел является арбитром.
8	`DOWN`	В какой-то момент узел был доступен и работал стабильно, но сейчас не отвечает на прозвон тактовыми сигналами.
9	`ROLLBACK`	Производится откат.

Отработка отказа и восстановление

При обсуждении тестового набора реплик мы уже приводили два примера отработки отказа. Здесь я подведу итоги и дам некоторые рекомендации по восстановлению.

[8] Отметим, что, помимо запроса информации о состоянии с помощью команды, можно получить полезную визуальную картину на веб-консоли. Веб-консоль обсуждается в главе 10, где приведен и пример ее использования для набора реплик.

Набор реплик считается работоспособным, когда все члены, описанные в конфигурации, могут обмениваться между собой данными. По умолчанию у каждого узла имеется один голос, который учитывается при выборе первичного узла. Это означает, что минимальное количество узлов (и голосов) в наборе реплик равно 2. Но начальное число голосов также определяет, что будет считаться большинством в случае отказа.

Предположим, что сконфигурирован набор из трех реплик (без арбитра) и, следовательно, рекомендации по минимальному числу членов для автоматической отработки отказа выдержаны. Если первичный узел выходит из строя, а оставшиеся вторичные видят друг друга, то можно выбрать новый первичный узел. На его роль будет выбран вторичный узел с самым свежим журналом операций (или с самым высоким приоритетом).

Виды отказов и восстановление

Восстановление – это процесс приведения набора реплик в исходное состояние после отказа. Следует рассмотреть две основных категории отказов. Первая – это так называемые *чистые отказы*, когда есть основания предполагать, что файлы данных в узле не повреждены. Примером может служить потеря связности сети. Если узел теряет соединение с другими узлами набора, то нужно только подождать, пока связность восстановится, после чего временно выпавший из сети узел присоединится к набору. Аналогичная ситуация возникает, когда процесс `mongod` по какой-то причине завершается, но может быть заново запущен без повреждения данных[9]. И в этом случае после перезапуска процесса узел сам присоединится к набору реплик.

К отказам второго типа относятся все *безоговорочные отказы* (categorical failure), когда либо файлы данных вообще утрачены, либо должны считаться поврежденными. Нештатный останов процесс `mongod`, работающего без журналирования, и сбой жесткого диска – это примеры отказов такого рода. Восстановить безоговорочно отказавший узел можно только путем замены файлов данных – посредством повторной синхронизации или извлечения из недавно снятой резервной копии. Рассмотрим обе стратегии по очереди.

Чтобы выполнить полную ресинхронизацию, запустите `mongod` на отказавшем узле с пустым каталогом данных. При условии, что имя сервера и номер порта не изменились, новый экземпляр `mongod`

[9] Например, если сервер MongoDB был остановлен штатно, то файлы данных в порядке. Или, если экземпляр MongoDB работает в режиме журналирования, то его можно восстановить даже после нештатного выключения.

присоединится к набору реплик и синхронизируется с существующими данными. Если имя или порт изменились, то после запуска mongod нужно будет переконфигурировать набор реплик. Для примера предположим, что узел arete:40001 не подлежит восстановлению и вместо него поднят новый узел по адресу foobar:40000. Чтобы переконфигурировать набор реплик, модифицируйте адрес сервера для второго узла и передайте измененный конфигурационный документ методу `rs.reconfig()`:

```
> use local
> config = db.system.replset.findOne()
{
   "_id" : "myapp",
   "version" : 1,
   "members" : [
      {
         "_id" : 0,
         "host" : "arete:30000"
      },
      {
         "_id" : 1,
         "host" : "arete:30001"
      },
      {
         "_id" : 2,
         "host" : "arete:30002"
      }
   ]
}
> config.members[1].host = "foobar:40000"
arete:40000
> rs.reconfig(config)
```

Теперь набор реплик опознает новый узел, и тот начнет синхронизацию с существующим членом.

Вместо полной ресинхронизации можно восстановить данные из недавней резервной копии. Обычно резервное копирование производится на одном из вторичных узлов путем получения снимка файла данных и сохранения его на неподключенном к системе носителе[10]. Восстановление из резервной копии возможно только в случае, когда хранящийся в копии журнал операций не устарел относительно журналов операций на текущих членах набора реплик. Это означает, что последняя операция в сохраненном журнале должна еще присутствовать в активных журналах. Проверить, так ли это, позволит информация, возвращаемая методом `db.getReplicationInfo()`. Получив эту

[10] Резервное копирование подробно обсуждается в главе 10.

информацию, не забудьте принять во внимание время, необходимое для восстановления. Если есть шанс, что последняя запись в журнале из резервной копии может устареть за время переноса этой копии на новую машину, то лучше не рисковать и прибегнуть к полной ресинхронизации.

Но восстановление из резервной копии может оказаться быстрее, отчасти потому, что не придется заново создавать индексы. Для восстановления скопируйте файлы данных из резервной копии в каталог данных `mongod`. Ресинхронизация должна начаться автоматически, и убедиться в этом можно, заглянув в журналы или вызвав метод `rs.status()`.

Стратегии развертывания

Вы уже знаете, что набор реплик может включать не более 12 узлов. Кроме того, вы познакомились с ошеломляющим разнообразием конфигурационных параметров и со способами отработки отказов и восстановления. Настроить набор реплик можно многими способами, но в этом разделе я покажу всего два, применимых в большинстве случаев.

Минимальный набор реплик, допускающий автоматическую отработку отказов, состоит из двух реплик и одного арбитра; именно такой набор мы и рассматривали выше. В производственной системе арбитр может работать на сервере приложений, а каждая реплика – на отдельном компьютере. Эта конфигурация, с одной стороны, экономична, а, с другой, достаточна для большинства практических приложений.

Но в тех случаях, когда доступность критически важна, потребуется набор, состоящий их *трех* полных реплик. Что дает дополнительная реплика? Представьте себе, что один узел окончательно вышел из строя. У вас тем не менее остается два полноценных узла, которые будут работать, пока третий восстанавливается. На протяжении этого времени (возможно, нескольких часов) набор реплик все еще способен к автоматической отработке отказа и содержит актуальные данные.

Для некоторых приложений необходимый уровень резервирования обеспечивают два центра обработки данных, и в этом случае набор из трех реплик тоже годится. Хитрость в том, чтобы использовать один ЦОД только для аварийного восстановления. На рис. 8.2 изображен пример подобной конфигурации. Здесь в основном ЦОДе размещен первичный и вторичный узел набора реплик, а в резервном – еще один вторичный узел, работающий в пассивном режиме (с приоритетом 0).

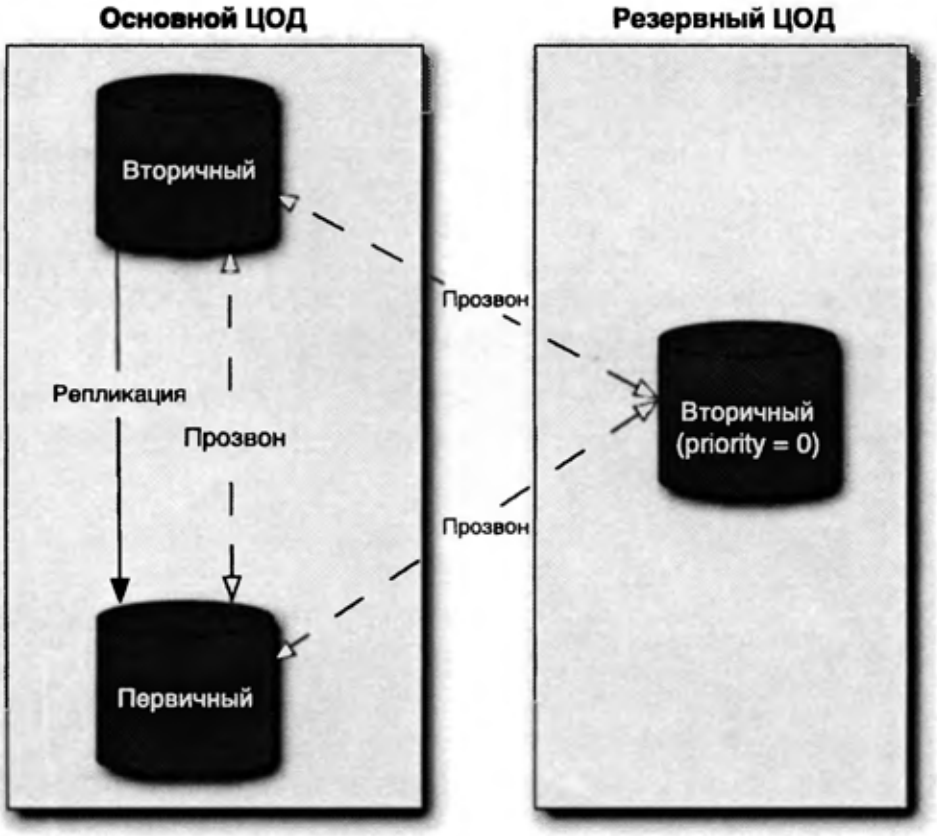


Рис. 8.2. Набор реплик с тремя узлами-членами, расположенными в двух центрах обработки данных

В такой конфигурации в качестве первичного узла набора реплик всегда будет выступать один из двух узлов в ЦОД *A*. Даже при выходе из строя одного любого узла или одного ЦОД приложение по-прежнему будет работать. Отработка отказа обычно происходит автоматически, за исключением случая, когда вышли из строя оба узла в центре *A*. Поскольку одновременный отказ двух узлов – событие редкое, то это, скорее всего, означает полный выход центра *A* из строя или потерю связи с ним. Чтобы быстро восстановиться, можно остановить член набора в центре *B* и перезапустить его без флага `--repl-Set`. Или запустить два новых узла в центре *B*, а затем произвести переконфигурирование набора реплик. Вообще говоря, не рекомендуется переконфигурировать набор, когда большинство узлов недоступно, но в крайнем случае это допускается – с помощью параметра `force`. Например, если определить новый конфигурационный документ `config`, то принудительно выполнить переконфигурирование можно так:

```
> rs.reconfig(config, {force: true})
```

Как всегда в производственной системе, тестирование должно

стоять на первом месте. Обязательно проверьте все типичные сценарии отработки отказов и восстановления на тестовой системе, сравнимой по характеристикам с производственной. Зная по опыту, как поведет себя набор реплик в тех или иных ситуациях, вы будете спать спокойнее и сохраните присутствие духа для устранения аварии.

8.3. Репликация типа главный-подчиненный

Репликация типа главный-подчиненный была исходной моделью репликации в MongoDB. Она легко конфигурируется и способна поддержать любое количество подчиненных узлов. Но теперь этот тип репликации не рекомендуется использовать в производственных системах. Тому есть две причины. Во-первых, отработка отказа должна производиться целиком вручную. Если главный узел выходит из строя, то администратор должен остановить подчиненный узел и перезапустить его в режиме главного. Затем необходимо переконфигурировать приложение, так чтобы оно работало с новым главным узлом. Во-вторых, восстановление затруднено. Поскольку журнал операций существует только на главном узле, в случае отказа приходится создавать новый журнал операций на новом главном узле. Это означает, что все прочие узлы должны будут заново синхронизироваться с новым главным.

Короче говоря, убедительных причин использовать репликацию типа главный-подчиненный не существует. Набор реплик – вот магистральный путь развития, ему и следует отдать предпочтение.

8.4. Драйверы и репликация

Если вы разрабатываете приложение, зависящее от репликации MongoDB, то должны знать о трех прикладных аспектах. Первый касается подключений и отработки отказов. Второй связан с гарантиями записи, то есть механизмом, позволяющим определить, до какой степени операция записи должна быть реплицирована прежде, чем приложение может продолжать работу. И последний аспект – масштабирование чтения – позволяет приложению распределять операции чтения между репликами. Рассмотрим эти вопросы поочередно.

8.4.1. Подключение и отработка отказов

Драйверы MongoDB предлагают более-менее единообразный интерфейс для подключения к наборам реплик.

Подключение к одному узлу

Вы всегда можете подключиться к одному узлу из набора реплик. Между подключением к первичному узлу набора реплик и обычному автономному узлу нет никакой разницы. В обоих случаях драйвер создает сокет соединения и выполняет команду `isMaster`. Эта команда возвращает документ такого вида:

```
{ "ismaster" : true, "maxBsonObjectSize" : 16777216, "ok" : 1 }
```

С точки зрения драйвера, важно то, что поле `isMaster` равно `true`; это означает, что данный узел либо автономный, либо главный узел в репликации типа главный- подчиненный, либо первичный узел набора реплик[11]. В любом из этих случаев на узел можно писать, а пользователю драйвера разрешено выполнять любую CRUD-операцию.

Но при прямом подключении к вторичному узлу набора реплик вы должны явно указать, что *знаете* о природе этого узла (по крайней мере, в большинстве драйверов). В драйвере для Ruby это достигается заданием параметра `:slave_ok`. Следовательно, чтобы подключиться напрямую к первому вторичному узлу, созданному в этой главе ранее, на Ruby нужно написать такой код:

```
@con = Mongo::Connection.new('arete', 40001, :slave_ok => true)
```

Без аргумента `:slave_ok` драйвер возбудит исключение с сообщением о невозможности подключиться к узлу. Эта проверка производится для того, чтобы предотвратить непреднамеренную запись на вторичный узел. Хотя сервер в любом случае отвергнет такую попытку, вы не увидите никаких исключений, если только не включен безопасный режим.

Предполагается, что обычно вы хотите подключиться к первичному узлу; параметр `:slave_ok` служит дополнительной страховкой.

Подключение к набору реплик

Хотя есть возможность подключаться к отдельному члену набора реплик, обычно подключаются к набору в целом. Это позволяет

[11] Команда `isMaster` возвращает также максимальный размер BSON-объекта для версии MongoDB на указанном сервере. Зная эту величину, драйвер может перед вставкой BSON-объекта проверить, что тот не превышает ограничения по размеру.

драйверу узнать, какой узел является первичным и в случае отказа повторно подключиться к новому первичному узлу.

Большинство официально поддерживаемых драйверов располагают средствами для подключения к набору реплик. В Ruby для этого нужно создать объект `ReplSetConnection`, передав его список начальных узлов:

```
Mongo::ReplSetConnection.new(['arete', 40000], ['arete', 40001])
```

Драйвер сам попытается подключиться к каждому начальному узлу и вызвать команду `isMaster`. Эта команда вернет важную информацию о наборе реплик:

```
> db.isMaster()
{
   "setName" : "myapp",
   "ismaster" : true,
   "secondary" : false,
   "hosts" : [
      "arete:40000",
      "arete:40001"
   ],
   "arbiters" : [
      "arete:40002"
   ],
   "maxBsonObjectSize" : 16777216,
   "ok" : 1
}
```

Получив эти сведения от начального узла, драйвер знает всё, что ему нужно. Теперь он может подключиться к первичному узлу, еще раз проверить, что тот по-прежнему является первичным, а затем позволить пользователю выполнять на этом узле операции чтения и записи. Полученный в ответ объект также дает драйверу возможность кэшировать адреса вторичных узлов и арбитра. Если операция на первичном узле завершится ошибкой, то при последующих запросах драйвер может попытаться подключиться к другим узлам на время, пока не восстановится соединение с первичным.

Важно иметь в виду, что хотя отработка отказов в наборе реплик производится автоматически, драйвер не пытается скрыть факт отказа. Последовательность событий выглядит примерно так. Сначала первичный узел выходит из строя или по какой-то причине назначаются выборы. При выполнении последующего запроса выясняется, что соединение с сокетом разорвано, драйвер возбуждает соответствующее исключение и закрывает все открытые соединения с базой данных. Что делать дальше, решает разработчик приложения, прини-

мая во внимание как выполняемую операцию, так и специфические потребности приложения.

Помня о том, что драйвер автоматически попытается заново подключиться при последующем запросе, представим себе два сценария. Сначала предположим, что вы только читаете из базы. В таком случае никакого вреда в повторной попытке выполнить неудачную операцию чтения не будет, так как состояние базы данных при этом не изменится. Но допустим, что вы также регулярно пишете в базу. Мы уже много раз говорили, что запись можно производить в безопасном и небезопасном режиме. В безопасном режиме драйвер после каждой операции записи выполняет команду `getlasterror`. Тем самым он проверяет, что операция благополучно дошла до сервера и в случае ошибки сообщает о ней приложению. В небезопасном режиме драйвер просто записывает данные в TCP-сокет.

Если в приложении безопасный режим выключен, то после отработки отказа оно окажется в неопределенном состоянии. Сколько недавно выполненных операций записи дошло до сервера? Сколько застряло в буфере сокета? Недетерминированность записи в TCP-сокет не позволяет точно ответить на эти вопросы. Насколько эта проблема серьезна, зависит от приложения. Если речь идет о протоколировании, то запись в небезопасном режиме, наверное, приемлема, так как потеря нескольких операций записи вряд ли изменит общую картину; но если приложение используется для ввода данных, то такая ошибка может привести к катастрофе.

Если безопасный режим включен, то сомнения вызывает только самая последняя операция записи; возможно, она дошла до сервера, а, возможно, и нет. Иногда имеет смысл повторить запись, а иногда сообщить об ошибке приложения. Драйвер-то в любом случае возбудит исключение, но как его обрабатывать, решает разработчик. Так или иначе, повторное выполнение операции заставит драйвер еще раз попытаться установить соединение с набором реплик. Но поскольку драйверы в части подключения к набору реплик ведут себя по-разному, следует обязательно проконсультироваться с документацией.

8.4.2. Гарантии записи

Сейчас вам уже должно быть ясно, что в большинстве приложений разумно работать в безопасном режиме, поскольку важно знать, что операция записи дошла до первичного сервера без ошибок. Но иногда

требуется еще более высокий уровень надежности, и тогда вступает в дело механизм *гарантий записи* (write concern), который позволяет разработчику указать, в какой мере операция записи должна быть реплицирована, чтобы приложение могло продолжать работу. Технически этот механизм контролируется двумя параметрами команды `getlasterror`: `w` и `wtimeout`. Параметр `w` может принимать одно из нескольких значений, но обычно задает общее число серверов, на которые необходимо реплицировать последнюю операцию записи. Параметр `wtimeout` задает величину таймаута в миллисекундах; если в течение этого времени не удалось реплицировать операцию, то команда вернет ошибку.

Например, если вы хотите быть уверены, что операция записи реплицирована по крайней мере на один сервер, то можете задать значение `w`, равное 2. Если требуется, чтобы репликация завершилась в течение 500 мс, то следует еще включить параметр `wtimeout`, равный 500. Кстати, если не задать `wtimeout`, то в случае, когда репликация по какой-то причине вообще не произойдет, программа будет заблокирована на неопределенно долгое время.

При работе с драйвером механизм гарантирования записи активируется не явным вызовом `getLastError`, а созданием объекта гарантии записи или путем задания соответствующего параметра безопасного режима; детали зависят от API конкретного драйвера[12]. В Ruby гарантия записи для отдельной операции задается так:

```
@collection.insert(doc, :safe => {:w => 2, :wtimeout => 200})
```

Иногда нужно лишь гарантировать, что операция записи реплицирована на большинство имеющихся узлов. Тогда в качестве значения `w` нужно указать `majority`:

```
@collection.insert(doc, :safe => {:w => "majority"})
```

Есть и еще более причудливые параметры. Например, если включен режим журналирования, то можно потребовать, чтобы запись в журнал на диске производилась синхронно; для этого предназначен параметр `j`:

```
@collection.insert(doc, :safe => {:w => 2, :j => true})
```

Многие драйверы поддерживают также задание гарантий записи на уровне соединения или базы данных. О том, как задавать гарантии записи для конкретного драйвера, читайте в прилагаемой к нему

[12] В приложении D приведены примеры задания гарантий записи на языках Java, PHP и C++.

документации. Несколько дополнительных примеров приведено в приложении D.

Механизм гарантирования записи работает как с наборами реплик, так и с репликацией типа главный-подчиненный. В базе данных `local` имеются две коллекции: `me` на вторичных узлах и `slaves` – на первичном, которые используются при реализации гарантий записи. Всякий раз, как вторичный узел опрашивает первичный, тот сохраняет информацию о последней записи в журнале операций, примененной на этом вторичном узле, в своей коллекции `slaves`. Таким образом, первичный узел в любой момент знает, что уже реплицировал каждый вторичный узел, и потому может дать надежный ответ на запрос со стороны команды `getlasterror`.

Имейте в виду, что при затребовании гарантий записи с параметром `w`, большим 1, возникает дополнительная задержка. Возможность конфигурирования гарантий записи позволяет *вам* выбирать компромисс между скоростью и долговечностью данных. При работе в режиме журналирования гарантии записи с `w`, равным 1, достаточно для большинства приложений. С другой стороны, для приложений, связанных с протоколированием или аналитикой, можно вообще отключить журналирование и гарантию записи и полагаться только на репликацию, допуская, что в случая сбоя несколько операций записи может быть потеряно. Тщательно обдумывайте такие компромиссы и тестируйте различные сценарии при проектировании приложения.

8.4.3. Масштабирование чтения

Реплицированные базы данных прекрасно подходят для масштабирования чтения. Если один сервер не справляется с количеством запросов на чтение, предъявляемых приложением, то эти запросы можно распределить по нескольким репликам. В большинство драйверов встроена поддержка для отправки запросов вторичным узлам. В драйвере для Ruby это делается с помощью параметра конструктора класса `ReplSetConnection`:

```
Mongo::ReplSetConnection.new(['arete', 40000],
                ['arete', 40001], :read => :secondary )
```

Если аргумент `:read` принимает значение `:secondary`, то объект соединения выберет случайный, расположенный поблизости вторичный узел и отправит ему запрос на чтение.

Другие драйверы можно сконфигурировать для чтения с вторичных узлов путем задания параметра `slaveOk`. Если драйвер для Java

подключен к набору реплик, то при задании `slaveOk = true` балансирование нагрузки между вторичными серверами производится на уровне потоков. Реализация балансирования нагрузки в драйверах проектируется в расчете на общий случай, но не исключено, что именно для вашего приложения она не подходит. В таком случае пользователи часто изобретают собственные механизмы. Как обычно, об особенностях реализации читайте в документации.

Многие пользователи MongoDB прибегают к репликации для масштабирования промышленной системы. Но есть три случая, когда такого вида масштабирования недостаточно. Первый касается количества потребных серверов. В версии MongoDB 2.0 набор реплик может состоять максимум из 12 членов, из которых 7 могут голосовать. Если для масштабирования необходимо больше реплик, то, конечно, *можно* прибегнуть к репликации типа главный-подчиненный. Но если вы не желаете приносить в жертву автоматическую отработку отказов и все-таки хотите выйти за пределы ограничения в 12 узлов-членов, то придется перейти на сегментированный кластер.

Второй случай относится к приложениям с высокой нагрузкой по записи. В начале этой главы уже отмечалось, что вторичные узлы должны успевать за этой нагрузкой. Если отправлять сильно загруженным узлам еще и запросы на чтение, то можно затормозить репликацию.

Третий случай связан с тем, что при масштабировании за счет репликации невозможно обеспечить согласованность чтения. Так как репликация по природе своей асинхронна, то в реплике не всегда отражается состояние первичного сервера, возникшее в результате последней по времени операции записи. Поэтому если приложение читает данные с произвольно выбираемых вторичных узлов, то нет гарантии, что пользователь увидит согласованную картину. В приложениях, основная задача которых – отображать некоторое содержимое, это почти никогда не является проблемой. Но есть другие приложения, пользователи которых активно манипулируют данными, и тогда согласованное чтение важно. В таких случаях у вас есть два варианта действий. Первый – отделить часть приложения, которой нужно согласованное чтение, от той, которая может без него обойтись. Первая часть всегда будет читать с первичного узла, а вторая может распределять операции чтения по вторичным узлам. Если такая стратегия слишком сложна или попросту не масштабируется, то следует идти по пути сегментирования[13].

[13] Отметим, что для обеспечения согласованного чтения с сегментированного кластера, необходимо всегда читать с первичных узлов каждого сегмента, а операции записи выполнять в безопасном режиме.

8.4.4. Тегирование

При использовании гарантий записи и масштабирования чтения может возникнуть желание более точно управлять тем, на какой именно вторичный узел направляется данная операция чтения или записи. Допустим, к примеру, что на двух ЦОД, *NY* и *FR*, развернут набор реплик из пяти узлов. В основном ЦОД, NY, находится три узла, а в резервном, FR, – остальные два. Предположим, что механизм гарантирования записи должен блокировать приложение, пока некоторая операция записи не будет реплицирована хотя бы на один узел в центре FR. На данный момент не видно, каким образом это можно сделать. Задать для `w` значение `majority` нельзя, потому что оно транслируется в число 3, и, скорее всего, репликацию сначала подтвердят три узла, находящиеся в центре NY. Можно было бы задать значение 4, но тогда возникнет проблема, если в каждом центре выйдет из строя по одному узлу.

Эту трудность помогает преодолеть механизм тегирования набора реплик, который позволяет определить специальные режимы гарантирования записи, снабдив членов набора реплик определенными метками (тегами). Чтобы понять, как он работает, нужно сначала научиться помечать члены набора реплик. В конфигурационном документе для каждого члена можно задать ключ `tags`, указывающий на объект, который содержит пары ключ-значение, например:

```
{
   "_id" : "myapp",
   "version" : 1,
   "members" : [
      {
         "_id" : 0,
         "host" : "ny1.myapp.com:30000",
         "tags": { "dc": "NY", "rackNY": "A" }
      },
      {
         "_id" : 1,
         "host" : "ny2.myapp.com:30000",
         "tags": { "dc": "NY", "rackNY": "A" }
      },
      {
         "_id" : 2,
         "host" : "ny3.myapp.com:30000",
         "tags": { "dc": "NY", "rackNY": "B" }
      },
      {
         "_id" : 3,
         "host" : "fr1.myapp.com:30000",
```

```
            "tags": { "dc": "FR", "rackFR": "A" }
        },
        {
            "_id" : 4,
            "host" : "fr2.myapp.com:30000",
            "tags": { "dc": "FR", "rackFR": "B" }
        }
    ],
    settings: {
        getLastErrorModes: {
            multiDC: { dc: 2 } },
            multiRack: { rackNY: 2 } },
        }
    }
}
```

Это тегированный конфигурационный документ для гипотетического набора реплик, распространяющегося на два центра обработки данных. Отметим, что в каждом вложенном документе-теге есть две пары ключ-значение: первая определяет ЦОД, а вторая содержит имя серверной стойки для данного узла. Имена совершенно произвольны и приобретают смысл только в контексте приложения. Поместить в документ-тег можно всё, что угодно; важно лишь то, как мы будем им пользоваться.

Именно для этого и предназначен ключ `getLastErrorModes`. Он позволяет определить режимы для команды `getLastError`, которая реализует конкретные требования к гарантиям записи. В нашем примере задано два режима. Первый, `multiDC`, определен как `{ "dc": 2 }`; это означает, что операция записи должна реплицироваться на узлы, помеченные как минимум двумя разными значениями `dc`. Взглянув на сами теги, вы увидите, что это требование гарантирует распространение данных на оба ЦОД. Второй режим говорит, что операцию записи должны получить по меньшей мере две серверные стойки в центре NY. Что означает это требование, опять же понятно из определения тегов.

В общем случае раздел `getLastErrorModes` состоит из документа, содержащего один или более ключей (в нашем случае `dc` и `rackNY`) с целочисленными значениями. Эти числа задают количество различных тегированных значений указанного ключа, которые должны быть удовлетворены, чтобы выполнение команды `getLastError` считалось успешным. Определив режимы, вы можете использовать их в качестве значения параметра `w` в своем приложении. Например, в Ruby для указания первого режима надо написать такой код:

```
@collection.insert(doc, :safe => {:w => "multiDC"})
```

Тегирование не только придает дополнительную гибкость механизму гарантирования записи, но и обещает обеспечить более точный контроль над репликами, используемыми для масштабирования чтения. К сожалению, на момент написания этой книги семантика использования тегов для управления чтением еще не определена или не реализована в официальных драйверах MongoDB. Актуальную информацию по этому вопросу применительно к драйверу для Ruby см. на странице https://jira.mongodb.org/browse/RUBY-326.

8.5. Резюме

Из всего вышесказанного должно быть понятно, что репликация – исключительно полезная вещь, которая к тому же в большинстве систем просто необходима. Механизм репликации MongoDB задуман простым для применения, и обычно его конфигурирование действительно не вызывает затруднений. Но когда речь заходит о резервном копировании и отработке отказов, без скрытых сложностей не обойтись. Надеемся, что в таких, более сложных, случаях на помощь придет опыт и материал, изложенный в этой главе.

ГЛАВА 9.
Сегментирование

В этой главе:

- Идея сегментирования.
- Настройка и загрузка тестового сегментированного кластера.
- Администрирование и отработка отказов.

MongoDB изначально проектировалась с учетом сегментирования. Эта цель всегда была амбициозной, потому что придумать систему, которая поддерживает автоматическое сегментирование и балансирование на базе диапазонов и не имеет общей точки отказа, – задача непростая. Поэтому первый вариант поддержки сегментирования промышленного уровня появился лишь в версии MongoDB 1.6, выпущенной в августе 2010 года. С тех пор в подсистему сегментирования были включены многочисленные улучшения. По существу, механизм сегментирования позволяет равномерно распределять большие объемы данных по нескольким узлам и при необходимости наращивать емкость. В этой главе я расскажу о слое ПО, который делает всё это возможным.

Мы начнем с обзора сегментирования, обсудим, что это такое, зачем нужно и как реализовано в MongoDB. Хотя этого будет достаточно для формирования базового представления о сегментировании, полное понимание придет только после настройки собственного сегментированного кластера. Этим мы займемся во втором разделе, где построим тестовый кластер для хранения данных массивного приложения типа Google Docs. Затем мы обсудим некоторые внутренние механизмы сегментирования и опишем, как в этом случае работают запросы и индексы. Мы также рассмотрим важнейший вопрос о выборе сегментного ключа. И в конце главы я дам конкретные рекомендации по эксплуатации сегментирования в производственной системе.

Сегментирование – сложная тема. Чтобы извлечь из этой главы максимум пользы, необходимо проработать примеры. Создать тестовый кластер вполне можно на одной машине; сделайте это, а потом приступайте к экспериментам. Для того чтобы понять, как превратить MongoDB в распределенную систему, нет ничего более полезного, чем «живой» сегментированный кластер.

9.1. Обзор сегментирования

Прежде чем приступать к построению своего первого сегментированного кластера, нужно понять, что такое сегментирование вообще и почему оно иногда необходимо. Объяснение смысла сегментирования приведет нас к одному из основных обоснований проекта MongoDB в целом. Поняв, почему сегментирование так важно, вы с большим интересом прочитаете о том, из каких базовых компонентов состоит сегментированный кластер, и о ключевых концепциях, стоящих за механизмом сегментирования в MongoDB.

9.1.1. Что такое сегментирование

До сих пор вы использовали MongoDB как одиночный сервер, то есть каждый экземпляр `mongod` управлял полной копией данных приложения. Даже в случае репликации каждая реплика содержит полную копию всех данных – такую же, как на любой другой реплике. Для большинства приложений хранение всего набора данных на одном сервере вполне приемлемо. Но по мере того как объем данных растет, приложение начинает предъявлять более высокие требования к пропускной способности по чтению и записи, и мощности стандартных серверов перестает хватать. Например, стандартный сервер, возможно, не способен адресовать достаточный объем ОЗУ или ему не хватает процессорных ядер для эффективной обработки рабочей нагрузки. К тому же, по мере роста размера данных хранение резервных копий на одном диске или RAID-массиве может оказаться практически неосуществимым. Если вы собираетесь и дальше пользоваться для размещения базы данных стандартным или виртуализированным оборудованием, то решением всех этих проблем будет распределение базы на несколько серверов. Соответствующая методика и называется сегментированием.

Самые разные веб-приложения, в том числе Flickr и LiveJournal, реализовали схемы ручного сегментирования для распределения на-

грузки между несколькими базами данных MySQL. В этих реализациях логика сегментирования целиком находится внутри приложения. Чтобы понять, как это работает, представьте, что число пользователей настолько велико, что таблицу Users необходимо распределить между несколькими серверами баз данных. Для этого можно было бы назначить одну базу данных справочной. В ней будут храниться метаданные об отображении идентификаторов пользователей (или диапазонов идентификаторов) на сегмент. Таким образом, запрос на получение информации о пользователе фактически включает в себя два запроса: сначала обратиться к справочной базе данных и получить адрес сегмента для данного пользователя, а затем запросить у этого сегмента данные.

В вышеупомянутых веб-приложениях ручная схема сегментирования решает проблему распределения нагрузки, но реализации свойственны определенные недостатки. Самый заметный – сложность миграции данных. Если какой-то сегмент оказывается перегружен, то миграция данных с него на другие сегменты осуществляется вручную. Вторая проблема – трудности, сопряженные с написанием надежного кода для маршрутизации операций чтения и записи и управления базой данных как единым целым. Сравнительно недавно были разработаны каркасы для ручного управления сегментированием, самый известный из них – Gizzard от Twitter (см. http://mng.bz/4qvd).

Но всякий, кто хоть раз занимался ручным сегментированием базы данных, скажет, что добиться желаемого непросто. Проект MongoDB во многом был затеян как раз для решения этой проблемы. Поскольку сегментирование встроено на самом нижнем уровне MongoDB, пользователям нет нужды проектировать внешние средства сегментирования для обеспечения горизонтального масштабирования. Особенно это важно для решения трудной задачи балансирования данных между сегментами. Необходимый для этого код за выходной не напишешь.

Но, пожалуй, еще важнее, что MongoDB предоставляет приложению единый интерфейс, не зависящий от того, сегментирована база данных или нет. Это означает, что после перехода на сегментированную архитектуру в приложение не придется вносить почти или вообще никаких изменений.

Теперь вы более-менее понимаете, какие соображения стоят за автоматизацией сегментирования. Но перед тем как описывать процедуру сегментирования в MongoDB более подробно, стоит остано-

виться и задать себе очевидный вопрос: когда сегментирование необходимо?

Когда имеет смысл сегментировать

Вопрос о том, когда сегментировать, проще, чем кажется на первый взгляд. Мы уже говорили о том, как важно, чтобы индексы и рабочий набор данных помещались в оперативной памяти. Обеспечение этого и есть основная причина сегментирования. Если объем данных приложения может неограниченно расти, то рано или поздно наступит момент, когда они больше не помещаются в память. Если вы разместили свое приложение в облаке Amazon EC2, то порогом является величина 68 ГБ, потому что на момент написания этой книги таким был объем памяти в самом большом из доступных экземпляров. Можно, конечно, разместить приложение на собственном оборудовании, оснащенном гораздо большим объемом памяти, тогда момент перехода на сегментированный кластер, быть может, удастся немного оттянуть. Но все равно объем ОЗУ не бесконечен, поэтому рано или поздно сегментирование станет неизбежностью.

Скажем честно – кое-какие поправки в сказанное выше надо внести. Например, если вы эксплуатируете собственное оборудование и можете хранить все данные на твердотельных дисках (а эта возможность становится все более доступной в финансовом плане), то отношение объема данных к объему ОЗУ можно увеличить без негативных последствий для производительности. Не исключено также, что размер рабочего набора составляет малую долю от общего размера данных, то есть приложение может работать при относительно небольшом объеме ОЗУ. Но верно и обратное: если рабочая нагрузка подразумевает особенно большое количество операций записи, то необходимость в сегментировании может возникнуть *задолго до того*, как размер данных сравняется с объемом ОЗУ, – просто потому, что для обеспечения требуемой пропускной способности по записи необходимо распределить нагрузку между несколькими машинами.

Как бы то ни было, решение сегментировать существующую систему всегда принимается на основе регулярного анализа дисковых операций, загруженности системы и отношения размера рабочего набора к объему оперативной памяти.

9.1.2. Как работает сегментирование

Чтобы разобраться в том, как устроено сегментирование в MongoDB, необходимо знать о компонентах сегментированного кластера и

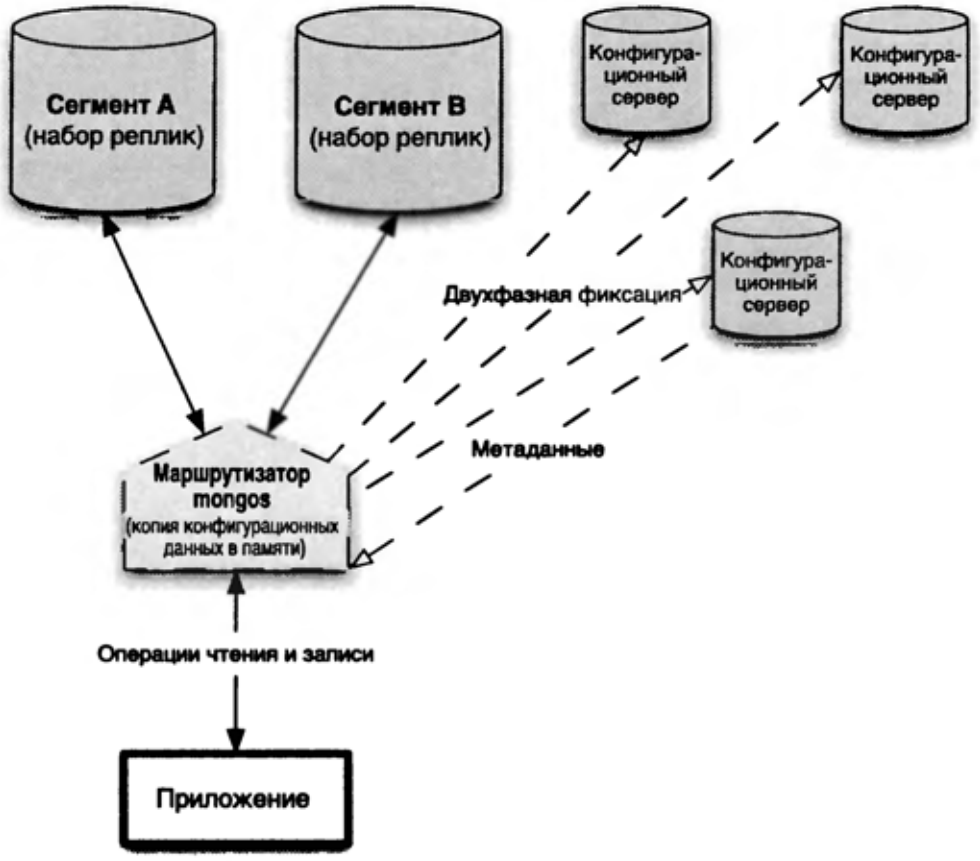


Рис. 9.1. Компоненты сегментированного кластера MongoDB

процессах, координирующих совместную работу этих компонентов. Это и будет темой следующих двух разделов.

Компоненты сегментирования

Сегментированный кластер состоит из сегментов, маршрутизаторов `mongos` и конфигурационных серверов. При обсуждении этих компонентов мы будем обращаться к рис. 9.1.

Сегменты

Сегментированный кластер MongoDB распределяет данные по одному или нескольким сегментам. Каждый сегмент развертывается в виде набора реплик, на котором хранится некоторая часть всего множества данных. Поскольку сегмент представляет собой набор реплик, то он поддерживает автоматическую отработку отказов. К отдельному сегменту можно подключаться так же, как к автономному набору реплик. Но если вы подключитесь к одному набору реплик, то будете видеть только часть всех данных кластера.

Маршрутизаторы mongos

Коль скоро каждый сегмент содержит лишь часть данных кластера, то необходим интерфейс для обращения к кластеру как к единому

целому. Тут-то и вступают в игру процессы mongos. Каждый такой процесс – это маршрутизатор, перенаправляющий все операции чтения и записи подходящему сегменту. За счет этого mongos предоставляет клиентам единый взгляд на систему в целом.

Процессы mongos потребляют мало ресурсов и не отвечают за сохранение данных. Обычно они запускаются на тех же машинах, где работают серверы приложений, так что для доступа к любому сегменту необходимо лишь один сетевой переприем. Иначе говоря, приложение локально соединяется с mongos, а mongos уже управляет соединениями с конкретными сегментами.

Конфигурационные серверы

Если процессы mongos не сохраняют данные, то должен быть какой-то компонент, отвечающий за долговременное хранение канонического состояния сегментированного кластера; этим занимаются конфигурационные серверы, которые хранят метаданные кластера. В состав этих данных входят: глобальная конфигурация кластера, местоположение каждой базы данных, коллекции и диапазоны хранящихся в них данных, а также журнал изменений, в котором сохраняется история миграции данных между сегментами.

Метаданные, хранящиеся на конфигурационных серверах, играют важнейшую роль для правильного функционирования и обслуживания кластера. Например, при каждом запуске процесс mongos получает копию метаданных от конфигурационных серверов. Без этих данных невозможно обеспечить согласованное представление сегментированного кластера. Таким образом, важность этих данных диктует стратегию проектирования и развертывания конфигурационных серверов.

На рис. 9.1 видно, что имеется три конфигурационных сервера, но они не являются наборами реплик. Для их работы асинхронной репликации недостаточно; для отправки им данных из процесса mongos применяется протокол двухфазной фиксации, который гарантирует согласованность всех конфигурационных серверов. В любой производственной системе, где применяется сегментирование, должно быть ровно три конфигурационных сервера, причем для обеспечения резервирования они должны размещаться на разных машинах[1].

---

[1] Можно работать и с одним конфигурационным сервером, но только ради упрощения тестирования. Наличие одного конфигурационного сервера в производственной системе можно сравнить с перелетом через Атлантику на самолете с одним двигателем: возможно, и долетите, но стоит двигателю отказать, и можете писать пропало.

Теперь вы знаете, из чего состоит сегментированный кластер, но, наверное, еще не понимаете, как работает сам механизм сегментирования. Каким образом происходит распределение данных? Я отвечу на этот вопрос в следующем разделе, описав основные операции сегментирования.

Основные операции сегментирования

Кластер MongoDB распределяет данные между сегментами на двух уровнях, из которых более грубым является база данных. Каждой создаваемой в кластере базе данных назначается свой сегмент. Если специально ничего не предпринимать, то эта база и все ее коллекции так и останутся в том сегменте, где были созданы.

Но поскольку приложения обычно хранят все свои данные в одной физической базе, то такое распределение не особенно полезно. Необходимо распределение на более детальном уровне, и коллекции как раз удовлетворяют этому требованию. Механизм сегментирования в MongoDB спроектирован специально для распределения отдельных коллекций между сегментами. Чтобы лучше разобраться, подумаем, как это можно использовать в реальном приложении.

Допустим, вы разрабатываете «облачный» комплект программ для работы с электронными таблицами и намереваетесь хранить все данные в MongoDB[2]. Пользователи смогут создавать сколько угодно документов, каждый из которых будет представлен отдельным документом MongoDB в одной коллекции `spreadsheets`. Предположим, что со временем количество пользователей приложения достигло миллиона. Теперь представьте себе две основных коллекции: `users` и `spreadsheets`. Коллекция `users` вполне управляема; даже при миллионе пользователей и размере документа о пользователе 1 КБ она занимает примерно 1 ГБ и легко обслуживается единственной машиной. Чего не скажешь о коллекции `spreadsheets`. В предположении, что каждый пользователь владеет в среднем 50 электронными таблицами и каждая таблица в среднем занимает 50 КБ, размер всей коллекции составит порядка 1 ТБ. Если приложение активно используется, то все данные желательно хранить в оперативной памяти. Но для этого коллекцию придется сегментировать и позаботиться о распределении операций чтения и записи.

Сегментирование коллекции

В MongoDB сегментирование производится по диапазону. Это означает, что для каждого документа в сегментированной коллекции

[2] Вспомните о комплекте Google Docs, который, среди прочего, позволяет создавать электронные таблицы и презентации.

определяется, в какой диапазон попадает значение заданного ключа. Чтобы отнести документ к конкретному диапазону, в MongoDB используется так называемый *сегментный ключ* (shard key)[3]. Идея станет понятнее на примере документа из гипотетического приложения для работы с электронными таблицами:

```
{
   _id: ObjectId("4d6e9b89b600c2c196442c21")
   filename: "spreadsheet-1",
   updated_at: ISODate("2011-03-02T19:22:54.845Z"),
   username: "banks",
   data: "raw document data"
}
```

При сегментировании этой коллекции необходимо объявить одно или несколько полей, которые станут сегментным ключом. Если в этом качестве использовать поле `_id`, то документы будут распределяться по диапазонам идентификаторов объектов. Однако по причинам, которые станут ясны позже, лучше в качестве сегментного ключа взять комбинацию полей `username` и `_id`; тогда каждый диапазон будет представлять некоторую последовательность имен пользователей.

Теперь можно перейти к понятию *порции* (chunk). Порция – это непрерывный диапазон значений сегментных ключей, находящихся в одном сегменте. Для примера представьте коллекцию `docs`, распределенную между двумя сегментами A и B и разбитую на порции, как показано в табл. 9.1. Диапазон каждой порции обозначен начальным и конечным значением.

Даже при беглом взгляде на эту таблицу бросается в глаза основное, часто противоречащее интуиции, свойство порций: хотя каждая порция представляет непрерывный диапазон данных, сами диапазоны могут находиться в любом сегменте.

Таблица 9.1. Порции и сегменты

Начало	Конец	Сегмент
–∞	abbot	B
abbot	dayton	A
dayton	harris	B
harris	norris	A
norris	∞	B

[3] В других распределенных базах данных используются также термины *секционный ключ* и *распределительный ключ*.

Еще одна важная характеристика порций – это то, что они не *физические*, а *логические*. Другими словами, порции не соответствует последовательность соседних документов на диске. Когда мы говорим, что в сегменте A имеется порция, начинающаяся с *harris* и кончающаяся *norris*, то имеем в виду, что всякий документ с сегментным ключом из этого диапазона будет находиться в коллекции `docs` в сегменте A. Никаких выводов о *физическом порядке* документов в этой коллекции сделать нельзя.

Расщепление и миграция

В основе механизма сегментирования лежит расщепление и миграция порций.

Сначала рассмотрим идею расщепления порций. Сразу после настройки сегментированного кластера существует всего одна порция. Ее диапазон охватывает всю сегментированную коллекцию. Но как же тогда образуется несколько порций? Ответ прост – порция расщепляется по достижении определенного порогового размера. По умолчанию максимальный размер порции составляет 64 МБ или 100 000 документов, в зависимости от того, что будет достигнуто раньше. По мере добавления данных в новый сегментированный кластер первоначальная порция в конце концов достигает того или другого порога, после чего происходит ее расщепление. Расщепление – простая операция; по существу, она сводится к разбиению исходного диапазона на два поддиапазона и созданию двух новых порций с одинаковым количеством документов в каждой.

Отметим, что расщепление порции – это *логическая* операция. MongoDB просто изменяет метаданные, так что из одной порции получается две. Поэтому расщепление не оказывает влияния на физическое расположение документов в сегментированной коллекции, а, значит, происходит быстро.

Но, как вы помните, одна из самых трудных проблем при проектировании сегментированной системы – обеспечить равномерное балансирование данных. Кластеры MongoDB достигают этого за счет перемещения порций между сегментами. Эта операция называется *миграцией* и, в отличие, от расщепления она подразумевает физические действия.

Миграциями управляет процесс, который называется *балансировщиком*. Его задача заключается в том, чтобы обеспечить равномерное распределение данных между сегментами. Чтобы ее решить, балансировщик учитывает количество порций в каждом сегменте. Эвристи-

ческие правила в какой-то мере зависят от полного размера данных, но обычно балансирование производится, когда разность между сегментом с наибольшим количеством порций и сегментом с наименьшим количеством порций оказывается больше восьми. В процессе балансирования порции перемещаются с сегмента, где их больше, на сегмент, где их меньше, пока количество порций не станет примерно одинаковым.

Если пока всё это не вполне понятно, ничего страшного. В следующем разделе я продемонстрирую сегментирование на примере тестового кластера, и понятия сегментного ключа и порции обретут практический смысл.

9.2. Тестовый сегментированный кластер

Лучший способ разобраться в механизме сегментирования – посмотреть на него в действии. К счастью, вполне возможно организовать сегментированный кластер на одной машине, что мы сейчас и сделаем[4]. Затем мы смоделируем поведение описанного в предыдущем разделе облачного приложения для работы с электронными таблицами. Попутно мы рассмотрим конфигурацию глобальных сегментов и на собственном опыте увидим, как производится распределение данных на основе сегментного ключа.

9.2.1. Настройка

Сегментированный кластер настраивается в два этапа. На первом запускаются все необходимые процессы `mongod` и `mongos`. На втором, более простом, выполняются команды инициализации кластера. Наш кластер будет состоять из двух сегментов и трех конфигурационных серверов. При этом мы запустим всего один процесс `mongos` для взаимодействия с кластером. На рис. 9.2 представлены все рабочие процессы, и для каждого в скобках указан номер порта.

Чтобы кластер заработал, нужно будет выполнить ряд команд; если по ходу дела вы перестанете видеть за деревьями лес, то обращайтесь к рисунку.

---

[4] Идея в том, чтобы запустить процессы `mongod` и `mongos` на одной машине для тестирования. Ниже в этой главе мы рассмотрим конфигурирование сегментированного кластера в производственной системе и поговорим о минимальном количестве машин, необходимых для сколько-нибудь полезного развертывания.

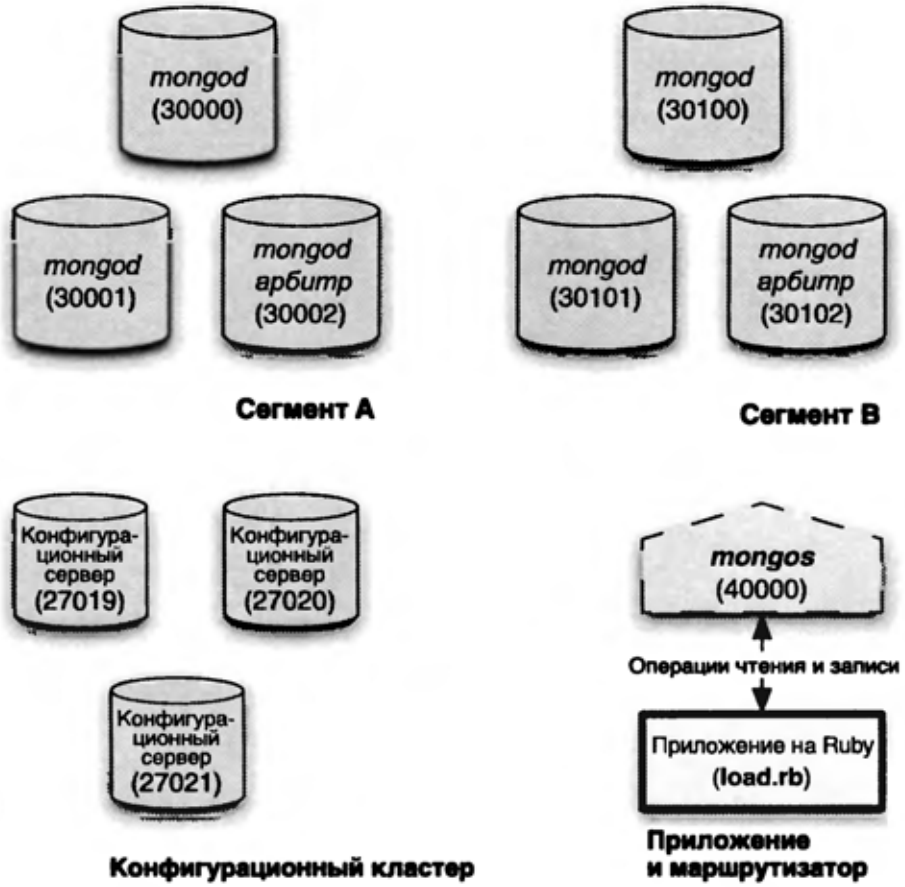


Рис. 9.2. Процессы, составляющие тестовый сегментированный кластер

Инициализация компонентов кластера

Начнем с создания каталогов данных для двух наборов реплик, которые станут нашими сегментами.

```
$ mkdir /data/rs-a-1
$ mkdir /data/rs-a-2
$ mkdir /data/rs-a-3
$ mkdir /data/rs-b-1
$ mkdir /data/rs-b-2
$ mkdir /data/rs-b-3
```

Затем запустим процессы `mongod`. Поскольку процессов будет много, то указывайте параметр `--fork`, чтобы все они работали в фоновом режиме[5]. Ниже показаны команды для запуска первого набора реплик.

```
$ mongod --shardsvr --replSet shard-a --dbpath /data/rs-a-1 \
         --port 30000 --logpath /data/rs-a-1.log --fork --nojournal
$ mongod --shardsvr --replSet shard-a --dbpath /data/rs-a-2 \
```

[5] В Windows параметр `fork` не работает. Поскольку для каждого процесса придется открывать отдельное консольное окно, рекомендую также опустить параметр `logpath`.

```
        --port 30001 --logpath /data/rs-a-2.log --fork --nojournal
$ mongod --shardsvr --replSet shard-a --dbpath /data/rs-a-3 \
        --port 30002 --logpath /data/rs-a-3.log --fork --nojournal
```

А это команды для запуска второго набора реплик:

```
$ mongod --shardsvr --replSet shard-b --dbpath /data/rs-b-1 \
        --port 30100 --logpath /data/rs-b-1.log --fork --nojournal
$ mongod --shardsvr --replSet shard-b --dbpath /data/rs-b-2 \
        --port 30101 --logpath /data/rs-b-2.log --fork --nojournal
$ mongod --shardsvr --replSet shard-b --dbpath /data/rs-b-3 \
        --port 30102 --logpath /data/rs-b-3.log --fork --nojournal
```

Как обычно, наборы реплик необходимо инициализировать. Подключитесь к каждому по очереди, вызовите метод `rs.initiate()`, а затем добавьте узлы. Для первого набора процедура выглядит так[6]:

```
$ mongo arete:30000
> rs.initiate()
```

Подождите одну-две минуты, пока начальный узел не станет первичным. После этого добавляйте остальные узлы:

```
> rs.add("arete:30000")
> rs.add("arete:30001", {arbiterOnly: true})
```

Второй набор реплик инициализируется аналогично. И в этом случае подождите минутку после вызова метода `rs.initiate()`:

```
$ mongo arete:30100
> rs.initiate()
> rs.add("arete:30100")
> rs.add("arete:30101", {arbiterOnly: true})
```

Наконец, проверьте, что оба набора реплик заработали, для чего вызовите для каждого метод `rs.status()` из оболочки. Если все нормально, то можно переходить к запуску конфигурационных серверов[7]. Создайте для каждого конфигурационного сервера каталог данных и запустите по одному процессу `mongod` с параметром `configsvr`:

```
$ mkdir /data/config-1
$ mongod --configsvr --dbpath /data/config-1 --port 27019 \
        --logpath /data/config-1.log --fork --nojournal

$ mkdir /data/config-2
$ mongod --configsvr --dbpath /data/config-2 --port 27020 \
        --logpath /data/config-2.log --fork --nojournal
```

---

[6] `arete` – имя локального компьютера.

[7] При работе в Windows опустите параметры `--fork` и `--logpath` и запускайте каждый процесс `mongod` в отдельном окне.

```
$ mkdir /data/config-3
$ mongod --configsvr --dbpath /data/config-3 --port 27021 \
         --logpath /data/config-3.log --fork --nojournal
```

Проверьте, что все три конфигурационных сервера запустились, для чего подключитесь к каждому из оболочки или выполните команду `tail` для файлов журналов и убедитесь, что каждый процесс прослушивает заданный для него порт. Конец журнала каждого конфигурационного сервера должен выглядеть примерно так:

```
Wed Mar 2 15:43:28 [initandlisten] waiting for connections on port 27020
Wed Mar 2 15:43:28 [websvr] web admin interface listening on port 28020
```

Если все конфигурационные серверы работают, то можно запускать `mongos`. При его запуске следует указать параметр `configdb`, значением которого является список адресов конфигурационных баз данных через запятую[8]:

```
$ mongos --configdb arete:27019,arete:27020,arete:27021 \
      --logpath /data/mongos.log --fork --port 40000
```

Конфигурирование кластера

Теперь все компоненты сегментированного кластера на месте и можно заняться его конфигурированием. Начнем с процесса `mongos`. Чтобы упростить задачу, воспользуемся вспомогательными методами сегментирования. Все они вызываются от имени глобального объекта `sh`. Чтобы вывести список имеющихся методов, выполните команду `sh.help()`.

Вам предстоит ввести последовательность конфигурационных команд, начиная с `addshard`. Вспомогательный метод для этой команды называется `sh.addShard()`. Он принимает строку, содержащую имя набора реплик, за которым следуют адреса двух или более начальных узлов, к которым можно подключаться. В данном случае мы указываем оба созданных ранее набора реплик, а также адреса двух членов каждого набора, не являющихся арбитрами:

```
$ mongo arete:40000
> sh.addShard("shard-a/arete:30000,arete:30001")
   { "shardAdded" : "shard-a", "ok" : 1 }
> sh.addShard("shard-b/arete:30100,arete:30101")
   { "shardAdded" : "shard-b", "ok" : 1 }
```

Если всё пройдет удачно, то в ответе команды будет присутствовать имя только что добавленного сегмента. Чтобы ознакомиться с

[8] Между адресами не должно быть пробелов.

результатом своих действий, можно опросить коллекцию shards в конфигурационной базе данных. Для смены текущей базы используйте не команду use, а метод getSiblingDB():

```
> db.getSiblingDB("config").shards.find()
{ "_id" : "shard-a", "host" : "shard-a/arete:30000,arete:30001" }
{ "_id" : "shard-b", "host" : "shard-b/arete:30100,arete:30101" }
```

Более короткая команда listshards возвращает ту же самую информацию:

```
> use admin
> db.runCommand({listshards: 1})
```

Раз уж мы заговорили о распечатке конфигурации сегментов, то стоит отметить метод оболочки sh.status(), который возвращает сводную информацию о кластере в удобном для восприятия формате. Попробуйте сами.

Следующий шаг – разрешить сегментирование базы данных. Это обязательное условие для сегментирования любой коллекции. База данных нашего приложения будет называться cloud-docs, поэтому команда выглядит так:

```
> sh.enableSharding("cloud-docs")
```

Как и раньше, можно опросить конфигурационные данные и посмотреть, какие были произведены изменения. В конфигурационной базе данных имеется коллекция databases, содержащая список баз данных. В каждом документе из этого списка указано местоположение основного сегмента базы и признак partitioned, говорящий о том, сегментирована она или нет:

```
> db.getSiblingDB("config").databases.find()
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "cloud-docs", "partitioned" : true, "primary" : "shard-a" }
```

Теперь осталось сегментировать коллекцию spreadsheets. При этом определяется сегментный ключ. В данном случае мы используем составной ключ {username: 1, _id: 1}, поскольку он удобен для распределения данных и позволяет наглядно представить диапазоны порций:

```
> sh.shardCollection("cloud-docs.spreadsheets", {username: 1, _id: 1})
```

После выполнения команды проверьте в конфигурационной базе данных, какие коллекции теперь сегментированы:

```
> db.getSiblingDB("config").collections.findOne()
```

```
{
   "_id" : "cloud-docs.spreadsheets",
   "lastmod" : ISODate("1970-01-16T00:50:07.268Z"),
   "dropped" : false,
   "key" : {
      "username" : 1,
      "_id" : 1
   },
   "unique" : false
}
```

Возможно, определение этой сегментированной коллекции кое-что вам напомнило; действительно, оно похоже на определение индекса, особенно в части ключа `unique`. Если сегментируется пустая коллекция, то MongoDB создает для каждого сегмента индекс по сегментному ключу[9]. Убедимся в этом, подключившись к сегменту и вызвав метод `getIndexes()`. В примере ниже мы подключаемся к первому сегменту, и, как и следовало ожидать, видим индекс по сегментному ключу:

```
$ mongo arete:30000
> use cloud-docs
> db.spreadsheets.getIndexes()
[
   {
      "name" : "_id_",
      "ns" : "cloud-docs.spreadsheets",
      "key" : {
         "_id" : 1
      },
      "v" : 0
   },
   {
      "ns" : "cloud-docs.spreadsheets",
      "key" : {
         "username" : 1,
         "_id" : 1
      },
      "name" : "username_1__id_1",
      "v" : 0
   }
]
```

После того как коллекция сегментирована, кластер наконец-то готов к работе. Если теперь вы начнете писать в кластер, то данные будут распределяться по сегментам. В следующем разделе мы увидим, как это происходит.

---

[9] Если сегментируется уже существующая коллекция, то такой индекс нужно построить до выполнения команды `shardcollection`.

9.2.2. Запись в сегментированный кластер

Мы будем писать в сегментированную коллекцию и наблюдать за формированием и перемещением порций, что и составляет суть сегментирования в MongoDB. Тестовый документ, представляющий одну электронную таблицу, имеет такой вид:

```
{
   _id: ObjectId("4d6f29c0e4ef0123afdacaeb"),
   filename: "sheet-1",
   updated_at: new Date(),
   username: "banks",
   data: "RAW DATA"
}
```

Отметим, что поле data будет содержать строку размером 5 КБ имитирующую неформатированные данные. В исходном коде, прилагаемом к этой книге, имеется скрипт на Ruby, с помощью которого можно записывать документы в кластер. Скрипт принимает в качестве аргумента число итераций, и на каждой итерации вставляет по одному документу размером 5 КБ для каждого из 200 пользователей Ниже приведен исходный код этого скрипта.

```
require 'rubygems'
require 'mongo'
require 'names'

@con = Mongo::Connection.new("localhost", 40000)
@col = @con['cloud']['spreadsheets']
@data = "abcde" * 1000

def write_user_docs(iterations=0, name_count=200)
   iterations.times do |n|
      name_count.times do |n|
         doc = {     :filename => "sheet-#{n}",
                           :updated_at => Time.now.utc,
                           :username => Names::LIST[n],
                           :data => @data
         }
         @col.insert(doc)
      end
   end
end

if ARGV.empty? || !(ARGV[0] =~ /^d+$/)
   puts "Usage: load.rb [iterations] [name_count]"
else
   iterations = ARGV[0].to_i

   if ARGV[1] && ARGV[1] =~ /^d+$/
```

```
      name_count = ARGV[1].to_i
   else
      name_count = 200
   end

   write_user_docs(iterations, name_count)
end
```

Если выполнить этот скрипт из командной строки, не задавая аргументов, то он вставит первые 200 документов:

```
$ ruby load.rb
```

Теперь подключитесь к `mongos` из оболочки. Опросив коллекцию `spreadsheets`, вы увидите, что она содержит ровно 200 документов общим объемом примерно 1 МБ. Можете запросить какой-нибудь документ, но не забудьте исключить поле `data` (если не хотите выводить на экран 5 КБ текста).

```
$ mongo arete:40000
> use cloud-docs
> db.spreadsheets.count()
200
> db.spreadsheets.stats().size
1019496
> db.spreadsheets.findOne({}, {data: 0})
{
   "_id" : ObjectId("4d6d6b191d41c8547d0024c2"),
   "username" : "Cerny",
   "updated_at" : ISODate("2011-03-01T21:54:33.813Z"),
   "filename" : "sheet-0"
}
```

Теперь можно посмотреть, что произошло с кластером. Переключитесь на базу данных `config` и запросите количество порций:

```
> use config
> db.chunks.count()
1
```

Пока что есть всего одна порция. Посмотрим, на что она похожа:

```
> db.chunks.findOne()
{
   "_id" : "cloud-docs.spreadsheets-username_MinKey_id_MinKey",
   "lastmod" : {
      "t" : 1000,
      "i" : 0
   },
   "ns" : "cloud-docs.spreadsheets",
   "min" : {
      "username" : { $minKey : 1 },
```

```
      "_id" : { $minKey : 1 }
   },
   "max" : {
      "username" : { $maxKey : 1 },
      "_id" : { $maxKey : 1 }
   },
   "shard" : "shard-a"
}
```

Понятно ли вам, какой диапазон представляет эта порция? Раз порция всего одна, то она покрывает всю сегментированную коллекцию. Это подтверждают поля `min` и `max`, показывающие, что диапазон порции ограничен ключами `$minKey` и `$maxKey`.

MINKEY и MAXKEY. Значения `$minKey` и `$maxKey` используются в операторах сравнения как границы типов BSON. `$minKey` меньше, а `$maxKey` больше любого значения любого типа BSON. Поскольку значение любого поля может принадлежать произвольному типу BSON, то MongoDB использует эти два специальных маркера, чтобы обозначить, что граничная точка порции совпадает с минимальным или максимальным значением в коллекции.

Чтобы увидеть более интересный пример диапазона порции, нужно добавить в коллекцию `spreadsheets` еще данные. Мы снова воспользуемся скриптом на Ruby, но на этот раз зададим 100 итераций, чтобы вставить 20 000 документов общим объемом 100 МБ:

```
$ ruby load.rb 100
```

Проверим, что вставка произошла:

```
> db.spreadsheets.count()
20200
> db.spreadsheets.stats().size
103171828
```

Скорость тестовой вставки

Вставка этих данных в сегментированный кластер может занять несколько минут. Такая медлительность объясняется тремя причинами. Во-первых, каждая вставка влечет за собой обращение к серверу, хотя в производственной системе можно было бы прибегнуть к массовой вставке. Во-вторых, вставку выполняет программа на Ruby, а BSON-сериализатор на этом языке работает медленнее, чем в некоторых других драйверах. Наконец, и это самое главное, все узлы сегментированного кластера организованы на одной машине. Это дает очень большую нагрузку на диск, поскольку запись на все четыре узла (два первичных и два вторичных узла наборов реплик) производится одновременно. Отметим, однако, что в правильно сконфигурированной производственной системе вставка работала бы гораздо быстрее.

После вставки такого объема данных порций точно будет несколько. Быстро узнать, сколько именно, можно, запросив количество документов в коллекции chunks:

```
> use config
> db.chunks.count()
10
```

Для получения более детальной информации вызовем метод sh.status(), который напечатает все порции с указанием их диапазонов. Для краткости ниже показаны только первые две порции:

```
> sh.status()
sharding version: { "_id" : 1, "version" : 3 }
   shards:
   { "_id": "shard-a", "host": "shard-a/arete:30000,arete:30001" }
   { "_id": "shard-b", "host": "shard-b/arete:30100,arete:30101" }

   databases:
   { "_id": "admin", "partitioned": false, "primary": "config" }
   { "_id": "test", "partitioned": false, "primary": "shard-a" }
   { "_id": "cloud-docs", "partitioned": true, "primary": "shard-b" }
         shard-a 5
         shard-b 5
      {  "username": { $minKey : 1 }, "_id" : { $minKey : 1 } } --
               >> { "username": "Abdul",
         "_id": ObjectId("4e89ffe7238d3be9f0000012") }
      on: shard-a { "t" : 2000, "i" : 0 }

      {  "username" : "Abdul",
         "_id" : ObjectId("4e89ffe7238d3be9f0000012") } -->> {
         "username" : "Buettner",
         "_id" : ObjectId("4e8a00a0238d3be9f0002e98") }
      on : shard-a { "t" : 3000, "i" : 0 }
```

Картина определенно изменилась. Теперь у нас 10 порций. Естественно, каждая порция представляет непрерывный диапазон данных. Как видите, в первой порции находятся документы с ключами от $minKey до Abdul, а во второй – от Abdul до Buettner[10]. Но мало того что порций стало больше, они еще и мигрировали на второй сегмент. Чтобы убедиться в этом, можно внимательно просмотреть выдачу sh.status(), но есть и более простой способ:

```
> db.chunks.count({"shard": "shard-a"})
5
> db.chunks.count({"shard": "shard-b"})
5
```

[10] Если вы прорабатывали примеры, то распределение документов по порциям может немного отличаться.

Пока размер кластера невелик, расщепление производится часто, что мы и наблюдаем. В результате с самого начала получается хорошее распределение данных и несколько порций. Но теперь равномерное распределение по уже имеющимся диапазонам может поддерживаться дольше, поэтому миграций будет меньше.

Раннее расщепление порций

Сегментированный кластер агрессивно производит расщепление порций на ранних стадиях, чтобы ускорить миграцию данных на сегменты. Точнее, когда количество порций меньше 10, расщепление происходит по достижении одной четверти максимального размера порции (16 МБ), а если имеется от 10 до 20 порций, то по достижении половины максимального размера (32 МБ). У такого подхода есть два достоинства. Во-первых, с самого начала создается много порций, что вызывает миграцию. Во-вторых, миграция обходится относительно недорого, так как благодаря небольшому размеру порции общий объем пересылаемых по сети данных мал.

Далее порог расщепления увеличивается. Чтобы убедиться в том, что расщепление действительно происходит реже, а размеры порций приближаются к максимальному, надо вставить еще больше данных. Попробуйте добавить в кластер еще 800 МБ:

```
$ ruby load.rb 800
```

Это займет много времени, так что можете отойти, выпить чашечку кофе и перекусить. Когда операция закончится, общий объем данных возрастет в восемь раз. Но, запросив количество порций, вы увидите, что оно увеличилось всего лишь вдвое:

```
> use config
> db.chunks.count()
21
```

Поскольку порций стало больше, средний диапазон одной порции уменьшился, но в каждой порции оказалось больше данных. Так, например, в первой порции находятся документы с ключами от Abbott до Bender, но ее размер уже приблизился к 60 МБ. Поскольку максимальный размер порции в настоящее время равен 64 МБ, то при последующей вставке данных эта порция должна скоро расщепиться.

Также стоит отметить, что распределение по-прежнему близко к равномерному:

```
> db.chunks.count({"shard": "shard-a"})
11
> db.chunks.count({"shard": "shard-b"})
10
```

Хотя после вставки 800 МБ количество порций возросло, можно предположить, что никаких миграций не было; вероятнее всего, каждая исходная порция расщепилась на две, и где-то произошло еще одно дополнительное расщепление. Убедиться в этом можно, опросив коллекцию `changelog` в конфигурационной базе данных:

```
> db.changelog.count({what: "split"})
20
> db.changelog.find({what: "moveChunk.commit"}).count()
6
```

Результат недалек от наших предположений. Всего имело место 20 расщеплений, что дало 20 порций, но при этом произошло только 6 миграций. Чтобы еще глубже разобраться в том, как развивался процесс, можно изучить записи в журнале изменений. Вот, например, запись, в которой зафиксировано первое перемещение порции:

```
> db.changelog.findOne({what: "moveChunk.commit"})
{
   "_id" : "arete-2011-09-01T20:40:59-2",
   "server" : "arete",
   "clientAddr" : "127.0.0.1:55749",
   "time" : ISODate("2011-03-01T20:40:59.035Z"),
   "what" : "moveChunk.commit",
   "ns" : "cloud-docs.spreadsheets",
   "details" : {
      "min" : {
         "username" : { $minKey : 1 },
         "_id" : { $minKey : 1 }
      },
      "max" : {
         "username" : "Abbott",
         "_id" : ObjectId("4d6d57f61d41c851ee000092")
      },
      "from" : "shard-a",
      "to" : "shard-b"
   }
}
```

Здесь мы видим перемещение порций из сегмента `shard-a` в сегмент `shard-b`. Вообще, документы, хранящиеся в журнале изменений, вполне понятны. Когда вы лучше познакомитесь с сегментированием и начнете строить прототипы собственных кластеров, журнал изменений станет замечательным подспорьем для изучения расщепления и миграции. Обращайтесь к нему почаще.

9.3. Индексирование сегментированного кластера и запросы к нему

С точки зрения приложения, нет никакой разницы между запросом к сегментированному кластеру и к одиночному серверу mongod. В обоих случаях интерфейс формулирования запроса и процедура обхода результирующего набора одинаковы. Но за кулисами всё происходит по-другому и интересно узнать, как именно.

9.3.1. Типы сегментированных запросов

Представьте, что вы предъявляете запрос сегментированному кластеру. К скольким сегментами должен обратиться процесс mongos для получения правильного ответа? Немного подумав, вы придете к выводу, что ответ зависит от того, присутствует ли в селекторе запроса сегментный ключ. Напомним, что конфигурационные серверы (и, следовательно, mongos) хранят соответствие между диапазонами и сегментами. И описывается оно в терминах порций, о которых мы говорили выше. Если запрос включает сегментный ключ, то mongos может быстро справиться с данными о порциях и определить, на какой сегменте искать результат. Запросы такого типа называются *направленными* (targeted query).

Но если сегментный ключ в запросе не упоминается, то для удовлетворения запроса придется посетить все сегменты. Такие запросы называются *глобальными* или *запросами с разделением и объединением* (scatter/gather query). На рис. 9.3 изображены запросы обоих типов.

Команда explain точно показывает путь выполнения запроса к сегментированному кластеру. Начнем с направленного запроса. В данном случае запрашивается документ, находящийся в первой порции нашей коллекции:

```
> selector = {username: "Abbott",
"_id" : ObjectId("4e8a1372238d3bece8000012")}
> db.spreadsheets.find(selector).explain()
{
   "shards" : {
      "shard-b/arete:30100,arete:30101" : [
         {
            "cursor" : "BtreeCursor username_1__id_1",
```

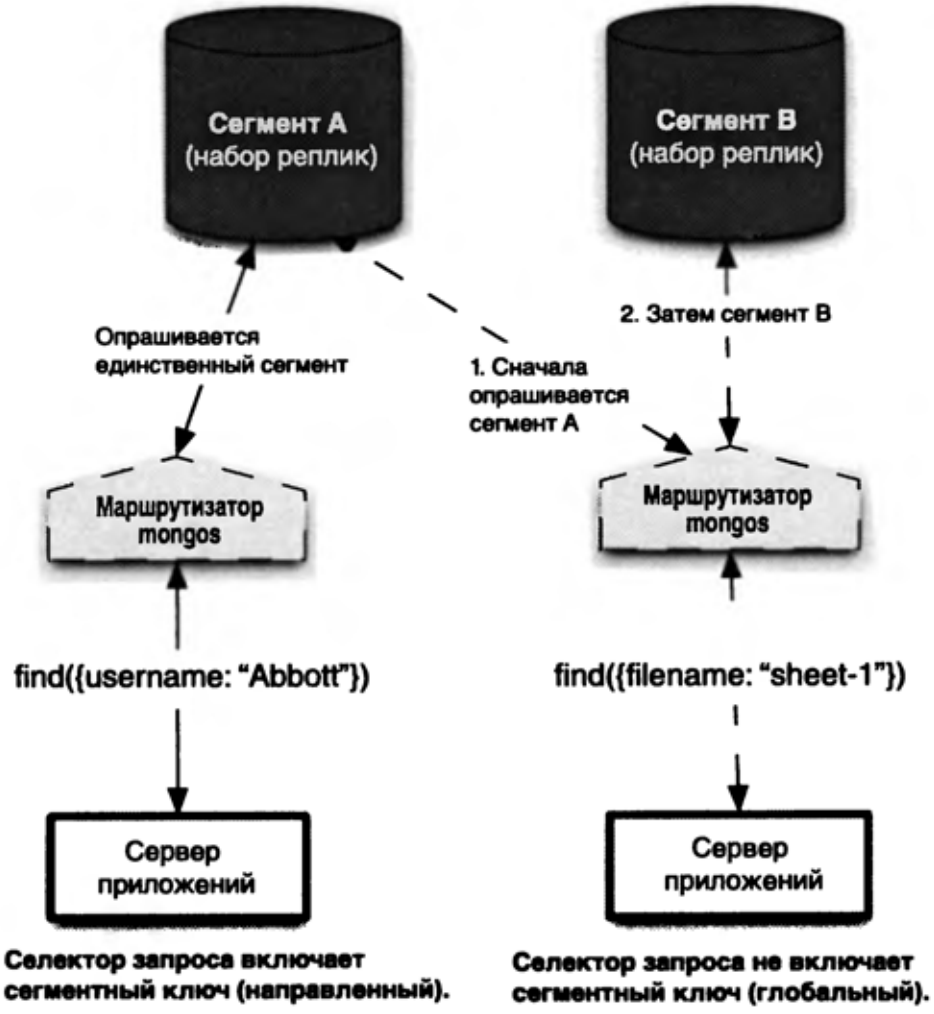


Рис. 9.3. Направленные и глобальные запросы к сегментированному кластеру

```
    "nscanned"    1,
          1,
    "millis"    0,
    "indexBounds"    {
       "username"    [
          [
             "Abbott"
             "Abbott"
          ]
       ],
       "_id"    [
          [
             ObjectId("4d6d57f61d41c851ee000092"),
             ObjectId("4d6d57f61d41c851ee000092")
          ]
       ]
    }
 }
]
```

```
    },
    "n" : 1,
    "nscanned" : 1,
    "millisTotal" : 0,
    "numQueries" : 1,
    "numShards" : 1
}
```

`Explain` четко показывает, что для возврата единственного документа запрашивался только сегмент B[11]. Оптимизатор запросов достаточно «умен», чтобы учитывать при маршрутизации запросов также префиксы сегментного ключа. Следовательно, в запросе можно указывать только имя пользователя:

```
> db.spreadsheets.find({username: "Abbott"}).explain()
{
    "shards" : {
        "shard-b/arete:30100,arete:30101" : [
            {
                "cursor" : "BtreeCursor username_1__id_1",
                "nscanned" : 801,
                "n" : 801,
            }
        ]
    },
    "n" : 801,
    "nscanned" : 801,
    "numShards" : 1
}
```

Этот запрос возвращает все 801 документов пользователях Abbott, но при этом обращается только к одному сегменту.

А как насчет глобальных запросов? Они также легко объясняются командой `explain`. Вот пример запроса по полю `filename`, которое не проиндексировано и не является частью сегментного ключа:

```
> db.spreadsheets.find({filename: "sheet-1"}).explain()
{
    "shards" : {
        "shard-a/arete:30000,arete:30002,arete:30001" : [
            {
                "cursor" : "BasicCursor",
                "nscanned" : 102446,
                "n" : 117,
                "millis" : 85,
            }
        ],
```

[11] В этом и последующих планах выполнения некоторые поля для краткости опущены.

```
      "shard-b/arete:30100,arete:30101" : [
         {
            "cursor" : "BasicCursor",
            "nscanned" : 77754,
            "nscannedObjects" : 77754,
            "millis" : 65,
         }
      ]
   },
   "n" : 2900,
   "nscanned" : 180200,
   "millisTotal" : 150,
   "numQueries" : 2,
   "numShards" : 2
}
```

Как и следовало ожидать, этот глобальный запрос требует полного сканирования коллекций на обоих сегментах. Если бы такой запрос встретился в реальном приложении, то нужно было бы построить индекс по полю `filename`. Но в любом случае для возврата полного результата придется произвести поиск во всем кластере.

Для некоторых запросов требуется выбирать результирующий набор параллельно. Предположим, например, что нужно отсортировать электронные таблицы по времени обновления. Для этого необходимо объединить результаты внутри процесса маршрутизации `mongos`. Без индекса такой запрос был бы очень неэффективен и мог бы даже выполняться недопустимо долго. Поэтому в следующем примере, где запрашиваются последние созданные документы, мы сначала построим нужный индекс:

```
> db.spreadsheets.ensureIndex({updated_at: 1})
> db.spreadsheets.find({}).sort({updated_at: 1}).explain()
{
   "shards" : {
      "shard-a/arete:30000,arete:30002" : [
         {
            "cursor" : "BtreeCursor updated_at_1",
            "nscanned" : 102446,
            "n" : 102446,
            "millis" : 191,
         }
      ],
      "shard-b/arete:30100,arete:30101" : [
         {
            "cursor" : "BtreeCursor updated_at_1",
            "nscanned" : 77754,
            "n" : 77754,
            "millis" : 130,
         }
```

```
      ]
   },
   "n" : 180200,
   "nscanned" : 180200,
   "millisTotal" : 321,
   "numQueries" : 2,
   "numShards" : 2
}
```

Как и следовало ожидать, курсор сканирует индекс `updated_at` на каждом сегменте.

На практике более вероятен запрос на получение последних документов, обновленных данным пользователем. И в этом случае мы создадим нужный индекс и только потом выполним запрос:

```
> db.spreadsheets.ensureIndex({username: 1, updated_at: -1})
> db.spreadsheets.find({username: "Wallace"}).sort(
   {updated_at: -1}).explain()
{
   "clusteredType" : "ParallelSort",
   "shards" : {
      "shard-1-test-rs/arete:30100,arete:30101" : [
         {
            "cursor" : "BtreeCursor username_1_updated_at_-1",
            "nscanned" : 801,
            "n" : 801,
            "millis" : 1,
         }
      ]
   },
   "n" : 801,
   "nscanned" : 801,
   "numQueries" : 1,
   "numShards" : 1
}
```

По поводу этого плана выполнения уместно будет сделать два замечания. Во-первых, этот запрос направляется только одному сегменту. Так как мы указали часть сегментного ключа, то маршрутизатор может определить, в каком сегменте находится соответствующая порция. Следует ясно понимать, что факт наличия сортировки еще не означает, что необходимо обращаться ко всем сегментам; если запрос с сортировкой включает сегментный ключ, то множество опрашиваемых сегментов часто удается сократить. В данном случае опрашивается только один сегмент, но можно представить себе аналогичные запросы, для выполнения которых нужно обратиться к нескольким, но не ко всем сегментам.

Во-вторых, из плана, показанного `explain`, видно, что при выполнении запроса к сегменту используется индекс `{username: 1, updated_at: -1}`. Это иллюстрирует важный момент, касающийся обработки запросов в сегментированном кластере. Сегментный ключ применяется для маршрутизации запроса к данному сегменту, но после того, как запрос дошел до сегмента, тот сам решает, с помощью какого индекса его обслужить. Имейте это в виду, когда будете проектировать запросы и индексы для своих приложений.

9.3.2. Индексирование

Выше мы видели примеры индексированных запросов в сегментированном кластере. Если вы не уверены, как будет обрабатываться данный запрос, поинтересуйтесь у `explain()`. Обычно никаких сложностей не возникает, но все же следует помнить еще о нескольких аспектах использования индексов при работе с сегментированным кластером.

1. На каждом сегменте поддерживаются собственные индексы. Вроде бы это очевидно, но давайте всё же уточним: когда вы объявляете индекс над сегментированной коллекцией, каждый сегмент строит отдельный индекс для своей части коллекции. Например, в предыдущем разделе мы выполняли команду `db.spreasheets.ensureIndex()` через `mongos`; в результате каждый сегмент обрабатывал команду создания индекса независимо от всех остальных.
2. Отсюда следует, что над сегментированными коллекциями в каждом сегменте должны быть построены одни и те же индексы. Если это не так, будет наблюдаться непостоянная производительность обработки запросов.
3. Над сегментированной коллекцией разрешается строить только два уникальных индекса: по полю `_id` и по сегментному ключу. Другие уникальные индексы запрещены, потому что для поддержки ограничения уникальности необходимо обмениваться данными между сегментами, а это сложно и работало бы слишком медленно.

Разобравшись, как маршрутизируются запросы и как работает индексирование, вы сможете правильнее писать запросы и выбирать индексы для сегментированного кластера. Большая часть приведенных в главе 7 рекомендаций относительно индексирования и оптими-

зации запросов применима и к кластеру, если требуется детальное исследование, то в вашем распоряжении имеется команда `explain()`.

9.4. Выбор сегментного ключа

От правильного выбора сегментного ключа зависит очень многое. Неудачный ключ не позволит приложению воспользоваться многими преимуществами сегментированного кластера. В патологических случаях производительность вставки и запросов может даже существенно снизиться. Ситуация осложняется еще и тем, что однажды выбранный сегментный ключ не подлежит изменению[12].

Поэтому, чтобы научиться работать с сегментированием, необходимо понимать, чем характеризуется хороший сегментный ключ. Так как интуитивно это не очевидно, я начну с описания того, что такое *плохой* ключ. И это естественно подведет нас к обсуждению ключей, которые решают поставленную задачу.

9.4.1. Неэффективные сегментные ключи

Для некоторых сегментных ключей характерно плохое распределение. Другие не дают возможности в полной мере воспользоваться принципом локальности. Третьи могут препятствовать расщеплению порций. Ниже мы рассмотрим различные виды неоптимальных ключей.

Плохое распределение

Идентификатор BSON-объекта по умолчанию является первичным ключом любого документа MongoDB. Казалось бы, тип данных, настолько глубоко интегрированный в MongoDB, должен быть неплохим кандидатом на роль сегментного ключа. Увы, это впечатление обманчиво. Напомним, что в старших битах любого идентификатора объекта хранится временная метка. И, следовательно, идентификаторы монотонно возрастают. А монотонно изменяющиеся значения совершенно непригодны в качестве сегментных ключей.

Чтобы понять, в чем тут проблема, нужно вспомнить, что сегментирование базируется на диапазонах. Если сегментный ключ монотонно возрастает, то все новые документы будут попадать в узкий не-

[12] Не существует приемлемого способа изменить созданный сегментный ключ. Лучшее, на что можно рассчитывать, – создать новую сегментированную коллекцию с другим ключом, экспортировать данные из старой коллекции, а затем загрузить их в новую.

прерывный диапазон. С точки зрения сегментирования, это означает, что все операции вставки маршрутизируются в одну-единственную порцию и, значит, в один сегмент. Тем самым практически сводится на нет одно из основных достоинств сегментирования: автоматическое распределение вставок между машинами[13]. Таким образом, если вы хотите, чтобы операции вставки равномерно распределялись между разными сегментами, сегментный ключ не должен монотонно возрастать. Необходимо внести элемент случайности.

Отсутствие локальности

Монотонно возрастающий сегментный ключ изменяется в одном направлении; для полностью случайного ключа вообще нельзя говорить о каком-то определенном направлении изменения. В первом случае операции вставки не распределяются вовсе, во втором – распределяются слишком хорошо. Возможно, последнее утверждение кажется противоречащим интуиции, поскольку смысл сегментирования в том и заключается, чтобы распределять операции чтения и записи. Но давайте поставим простой мысленный эксперимент.

Представьте себе, что каждый документ в сегментированной коллекции содержит MD5-свертку и что это поле является сегментным ключом. Значение свертки изменяется случайно, поэтому операции вставки гарантированно будут равномерно распределяться между всеми сегментами кластера. Это хорошо. Но подумайте, как в этом случае будет происходить вставка в *индексы* на каждом сегменте. Так как MD5-свертка – случайная величина, то при вставке с одинаковой вероятностью может производиться обращение к любой странице виртуальной памяти. На практике это означает, что индекс обязан целиком помещаться в память, а, если памяти для индекса и данных не хватает, то неизбежны страничные отказы и, следовательно, падение производительности.

Это и называется проблемой *локальности ссылок*. Идея локальности, по крайней мере в этом случае, заключается в том, что данные, к которым производятся обращения в течение некоторого промежутка времени, как правило взаимосвязаны, и эту близость можно положить в основу оптимизации. Например, хотя идентификаторы объектов не годятся на роль сегментных ключей, они прекрасно обеспечивают локальность – именно потому, что монотонно возрастают. Это означает, что последовательные вставки в индекс всегда будут

[13] Отметим, что факт монотонного возрастания сегментного ключа не отражается на операциях обновления при условии, что документы обновляются в случайном порядке.

происходить в несколько недавно использованных страниц виртуальной памяти и, значит, в каждый момент времени в ОЗУ должна будет присутствовать лишь небольшая часть индекса.

Возьмем менее абстрактный пример – пусть приложение позволяет пользователям загружать фотографии, и метаданные фотографий хранятся в документах из некоторой коллекции. Предположим, что пользователь пакетно загружает сразу 100 фотографий. Если сегментный ключ абсолютно случаен, то база данных не может извлечь преимуществ из локальности; новые записи будут вставлены в 100 мест, разбросанных по всему индексу. А теперь допустим, что в качестве сегментного ключа выбран идентификатор пользователя. В этом случае запись в индекс будет происходить примерно в одном месте, так как у всех вставляемых документов идентификатор пользователя один и тот же. Здесь мы обращаем локальность себе на пользу и потенциально сможем повысить производительность.

У случайных сегментных ключей есть еще один недостаток: сколько-нибудь осмысленный запрос по такому ключу должен быть отправлен всем сегментам. Снова вернемся к примеру коллекции фотографий. Если приложение должно показать 10 последних загруженных фотографий (в обычных обстоятельствах тривиальный запрос), то случайность сегментного ключа заставит разослать запрос всем сегментам. В следующем разделе мы увидим, что использование низкоизбирательных ключей позволяет выполнить такой запрос по диапазону на одном сегменте.

Нерасщепляемые порции

Если и случайные, и монотонно возрастающие сегментные ключи не годятся, то следующий вариант – низкоизбирательные (coarse-grained) ключи, то есть ключи, общие для многих элементов коллекции. Хорошим примером может служить идентификатор пользователя. Если коллекция фотографий сегментирована по идентификатору пользователя, то можно ожидать более-менее равномерного распределения операций вставки между сегментами – просто потому, что невозможно предсказать, когда какой пользователь будет загружать данные. Поэтому низкоизбирательный сегментный ключ обладает элементами случайности, что кластеру только на пользу.

Второе достоинство низкоизбирательного ключа в том, что он позволяет системе воспользоваться преимуществами локальности ссылок. Если пользователь вставляет 100 документов с метаданными

фотографий, то выбор идентификатора пользователя в качестве сегментного ключа гарантирует, что все вставки будут маршрутизированы одному и тому же сегменту и затронут близкие участки индекса. Очень эффективно.

Равномерное распределение и локальность – это замечательно, но низкоизбирательный сегментный ключ несет с собой трудноразрешимую проблему – потенциальную возможность неограниченного роста порции. Как такое возможно? Допустим, что мы выбрали в качестве ключа идентификатор пользователя. Каким при этом будет минимально возможный диапазон порции? Понятно, что наименьший диапазон состоит из одного идентификатора. Но тут-то и таится проблема, потому что в любом наборе данных бывают резко отклоняющиеся значения – выбросы. Допустим, что есть несколько аномальных пользователей, которые загрузили на много миллионов фотографий больше, чем средний пользователь. Сможет ли система расщепить порцию, содержащую фотографии такого пользователя? Нет, не сможет. Такая порция нерасщепляема, и это создает угрозу кластеру, так образуется дисбаланс между сегментами.

Понятно, что в идеале сегментный ключ должен сочетать достоинства низкоизбирательных и высокоизбирательных ключей. Пример мы увидим в следующем разделе.

9.4.2. Идеальные сегментные ключи

Из предыдущего обсуждения ясно, что сегментный ключ должен обладать следующими характеристиками:

1. Равномерно распределять операции вставки между сегментами.
2. Обеспечивать локальность операции CRUD.
3. Обладать достаточной избирательностью, чтобы не препятствовать расщеплению порцию.

Сегментные ключи, удовлетворяющие всем этим требованиям, обычно состоят из двух полей: первое низкоизбирательное, второе высокоизбирательное. Хорошим примером может служить сегментный ключ в примере задачи о хранении электронных таблиц. Там мы объявили составной ключ вида `{username: 1, _id: 1}`. Если в кластер вставляют данные разные пользователи, то можно ожидать, что большинство, если не все, таблицы одного пользователя окажутся в одном сегменте. Но даже если документы пользователя находятся в нескольких сегментах, то присутствие в сегментном ключе уникального поля `_id` гарантирует, что все операции чтения и обновления

одного документа всегда маршрутизируются одному сегменту. А если потребуется более сложный запрос к данным некоторого пользователя, то можно быть уверенным, что он будет направлен только сегментам, содержащим данные этого пользователя.

Но самое главное то, что сегментный ключ `{username: 1, _id: 1}` гарантирует расщепляемость порций, даже если некоторый пользователь создаст аномально много документов.

Рассмотрим еще один пример. Допустим, что вы разрабатываете систему веб-аналитики. В приложении В будет показано, что для такой системы разумно хранить по одному документу на каждую страницу каждого месяца. Внутри него будут данные за каждый день месяца, и программа будет увеличивать счетчики в разных полях при каждом посещении страницы. Вот некоторые поля из документа в аналитической системе, представляющие интерес с точки зрения выбора сегментного ключа:

```
{  _id: ObjectId("4d750a90c35169d10fc8c982"),
   domain: "org.mongodb",
   url: "/downloads",
   period: "2011-12"
}
```

Простейший сегментный ключ для коллекции, содержащей такого рода документы, состоит из домена и URL-адреса страницы: `{domain: 1, url: 1}`. Все страницы из одного домена будут как правило находиться в одном сегменте, но аномальные домены, в которых страниц очень много, все же могут распределяться по нескольким сегментам.

9.5. Сегментирование в производственных системах

При развертывании сегментированного кластера в производственной системе необходимо решить ряд задач и принять несколько важных решений. В этом разделе я опишу две рекомендуемые топологии развертывания и дам ответы на типичные вопросы. Затем мы рассмотрим задачи администрирования сервера, включая мониторинг, резервное копирование, отработку отказов и восстановление.

9.5.1. Развертывание и конфигурирование

Правильно произвести развертывание и конфигурирование с первого раза бывает трудно. Ниже приведены некоторые рекомендации по организации и настройке кластера.

Топологии развертывания

Чтобы развернуть тестовый сегментированный кластер MongoDB, нам пришлось запустить в общей сложности девять процессов (по три процесса `mongod` для каждого набора реплик плюс три конфигурационных сервера). Немудрено и напугаться. Начинающий пользователь может прийти к выводу, что для кластера из двух сегментов понадобится девять отдельных машин. Но, к счастью, можно обойтись куда меньшим числом. Чтобы понять, почему это так, нужно оценить ожидаемые требования к ресурсам, потребляемым каждым компонентом кластера.

Сначала рассмотрим наборы реплик. Каждый член, принимающий участие в репликации, содержит полную копию данных своего сегмента и может исполнять роль первичного или вторичного узла. Для этих процессов обязательно наличие места на дисках, достаточного для хранения всех данных, и оперативной памяти, достаточной для эффективного обслуживания этих данных. Поэтому реплицирующие экземпляры `mongod` потребляют больше всего ресурсов и заслуживают отдельных машин.

А как насчет арбитров наборов реплик? Эти процессы хранят только конфигурационные данные о наборах реплик, занимающие всего один документ. Поэтому накладные расходы, сопряженные с арбитрами, невелики, и отдельные серверы им *не нужны*.

Теперь о конфигурационных серверах. На них хранится сравнительно немного данных. Так, для управления тестовым набором реплик требуется всего 30 КБ данных. Если предположить, что размер этих данных линейно растет с увеличением объема данных в кластере, то для кластера объемом 1 ТБ на конфигурационных серверах нужно будет хранить всего 30 МБ данных[14]. Следовательно, конфигурационным серверам отдельные машины тоже не нужны. Однако, учитывая, какую важную роль они играют, некоторые пользователи все же предпочитают размещать их на выделенных компьютерах скромной комплектации (или на виртуальных машинах).

Основываясь на том, что вы уже знаете о наборах реплик и сегментированных кластерах, можно составить следующий перечень минимальных требований к развертыванию:

1. Каждый член набора реплик, будь то полная реплика или арбитр, должен размещаться на отдельной машине.
2. Каждый реплицирующий член набора реплик нуждается в собственной машине.

[14] Это консервативная оценка. Реальное значение будет гораздо меньше.

3. Арбитры наборов реплик нетребовательны к ресурсам и могут размещаться на одной машине с каким-нибудь другим процессом.
4. Конфигурационные серверы могут размещаться на одной машине с другим процессом. Единственное непререкаемое требование заключается в том, чтобы все серверы, принадлежащие одному конфигурационному кластеру, находились на разных машинах.

Может показаться, что найти размещение, удовлетворяющее всем этим требованиям, – головоломная логическая задача. Но мы прямо сейчас приведем две разумные топологии развертывания тестового кластера из двух сегментов, которые отвечают всем требованиям. Решение показано на рис. 9.4.

Эта конфигурация удовлетворяет всем приведенным выше требованиям к развертыванию. Главными «обитателями» каждой машины являются реплицирующие узлы сегментов. Остальные процессы распределены так, чтобы все три конфигурационных сервера и все члены наборов реплик находились на разных машинах. Такая топология устойчива к отказам любой, но только одной, машины. Какая бы машина ни вышла из строя, кластер сможет продолжать выполнение

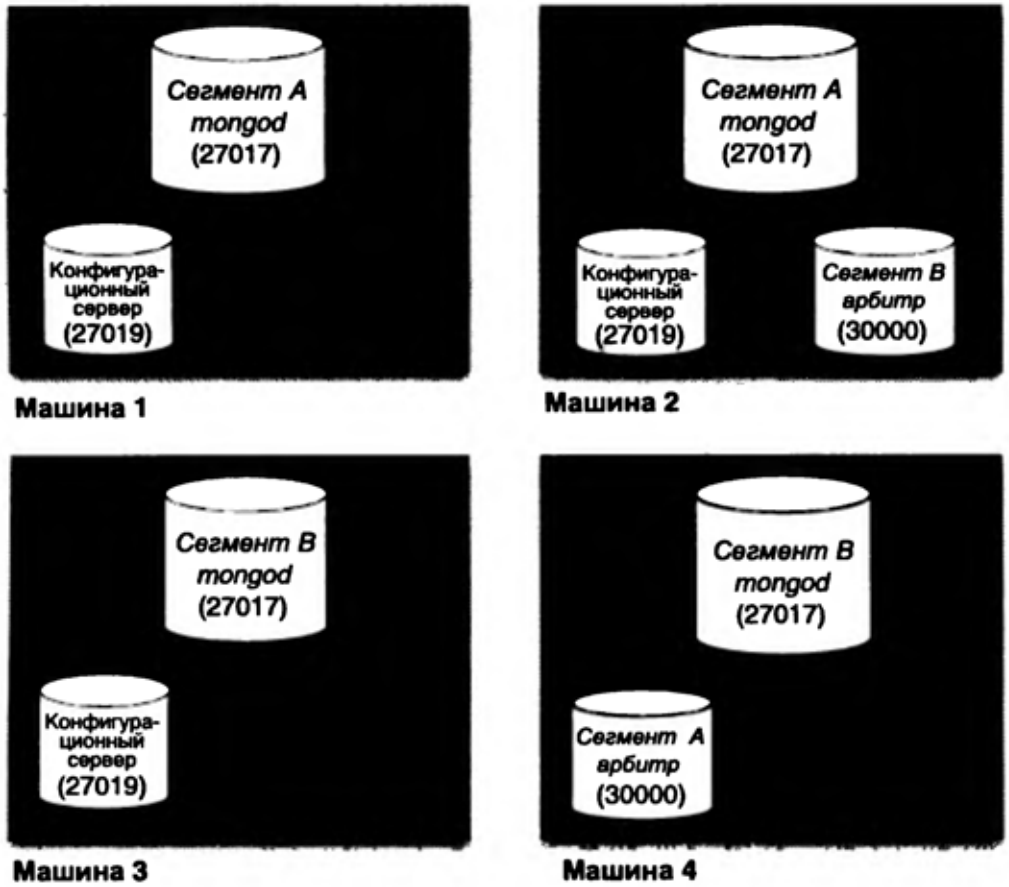


Рис. 9.4. Двухсегментный кластер, развернутый на четырех машинах

операций чтения и записи. Если откажет один из конфигурационных серверов, то расщепление и миграция будут приостановлены[15]. К счастью, приостановка операций сегментирования редко сказывается на работе кластера; расщепление и миграцию можно отложить до тех пор, пока машина не будет восстановлена.

Это *минимальная* рекомендуемая конфигурация двухсегментного кластера. Но если приложение должно быть высоко доступным и быстро восстанавливаемым, то понадобится что-то более надежное. В предыдущей главе было сказано, что набор из двух реплик и одного арбитра при восстановлении уязвим. Наличие трех узлов снижает риск сбоя во время восстановления, а кроме того позволяет разместить один узел в резервном ЦОД на случай аварийного восстановления. На рис. 9.5 показана надежная топология кластера с двумя сегментами. Каждый сегмент включает набор из трех реплик, на каждом узле которого хранится полная копия данных. На случай аварийного восстановления один конфигурационный сервер и один узел каждого сегмента вынесены в резервный ЦОД; чтобы эти узлы никогда не стали первичными, им назначен приоритет 0.

При такой конфигурации каждый сегмент реплицируется два раза вместо одного. Кроме того, в резервном ЦОД имеются все данные, необходимые для полной реконструкции сегментированного кластера в случае выхода из строя основного ЦОД.

Отказы центра обработки данных

Наиболее вероятная причина отказа ЦОД – отключение энергоснабжения. Если серверы MongoDB работают без журналирования, то выключение питания приведет к их нештатному останову с возможным повреждением файлов данных. Единственный надежный способ восстановления после такого сбоя – ремонт базы данных, а это длительный процесс, на протяжении которого приложение будет недоступно.

Чаще всего кластер находится целиком в одном ЦОД, и для большинства приложений этого вполне достаточно. Основная мера предосторожности в такой конфигурации – проследить за тем, чтобы по крайней мере один узел в каждом сегменте и один конфигурационный сервер работали с включенным режимом журналирования. Это намного ускорит восстановление после устранения аварии электроснабжения. Журналирование рассматривается в главе 10.

Но случаются и более серьезные аварии. Бывает, что электроснабжение отключается на несколько дней. Наводнения, землетрясения и прочие природные катастрофы могут привести к физическому разрушению ЦОД. Если необходимо, чтобы и в таких случаях можно было быстро произвести восстановление, то сегментированный кластер нужно размещать в нескольких ЦОД.

[15] Чтобы операции сегментирования могли выполняться, должны быть доступны все три конфигурационных сервера.

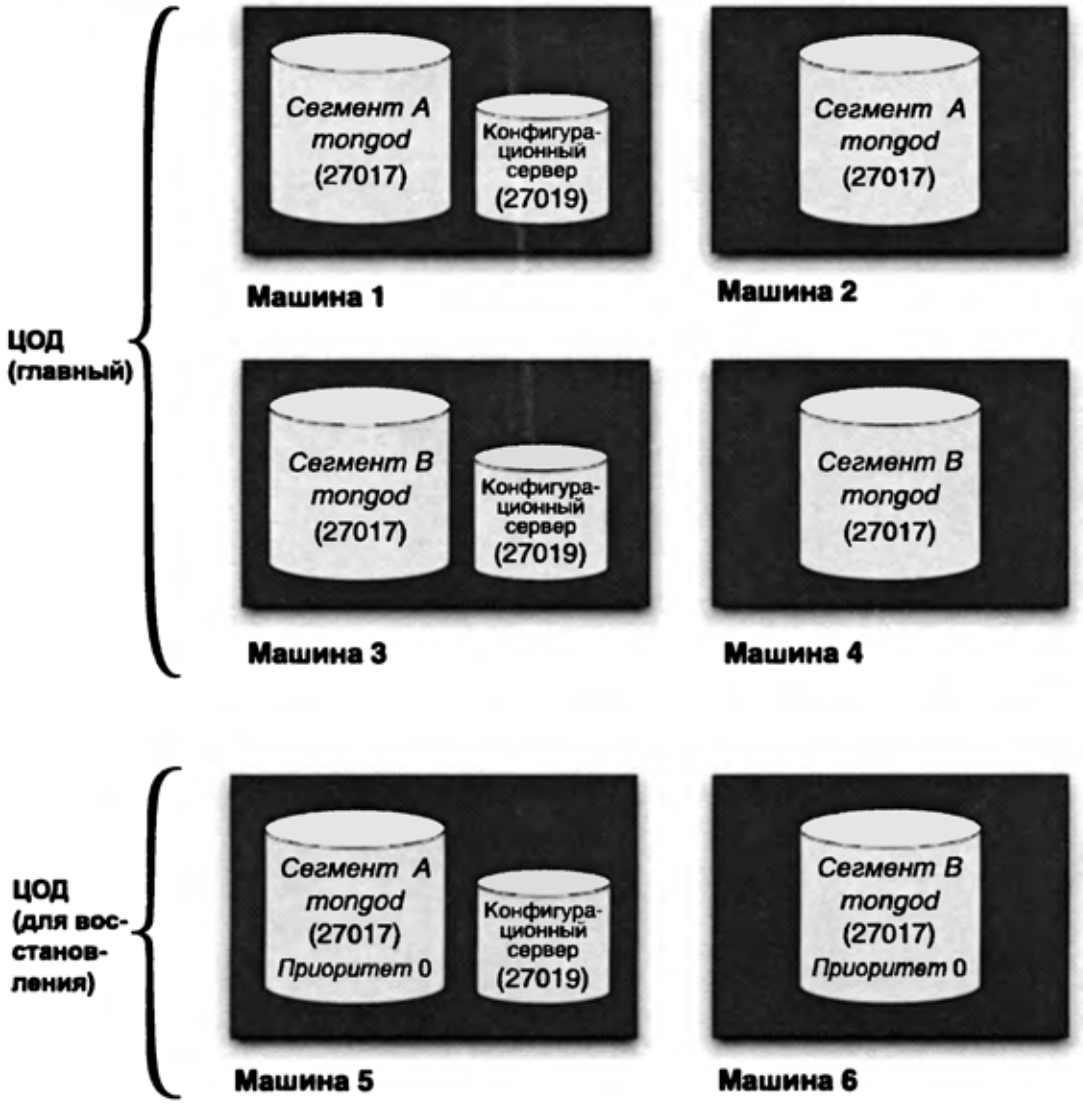


Рис. 9.5. Двухсегментный кластер, развернутый на шести машинах в двух центрах обработки данных

Решение о том, какую топологию кластера выбрать, должно приниматься с учетом допустимого времени простоя, измеряемом в терминах среднего времени до восстановления (MTR) в вашем центре. Продумайте потенциальные сценарии отказов и смоделируйте их. Рассмотрите, какие последствия для вашего приложения (и бизнеса) повлечет выход ЦОД из строя.

Замечания о конфигурировании

Ниже приведены некоторые полезные замечания о конфигурировании сегментированного кластера.

Оценка размера кластера

Пользователи часто хотят знать, сколько развертывать сегментов и каким должен быть размер каждого сегмента. Разумеется, ответ за-

висит от конкретных условий. Если вы развертываете кластер в облаке Amazon EC2, то не нужно ничего сегментировать, пока не исчерпаны возможности самого большого из предлагаемых экземпляров. На момент написания этой книги максимальный размер узла EC2 составлял 68 ГБ ОЗУ. Если вы работаете на собственном оборудовании, то это далеко не предел. Но было бы неразумно доводить объем ОЗУ до 100 ГБ прежде, чем переходить к сегментированию.

Естественно, каждый дополнительный сегмент увеличивает сложность решения, и для каждого сегмента нужны также наборы реплик. Поэтому лучше иметь немного больших сегментов, чем много маленьких.

Сегментирование существующей коллекции

Существующие коллекции можно сегментировать, но не удивляйтесь, что для распределения данных между сегментами необходимо время. За один раз можно провести только один раунд балансирования, а скорость миграции составляет 100 – 200 МБ в минуту. Таким образом, на сегментирование коллекции размером 50 ГБ уйдет около восьми часов, и при этом будет наблюдаться умеренная активность диска. Кроме того, при начальном сегментировании такой большой коллекции для ускорения процесса, возможно, придется производить расщепление порций вручную, поскольку в нормальном режиме расщепление инициируется операцией вставки.

Таким образом, понятно, что сегментирование коллекции в последнюю минуту – не лучшее решение проблемы низкого быстродействия. Если вы планируете сегментировать коллекцию в будущем, то должны делать это задолго до прогнозируемого снижения производительности.

Предварительное расщепление порций в предвидении высокой нагрузки

Если имеется большой набор данных, который нужно загрузить в сегментированную коллекцию, и есть априорная информация о распределении данных, то можно сэкономить немало времени, организовав расщепление и миграцию порций заранее. Представим, к примеру, что вы собираетесь импортировать данные из электронных таблиц в только что созданный сегментированный кластер MongoDB. Чтобы данные гарантированно были распределены равномерно, можно сначала расщепить, а затем мигрировать порции на разные сегменты. Для этого предназначены команды `split` и `moveChunk`. В оболочке для них определены вспомогательные методы `sh.splitAt()` и `sh.moveChunks()` соответственно.

Ниже приведен пример ручного расщепления порции. При вызове команды `split` указывается требуемая коллекция и точка расщепления:

```
> sh.splitAt( "cloud-docs.spreadsheets",
{ "username" : "Chen", "_id" : ObjectId("4d6d59db1d41c8536f001453") })
```

Эта команда находит порцию, которая логически содержит документ, в котором поле `username` равно Chen, `_id` равно `ObjectId("4d6d59db1d41c8536f001453")`[16]. Затем в этой точке производится расщепление, так что в результате образуются две порции. Эту процедуру можно продолжать до тех пор, пока не получится множество порций с равномерным распределением данных. Количество порций следует выбирать так, что средний размер порции был меньше пороговой величины 64 МБ. Следовательно, если ожидаемый объем загружаемых данных равен 1 ГБ, то следует запланировать примерно 20 порций.

На втором шаге нужно сделать так, чтобы на всех сегментах было примерно одинаковое количество порций. Поскольку первоначально все порции находятся на одном сегменте, то их необходимо переместить. Для перемещения порции служит команда `moveChunk` и соответствующий вспомогательный метод:

```
> sh.moveChunk("cloud-docs.spreadsheets", {username: "Chen"}, "shardB")
```

Здесь говорится, что нужно переместить на сегмент B порцию, которая логически содержала бы документ `{username: "Chen"}`.

9.5.2. Администрирование

В заключение главы хотелось бы сказать несколько слов об администрировании сегментированного кластера.

Мониторинг

Сегментированный кластер – это сложное образование, поэтому за его состоянием нужно внимательно следить. Для любого процесса `mongos` можно запустить команды `serverStatus` и `currentOp()`, которые выдают сводную статистику по всем сегментам. В следующей главе мы рассмотрим их более подробно.

Помимо получения сводной статистики по серверам, нужно следить за распределением и размерами порций. На примере тестового кластера мы видели, что вся эта информация хранится в базе данных `config`. Если в какой-то момент обнаружатся несбалансированные

[16] Такого документа может и не быть. Это с очевидностью следует из того факта, что в данном случае мы расщепляем пустую коллекцию.

порции или неконтролируемый рост порций, то можно воспользоваться командами `split` и `movechunk`. Или посмотреть в журналах, не остановилась ли по какой-то причине операция балансирования.

Ручное сегментирование

Есть два случая, когда имеет смысл вручную расщеплять и мигрировать порции на работающем кластере. Например, в версии MongoDB 2.0 балансировщик не учитывает нагрузку на отдельные сегменты. Очевидно, что чем больше данных записывается на сегмент, тем больше становятся находящиеся на нем порции и тем больше вероятность, что рано или поздно они мигрируют. Тем не менее, нетрудно представить ситуацию, в которой было бы желательно снизить нагрузку на сегмент, переместив с него часть порций. В этом снова поможет команда `movechunk`.

Добавление сегмента

Если вы пришли к выводу, что емкость необходимо увеличить, то можно добавить новый сегмент в существующий кластер, воспользовавшись тем же способом, что и выше:

```
sh.addShard("shard-c/rs1.example.net:27017,rs2.example.net:27017")
```

Но расширяя емкость таким образом, нужно ясно понимать, сколько времени займет миграция данных на новый сегмент. Выше уже отмечалось, что приблизительная скорость миграции составляет 100 – 200 МБ в минуту. Стало быть, увеличивать емкость кластера следует задолго до того, как снижение производительности станет серьезной проблемой. Решить, когда пора добавлять новый сегмент, поможет оценка скорости роста набора данных. Разумеется, желательно, чтобы индексы и рабочий набор помещались в оперативной памяти. Поэтому добавление сегмента стоит планировать по меньшей мере за несколько недель до того, как суммарный размер индексов и рабочего набора на каждом из существующих сегментов составит 90% ОЗУ.

Не подстраховавшись заранее, вы потом будете горько сожалеть. Когда индексы и рабочий набор перестанут помещаться в память, приложение начнет сильно «тормозить», особенно если для нормальной работы ему требуется высокая пропускная способность по чтению и записи. Проблема в том, что база данных будет выгружать на диск и снова загружать с диска страницы, ставя в очередь операции чтения-записи, которые не успевает выполнить. В этот момент расширить емкость кластера трудно, потому что миграция порций

только увеличивает нагрузку на существующие сегменты. Понятно, что когда база данных уже перегружена, дополнительное увеличение нагрузки – последнее, что следует делать.

Цель всего вышесказанного – утвердить вас в мысли о необходимости мониторинга кластера и заблаговременного расширения его емкости.

Удаление сегмента

Необходимость удалить сегмент, хотя и редко, но все же возникает. Для этого предназначена команда `removeshard`:

```
> use admin
> db.runCommand({removeshard: "shard-1/arete:30100,arete:30101"})
{
   "msg" : "draining started successfully",
   "state" : "started",
   "shard" : "shard-1-test-rs",
   "ok" : 1 }
```

Она сообщает, что начата эвакуация порции с указанного сегмента на другие. Чтобы узнать о ходе процедуры эвакуации, выполните команду еще раз:

```
> db.runCommand({removeshard: "shard-1/arete:30100,arete:30101"})
{
   "msg" : "draining ongoing",
   "state" : "ongoing",
   "remaining" : {
      "chunks" : 376,
      "dbs" : 3
   },
   "ok" : 1 }
```

По завершении эвакуации нужно проверить, что удаляемый сегмент не является первичным узлом какой-либо базы данных. Узнать о принадлежности сегментов можно, опросив коллекцию `config.databases`:

```
> use config
> db.databases.find()
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "cloud-docs", "partitioned" : true, "primary" : "shardA" }
{ "_id" : "test", "partitioned" : false, "primary" : "shardB" }
```

Здесь мы видим, что базой данных `cloud-docs` владеет машина `shardB`, а базой данных `test` – машина `shardA`. Поскольку мы собираемся удалить `shardB`, то нужно изменить первичный узел базы данных. Для этого служит команда `moveprimary`:

```
> db.runCommand({moveprimary: "test", to: "shard-0-test-rs" });
```

Ее следует выполнить для каждой базы данных, первичный узел которой является удаляемым сегментом. И наконец еще раз выполните команду `removeshard`, чтобы убедиться, что сегмент полностью эвакуирован:

```
> db.runCommand({removeshard: "shard-1/arete:30100,arete:30101"})
{  "msg": "remove shard completed successfully",
   "stage": "completed",
   "host": "arete:30100",
   "ok" : 1
}
```

После того как данных на сегменте не останется, можно отключать машину от сети.

Воссоединение коллекции

Удалить сегмент можно, но официального способа отменить сегментирование коллекции не существует. Если это все же необходимо, то проще всего выгрузить коллекцию целиком, а затем загрузить ее в новую коллекцию с другим именем[17]. Затем можно удалить сегментированную коллекцию. Пусть, например, коллекция `foo` сегментирована. Для ее выгрузки нужно подключиться к процессу `mongos` с помощью команды `mongodump`:

```
$ mongodump -h arete --port 40000 -d cloud-docs -c foo
connected to: arete:40000
DATABASE: cloud-docs to dump/cloud-docs
cloud-docs.foo to dump/cloud-docs/foo.bson
100 objects
```

В результате коллекция будет выгружена в файл `foo.bson`. Для ее восстановления нужно выполнить команду `mongorestore`:

```
$ mongorestore -h arete --port 40000 -d cloud-docs -c bar
Tue Mar 22 12:06:12 dump/cloud-docs/foo.bson
Tue Mar 22 12:06:12 going into namespace [cloud-docs.bar]
Tue Mar 22 12:06:12 100 objects found
```

Перенеся данные в несегментированную коллекцию, старую коллекцию `foo` можно удалить.

Резервное копирование сегментированного кластера

В резервную копию сегментированного кластера должны входить копия конфигурационных данных и копия данных с каждого сегмен-

[17] Утилиты выгрузки и восстановления, `mongodump` и `mongorestore`, рассматриваются в следующей главе.

та. Получить эти данные можно двумя способами. Первый – воспользоваться утилитой mongodump. Сначала следует выгрузить данные с одного из конфигурационных серверов, а затем с каждого кластера. Альтернативно можно выполнить mongodump через маршрутизатор mongos, то есть сразу выгрузить всю сегментированную коллекцию и базу данных config. Основной недостаток этого подхода в том, что общий объем данных на сегментированном кластере может оказаться слишком велик для выгрузки на одну машину.

Еще один употребительный способ резервного копирования кластера состоит в копировании файлов данных с одного какого-нибудь члена каждого сегмента и с конфигурационного сервера. Такая процедура для отдельного процесса mongod и набора реплик описана в следующей главе. Нужно лишь применить ее к каждому сегменту и одному конфигурационному серверу. Но при любом подходе необходимо гарантировать, что во время копирования системы порции не будут перемещаться. Следовательно, придется остановить процесс балансировщика.

Останов балансировщика

В текущей версии для останова балансировщика нужно вставить или обновить документ в коллекции settings базы данных config:

```
> use config
> db.settings.update({_id: "balancer"}, {$set: {stopped: true}}, true);
```

Но будьте осторожны: после обновления config балансировщик еще может работать. Прежде чем приступать к резервному копированию кластера, нужно удостовериться, что балансировщик завершил последний раунд балансирования. Проще всего это сделать, посмотрев, есть ли в коллекции locks запись, содержащая идентификатор _id балансировщика, и убедиться, что его состояние равно 0. Вот пример запроса к этой коллекции:

```
> use config
> db.locks.find({_id: "balancer"})
{  "_id" : "balancer", "process" : "arete:40000:1299516887:1804289383",
   "state" : 1,
   "ts" : ObjectId("4d890d30bd9f205b29eda79e"),
   "when" : ISODate("2011-03-22T20:57:20.249Z"),
   "who" : "arete:40000:1299516887:1804289383:Balancer:846930886",
   "why" : "doing balance round"
}
```

Любое состояние, большее 0, означает, что балансирование продолжается. Поле process содержит имя и порт сервера, на котором

работает процесс mongos, инициировавший данный раунд балансирования. В данном случае это сервер arete:40000. Если балансирование не остановилось после модификации коллекции settings, то следует посмотреть, нет ли сообщений об ошибках в журнале ответственного процесса mongos.

Убедившись, что балансировщик остановлен, можно начинать резервное копирование. Когда оно закончится, не забудьте снова запустить балансировщик. Для этого нужно сбросить значение поля stopped:

```
> use config
> db.settings.update({_id: "balancer"}, {$set: {stopped: false}}, true);
```

Для упрощения этих операций в версию MongoDB 2.0 включены два вспомогательных метода оболочки. Например, запустить и остановить балансировщик позволяет метод sh.setBalancerState():

```
> sh.setBalancerState(false)
```

Он эквивалентен изменению значения поля stopped в коллекции settings. Деактивировав балансировщик, можно проверять, остановился ли он, повторно вызывая метод sh.isBalancerRunning().

Отработка отказов и восстановление

Выше мы рассмотрели отказы набора реплик в общем случае, но важно также обратить особое внимание на потенциальные точки отказа сегментированного кластера и дать рекомендации по восстановлению.

Отказ члена сегментированного кластера

Любой сегмент состоит из набора реплик. Следовательно, если какой-то член набора выходит из строя, то произойдут выборы нового первичного узла, и процесс mongos автоматически подключится к нему. В главе 8 были описаны шаги восстановления отказавшего члена набора реплик. Какой метод выбрать, зависит от того, что именно случилось с членом, но рекомендации не зависят от того, является набор реплик частью сегментированного кластера или нет.

Если наблюдается аномальное поведение после отработки отказа набора реплик, то систему можно привести в исходное состояние, перезапустив все процессы mongos. В результате будут правильно установлены соединения с новыми наборами реплик. Кроме того, заметив, что балансирование не работает, вы должны поискать в коллекции locks базы данных config записи, в которых поле process указывает на бывшие первичные узлы. Если такая запись обнаружится, то соответствующий документ о блокировке неактуален, и его можно без опаски удалить вручную.

Отказ конфигурационного сервера

Для нормальной работы сегментированного кластера необходимы три конфигурационных сервера, но допустимы отказы не более двух из них. Если число работающих конфигурационных серверов меньше трех, то они перейдут в режим чтения, то есть все операции расщепления и балансирования будут приостановлены. Негативного воздействия на кластер в целом это не оказывает. Операции чтения и записи в кластер по-прежнему работают, и после восстановления всех трех конфигурационных серверов балансировщик продолжит работу с того места, где остановился.

Чтобы восстановить конфигурационный сервер, просто скопируйте файлы данных с одного из работающих серверов на отказавший. Затем перезапустите сервер[18].

Отказ mongos

Отказ процесса `mongos` – не повод для беспокойства. Если `mongos` запущен на сервере приложений, то вместе с `mongos`, скорее всего, «упал» и сам этот сервер. В этом случае все сводится к восстановлению сервера.

По какой бы причине `mongos` ни отказал, собственного состояния у него всё равно нет. Следовательно, для восстановления `mongos` достаточно перезапустить процесс, сказав ему, где находятся конфигурационные серверы.

9.6. Резюме

Сегментирование – эффективная стратегия обеспечения высокой производительности чтения и записи больших наборов данных. Эта методика применяется во многих производственных системах на основе MongoDB, поможет и вам. Вместо того чтобы «на коленке» реализовывать собственную схему сегментирования, вы можете воспользоваться плодами усилий, вложенных в разработку механизма сегментирования в MongoDB. Если вы решите последовать советам, данным в этой главе, обращайте особое внимание на рекомендованные топологии развертывания, стратегии выбора сегментного ключа и следите, чтобы данные помещались в ОЗУ, – тогда сегментирование станет вам хорошим подспорьем.

---

[18] Как всегда, перед копированием файлов данных либо заблокируйте процесс `mongod` (как описано в главе 10), либо штатно остановите его. Никогда не копируйте файлы данных с работающего сервера.

ГЛАВА 10. Развертывание и администрирование

В этой главе:

- Вопросы развертывания и требования к оборудованию.
- Администрирование, резервное копирование и безопасность.
- Разрешение проблем, связанных с производительностью.

Эта книга была бы неполной без некоторых замечаний по поводу развертывания и администрирования. В конце концов, использовать MongoDB – это одно, а обеспечивать бесперебойную работу в производственном режиме – совсем другое. Поэтому цель этой последней главы – подготовить вас к принятию правильных решений, касающихся развертывания и администрирования MongoDB. Можете считать ее кладезем мудрости, который позволит избежать печалей, сопровождающих потерю производственной базы данных.

Я начну с общих вопросов развертывания: требования к оборудованию, безопасность, а также импорт и экспорт данных. Затем я вкратце расскажу о методах мониторинга MongoDB. Мы поговорим о том, что входит в понятие обслуживания базы данных, и прежде всего о резервном копировании. И закончим главу изложением общей стратегии поиска неполадок, влияющих на производительность.

10.1. Развертывание

Для успешного развертывания MongoDB необходимо правильно выбрать оборудование и топологию серверов. Если уже имеются какие-то данные, то нужно знать, как их импортировать (и экспортировать). И, наконец, необходимо обеспечить безопасность. Все эти вопросы мы рассмотрим ниже.

10.1.1. Среда развертывания

В этом разделе я поделюсь соображениями по поводу того, что считать хорошей средой развертывания для MongoDB. Я расскажу о конкретных требованиях к оборудованию – ЦП, памяти и дискам – и предложу рекомендации по оптимизации операционной системы. Кроме того, я дам несколько советов по развертыванию в облаке.

Архитектура

Уместно будет высказать два замечания по поводу архитектуры.

Во-первых, поскольку MongoDB проецирует файлы данных на виртуальное адресное пространство, то в производственной системе следует использовать только 64-разрядные машины. Выше уже отмечалось, что 32-разрядная архитектура ограничивает доступную MongoDB память 2 гигабайтами. А если еще включено журналирование, то остается всего около 1,5 ГБ. Это опасно, так как в случае превышения этих пределов поведение MongoDB становится непредсказуемым. Конечно, вы можете проводить автономное тестирование и предварительное развертывание на 32-разрядных машинах, но для работы в производственном режиме и для нагрузочного тестирования переходите на 64-разрядную архитектуру.

Далее, MongoDB работает только на компьютерах с «остроконечным» (little-endian) порядком следования байтов. Обычно удовлетворить этому требованию нетрудно, но владельцев систем на базе процессоров SPARC, PowerPC, PA-RISC и других «тупоконечников» сразу следует исключить[1]. Большинство драйверов поддерживают оба порядка байтов, так что клиенты MongoDB как правило способны работать на компьютерах с любой архитектурой.

Процессор

MongoDB не слишком требовательна к процессору; операции базы данных редко ориентированы на вычисления. Ваша главная цель при

[1] О поддержке «тупоконечных» архитектур в сервере см. https://jira.mongodb.org/browse/SERVER-1625.

оптимизации MongoDB – позаботиться о том, чтобы не столкнуться с ограничениями ввода/вывода (см. следующие два раздела, посвященные ОЗУ и дискам).

Но если диски и рабочий набор целиком помещаются в память, то может наблюдаться повышенное потребление ресурсов процессора. Если имеется единственный экземпляр MongoDB, обслуживающий десятки (или сотни) тысяч запросов в секунду, то для повышения производительности можно увеличить количество процессорных ядер. При выполнении операций чтения, в которых не используется JavaScript, MongoDB в состояния задействовать все имеющиеся ядра.

Если вы заметите, что при выполнении операций чтения процессора загружен на 100 процентов, проверьте, нет ли в журналах сообщений о медленных запросах. Быть может, недостает необходимых индексов, что приводит к полному сканированию коллекций. Если у приложения много клиентов и каждый из них сканирует коллекции, то в сочетании с контекстными переключениями это может стать причиной повышенного потребления ресурсов ЦП. Решение состоит в том, чтобы добавить необходимые индексы.

При выполнении операций записи MongoDB в каждый момент времени использует только одно ядро, что обусловлено наличием глобальной блокировки записи. Следовательно, единственный способ масштабировать нагрузку по записи – снизить потребление ресурсов подсистемы ввода/вывода, а это означает горизонтальное масштабирование за счет сегментирования. В версии MongoDB 2.0 проблема несколько смягчена, поскольку операция записи не удерживает блокировку на все время обработки страничного отказа, а уступает процессор другой операции. В перспективе рассматривается также реализация блокировки на уровне коллекций и на уровне экстентов. О продвижении дел в этой области читайте обсуждения на сайте JIRA и примечания к последним выпускам.

ОЗУ

Как и любая СУБД, MongoDB работает тем быстрее, чем больше памяти. Поэтому выбирайте оборудование (физическое или виртуальное) с достаточным объемом ОЗУ, чтобы в память поместились часто используемые индексы плюс рабочий набор данных. А по мере накопления данных следите за отношением объема ОЗУ к размеру рабочего набора. Когда рабочий набор перестанет помещаться в памяти, производительность начнет резко падать. Сам по себе странич-

ный обмен с диском – не проблема, так как это необходимое условие загрузки данных в память. Проблема возникает тогда, когда загрузка и выгрузка страниц происходит слишком часто. В главе 7 подробно обсуждаются взаимоотношения между размерами индексов, рабочего набора и объема памяти. А в конце этой главы я покажу, как узнать о недостатке памяти.

Существует несколько сценариев, при которых превышение размера данных над объемом доступной памяти не составляет проблемы, но это скорее исключения, а не правило. Один такой пример – использование MongoDB в качестве архива, когда данные читаются и записываются редко, и потому быстрый ответ необязателен. В таком случае оснащение компьютера памятью, сравнимой по объему с данными, может оказаться слишком дорогостоящим предприятием, а выгода его иллюзорна, так как приложение все равно не использует всю память. Как бы то ни было, решение всегда следует основывать на результатах тестирования. Протестируйте приложение на репрезентативном наборе данных, чтобы определить эталонные показатели производительности.

Диски

При выборе дисков следует учитывать IOPS (число операций ввода/вывода в секунду) и время подвода головки. Между работой с одним стандартным жестким диском, работой в облаке с виртуальным диском (скажем, в EBS) и работой с высокопроизводительной сетью хранения данных (SAN) существуют разительные отличия. Некоторые приложения демонстрируют приемлемую производительность при наличии одного сетевого тома EBS, но для особо требовательных необходимо нечто большее.

Быстродействие дисков важно по нескольким причинам. Во-первых, при записи в базу MongoDB сервер по умолчанию производит синхронизацию с диском каждые 60 секунд. Это называется *фоновым сбросом*. Если нагрузка по записи высока, а диск медленный, то фоновый сброс может негативно сказаться на производительности системы в целом. Во-вторых, высокопроизводительный диск обеспечивает гораздо более быстрый прогрев сервера. После перезапуска сервера необходимо загрузить набор данных в память. Это происходит постепенно; при каждой операции чтения или записи MongoDB загружает очередную виртуальную страницу, пока вся физическая память не заполнится. Чем быстрее диск, тем быстрее происходит эта процедура, и, следовательно, повышается производительность после холодного

перезапуска. Наконец, быстрый диск способен изменить требования к отношению размера рабочего набора к объему ОЗУ. Например, при использовании твердотельного накопителя можно обойтись гораздо меньшим объемом памяти (или позволить себе больший размер данных), чем при работе с жестким диском.

Независимо от типа используемых дисков в серьезных вычислительных центрах обычно устанавливают не один диск, а массив дисков с избыточностью (RAID). Для управления RAID-массивом в кластере чаще всего применяется менеджер логических томов, входящий в состав Linux (LVM), и конфигурируется уровень RAID 10, который, с одной стороны, обеспечивает избыточность, а с другой – приемлемую производительность[2].

Если данные хранятся в нескольких базах на одном сервере MongoDB, то обеспечить наращивание емкости можно путем запуска сервера с флагом `--directoryperdb`. При этом для каждой базы создается отдельный подкаталог в общем каталоге данных. Это теоретически позволяет смонтировать каждую базу на отдельный том (автономный или в составе RAID-массива) и заодно несколько повысить пропускную способность, так как можно будет читать данные с разных жестких (или твердотельных) дисков.

Файловые системы

Чтобы MongoDB демонстрировала оптимальную производительность, необходимо правильно выбрать файловую систему. Особенно хороши для быстрого выделения непрерывных областей диска файловые системы `ext4` и `xfs`. Их использование ускорит частые операции предварительного выделения места на диске, выполняемые MongoDB.

Помимо монтирования быстрой файловой системы, повышения производительности можно добиться еще одним способом: отключив фиксацию времени последнего доступа к файлам (`atime`). Обычно операционная система обновляет атрибут `atime` при каждом чтении или записи в файл. Отключить эту функцию в Linux несложно. Сначала сделайте резервную копию конфигурационного файла файловой системы. Затем откройте исходный файл в каком-нибудь текстовом редакторе:

```
sudo mv /etc/fstab /etc/fstab.bak
sudo vim /etc/fstab
```

[2] Обзор уровней RAID см. на странице http://en.wikipedia.org/wiki/Standard_RAID_levels.

В этом файле для каждого тома имеется отдельная строка параметров, разбитая на столбцы. В столбце `options` добавьте директиву `noatime`:

```
# file-system mount type options dump pass
UUID=8309beda-bf62-43 /ssd ext4 noatime 0 2
```

Сохраните файл. Новые параметры вступают в силу немедленно.

Файловые дескрипторы

В некоторых дистрибутивах Linux количество открытых файловых дескрипторов ограничено 1024. Для MongoDB это слишком мало и может привести к ошибкам при открытии соединений (о чем недвусмысленно свидетельствуют сообщения в журнале). Оно и понятно – ведь MongoDB нуждается в файловом дескрипторе для каждого открытого файла и сетевого соединения. В предположении, что файлы данных хранятся в каталоге, имя которого содержит слово *data*, узнать, сколько сейчас открыто дескрипторов, можно с помощью утилиты `lsof` и правильно подобранной последовательности фильтров:

```
lsof | grep mongo | grep data | wc -l
```

Подсчет сетевых соединений производится аналогично:

```
lsof | grep mongo | grep TCP | wc -l
```

Поэтому будет разумно с самого начала увеличить максимальное число открытых файловых дескрипторов, чтобы не испытывать проблем во время эксплуатации. Временно увеличить верхний предел позволяет утилита `ulimit`:

```
ulimit -Hn
```

Чтобы сделать изменение постоянным, откройте в редакторе файл `limits.conf`:

```
sudo vim /etc/security/limits.conf
```

Теперь измените жесткое и мягкое ограничение. То и другое задается для каждого пользователя. В примере ниже предполагается, что процесс `mongod` запускается от имени пользователя `mongodb`:

```
mongod hard nofile 2048
mongod hard nofile 10240
```

Новые значения вступят в силу после очередного входа пользователя в систему.

Отсчет времени

Репликация чувствительна к *рассинхронизации часов*. Если время на различных узлах набора реплик отличается, то репликация будет работать неправильно. Это печально, но, к счастью, есть решение. Нужно только включить на всех серверах протоков *NTP* (*Network Time Protocol*), тогда их часы будут синхронизироваться.

Во всех вариантах Unix это достигается запуском демона `ntpd`. В Windows ту же функция выполняется служба времени (Windows Time Services).

Облако

Все больше становится пользователей, которые запускают MongoDB в виртуализированных средах, получивших общее название *облако*. Самой популярной из таких сред является Amazon EC2 – благодаря простоте использования, присутствию в разных географических регионах и конкурентоспособному ценообразованию. EC2 и другие подобные системы вполне можно использовать для развертывания MongoDB, но не следует забывать и об их недостатках, особенно если приложение использует MongoDB на пределе возможностей.

Первая проблема заключается в том, что EC2 предлагает на выбор ограниченный набор типов экземпляров. На момент написания этой книги максимальный объем памяти составлял 68 ГБ. Следовательно, вы будете вынуждены сегментировать базу данных, когда размер рабочего набора превысит 68 ГБ, а не для всякого приложения это приемлемо. При развертывании на физическом оборудовании объем памяти можно наращивать и дальше, что повлияет на решение о переходе к сегментированному кластеру.

Другая проблема – то, что EC2 по существу представляет собой черный ящик. Если обслуживание внезапно прервется или экземпляр начнет «тормозить», то ни диагностировать причину, ни устранить ее вы не сможете.

Третья проблема касается системы хранения данных. EC2 позволяет монтировать виртуальные блочные устройства, которые называются *EBS-томами*. Они обладают большой гибкостью, позволяя наращивать емкость системы хранения и перемещать тома между машинами. EBS также умеет делать мгновенные снимки, пригодные в качестве резервных копий. Беда в том, что EBS-тома не обеспечивают высокой пропускной способности, особенно если сравнивать их с физическими дисками. Поэтому пользователи, развернувшие серьез-

ные приложения в облаке EC2, часто совмещают EBS с RAID 10 для повышения пропускной способности чтения. Для высокопроизводительных приложений это настоятельная необходимость.

В свете вышесказанного многие пользователи предпочитают не бороться с ограничениями и непредсказуемостью EC2, а развертывать MongoDB на собственном физическом оборудовании. Однако подчеркнем, что EC2 и вообще облачные вычисления удобны и полностью отвечают потребностям различных категорий пользователей. Важно лишь тщательно протестировать работу приложения в облаке, прежде чем принимать решение о развертывании в нем.

10.1.2. Конфигурирование сервера

Определившись со средой развертывания, нужно принять общие решения о конфигурации сервера: выбрать топологию развертывания и решить, будет ли использоваться журналирование.

Выбор топологии

Рекомендуется включать в набор реплик не менее трех узлов. По меньшей мере два узла набора должны использоваться для хранения данных (а не в роли арбитра) и размещаться на разных машинах. Третий член может также содержать реплику или быть арбитром, для которого специальную машину выделять необязательно; в частности, арбитр может работать на той же машине, где запущен сервер приложений. В главе 8 были представлены две практичные конфигурации набора реплик.

Если вы ожидаете, что размер рабочего набора с самого начала превысит объем ОЗУ, то имеет смысл сразу настраивать сегментированный кластер, состоящий мо меньшей мере из двух наборов реплик. Подробные рекомендации по развертыванию и запуску кластера приведены в главе 9.

Для тестирования и промежуточного развертывания достаточно одного сервера. Но развертывать на одном сервере производственную систему не рекомендуется, даже если включен режим журналирования. Наличие всего одной машины затрудняет резервное копирование и восстановление, а случае отказа просто не окажется ресурса, на который можно было бы переключиться.

Впрочем, в редких случаях возможны исключения. Если приложению не требуется высокая доступность и гарантии быстрого восстановления, а размер набора данных сравнительно невелик (скажем, меньше 1 ГБ), то развертывать его на одном сервере допустимо. И

все же, принимая во внимание постоянно снижающиеся цены на оборудование и многочисленные достоинства репликации, трудно найти убедительные аргументы в пользу отказа от второй машины.

Журналирование

Режим журналирования появился в версии MongoDB 1.8, а в MongoDB 2.0 он включен по умолчанию. В этом режиме все операции записи сначала заносятся в журнал, а только потом в основные файлы данных. Это позволяет серверу быстро и надежно восстановиться после нештатного останова.

До версии 1.8 этой возможности не было, поэтому нештатный останов часто приводил к катастрофе. В чем причина? Я уже неоднократно говорил, что MongoDB проецирует файлы данных на виртуальную память. Это означает, что любая запись производится по адресу в виртуальной памяти, а не напрямую на диск. Ядро ОС периодически синхронизирует память с диском, но частота и полнота этих действий не детерминированы. Поэтому MongoDB принудительно синхронизирует все файлы данных один раз в 60 секунд с помощью системного вызова `fsync`. Проблема в том, что если остановить процесс MongoDB в момент, когда имеются не сброшенные на диск операции записи, то невозможно сказать, в каком состояния окажутся файлы данных. Не исключено, что они будут повреждены.

В случае нештатного останова процесса `mongod`, работающего без журналирования, для приведения файловых данных в согласованное состояние требуется операция ремонта. В процессе ремонта файлы данных перезаписываются и все, что невозможно интерпретировать (поврежденные данные), отбрасывается. Поскольку длительный простой и потеря данных никому не нравятся, то к ремонту прибегают лишь в крайнем случае, когда никакие другие средства восстановления не помогают. Ресинхронизация с существующей репликой почти всегда проще и надежнее. Такой способ восстановления – один из аргументов в пользу развертывания с репликацией.

Журналирование избавляет от необходимости ремонта базы данных, потому что MongoDB может привести данные в согласованное состояние с помощью журнала. В MongoDB 2.0 режим журналирования по умолчанию включен, но его можно отключить, запустив сервер с флагом `--nojournal`:

```
$ mongod --nojournal
```

Если этот режим включен, то файлы журнала находятся в подкаталоге `journal` главного каталога данных.

При запуске сервера MongoDB в режиме журналирования нужно помнить о двух моментах. Во-первых, журналирование негативно сказывается на производительности. Если требуется обеспечить максимальное быстродействие, не отказываясь от журнала, то есть два варианта действий. Первый – включать журналирование только на пассивных репликах. При условии, что реплика успевает за первичным узлом, производительность не теряется. Второе, в какой-то степени дополняющее решение, состоит в том, чтобы отвести под журнал отдельный диск, поставив символическую ссылку с каталога журнала на вспомогательный том. Этот том может быть небольшим – диска емкостью 120 ГБ более чем достаточно, а твердотельные накопители такого размера стоят сравнительно недорого. Монтирование отдельного твердотельного диска под журнал позволяет снизить потерю производительности до минимума.

Второй момент состоит в том, что журналирование само по себе *не* гарантирует сохранение всех без исключения операций записи. Гарантируется лишь, что MongoDB сможет восстановить непротиворечивое состояние файлов данных. В режиме журналирования буфер записи сбрасывается на диск каждые 100 мс. Поэтому нештатный останов может привести к потере операций записи, произведенных в последние 100 мс. Если в каких-то частях приложения это неприемлемо, то можно задавать параметр `j` в команде `getlasterror`, – тогда сервер дождется синхронизации журнала и только потом вернет управление:

```
db.runCommand({getlasterror: 1, j: true})
```

На уровне приложения ожидание задается как один из параметров безопасного режима (наряду с `w` и `wtimeout`). В Ruby параметр `j` указывается следующим образом:

```
@collection.insert(doc, :safe => {:j => true})
```

Но имейте в виду, что задавать его при каждой операции записи неразумно, так как результатов придется ждать до следующей синхронизации журнала. Иными словами, каждая операция записи может занимать до 100 мс. Так что применяйте с осторожностью[3].

10.1.3. Импорт и экспорт данных

Если вы переносите на MongoDB существующую систему или хотите загрузить в базу информацию из какого-нибудь хранилища данных, то

[3] В будущих версиях MongoDB разработчики обещают более точный контроль над синхронизацией журнала. Детали читайте в примечаниях к выпускам.

понадобится эффективный метод импорта. Может также возникнуть необходимость в хорошей стратегии экспорта, так как иногда данные из MongoDB нужно выгружать для обработки внешними программами. Например, стало общепринято экспортировать данные в Hadoop для пакетной обработки[4].

В MongoDB существует два способа импорта и экспорта данных. Можно воспользоваться штатными утилитами `mongoimport` и `mongoexport` или написать простую программу с помощью какого-либо драйвера[5].

Утилиты *mongoimport* и *mongoexport*

В состав MongoDB входят утилиты импорта и экспорта данных: `mongoimport` и `mongoexport`. `Mongoimport` позволяет импортировать данные в форматах JSON, CSV TSV. Она часто применяется для загрузки данных из реляционных баз:

```
$ mongoimport -d stocks -c values --type csv --headerline stocks.csv
```

В данном случае мы импортируем CSV-файл `stocks.csv` в коллекцию `values` базы данных `stocks`. Флаг `--headerline` говорит, что первая строка CSV-файла содержит имена полей. Чтобы получить информацию обо всех параметрах утилиты `mongoimport`, запустите ее с флагом `--help`.

Для экспорта всех данных из коллекции в формате JSON или CSV служит утилита `mongoexport`:

```
$ mongoexport -d stocks -c values -o stocks.csv
```

Показанная выше команда экспортирует данные в файл `stocks.csv`. Полную справку о параметрах также позволяет получить флаг `--help`.

Скрипты для специализированного импорта и экспорта

Штатные утилиты импорта и экспорта чаще всего используются, когда данные относительно плоские. Для документов, содержащих поддокументы и массивы, формат CSV малопригоден, так как не рассчитан на представление вложенных структур. Если возникает необходимость выгрузить или загрузить в формате CSV документ

[4] Что касается этого конкретного случая, то на странице http://github.com/mongodb/mongo-hadoop имеется официально поддерживаемый адаптер MongoDB-Hadoop.

[5] Отметим, что импорт и экспорт – не то же самое, что резервное копирование, о котором речь пойдет ниже.

со сложной структурой, то, пожалуй, проще написать специальную программу. Это позволяет сделать любой драйвер. Например, такие программы часто пишутся, когда нужно объединить данные из двух реляционных таблиц в одной коллекции.

Именно различие в способах моделирования и составляет основную сложность при переносе данных из одной системы в другую. В таких случаях драйвер становится инструментом конвертации.

10.1.4. Безопасность

В большинстве РСУБД имеется развитая подсистема защиты, позволяющая назначать детальные права отдельным пользователям и группам. Напротив, в MongoDB 2.0 поддерживается только простой механизм аутентификации на уровне всей базы данных. Поэтому так важно обеспечить безопасность машин, на которых работает MongoDB. В этом разделе мы поговорим о запуске MongoDB в безопасной среде и объясним, как работает аутентификация.

Безопасная среда

MongoDB, как и любая СУБД, должна работать в безопасной среде. В производственной системе необходимо обеспечивать защиту данных с помощью средств, встроенных в современные операционные системы. Единственная потенциальная сложность при использовании брандмауэра – понять, какие машины должны взаимодействовать между собой. К счастью, правила тут простые. Каждый узел набора реплик должен иметь возможность обратиться к любому другому узлу. Кроме того, любому клиенту базы данных должно быть разрешено подключаться к любому узлу набора реплик, который потенциально может быть ему необходим.

Частью сегментированного кластера является набор реплик. Поэтому применимы все относящиеся к наборам реплик правила; клиентом в данном случае выступает маршрутизатор `mongos`. Дополнительно:

- все сегменты должны иметь возможность взаимодействовать друг с другом;
- сегменты и маршрутизаторы `mongos` должны иметь возможность взаимодействовать с конфигурационными серверами.

С проблемой безопасности также связан вопрос об *адресе привязки*. По умолчанию сервер MongoDB прослушивает любой адрес данной машины. Но иногда требуется прослушивать только один

или несколько конкретных адресов. Для этого следует запускать процессы mongod и mongos с параметром --bind_ip, значением которого является список IP-адресов через запятую. Например, чтобы прослушивать возвратный интерфейс и внутренний IP-адрес 10.4.1.55, запустите mongod следующим образом:

```
mongod --bind_ip 127.0.0.1,10.4.1.55
```

Отметим, что данные между машинами передаются в открытом виде. Официальная поддержка SSL запланирована в версии MongoDB 2.2.

Аутентификация

Механизм аутентификации в MongoDB изначально был разработан в предположении, что сервер работает в разделяемой инфраструктуре. Он не блещет богатством функций, но достаточен для случая, когда требуется небольшая дополнительная защита. Сначала мы рассмотрим API аутентификации, а затем покажем, как использовать его в сочетании с набором реплик и сегментированием.

API аутентификации

Чтобы познакомиться с аутентификацией, создайте пользователя-администратора, для чего переключитесь на базу данных admin и вызовите метод db.addUser(). Этот метод принимает два аргумента: имя и пароль пользователя.

```
> use admin
> db.addUser("boss", "supersecret")
```

Администраторы могут заводить других пользователей и обращаться ко всем базам данных на сервере. После того как администратор создан, можно включать аутентификацию. Для этого перезапустите mongod с флагом --auth:

```
$ mongod --auth
```

Теперь получить доступ к базе данных смогут только авторизованные пользователи. Перезапустите оболочку и войдите от имени администратора с помощью метода db.auth():

```
> use admin
> db.auth("boss", "supersecret")
```

Далее можно завести пользователей для отдельных баз данных. Если пользователю нужно разрешить только чтение, передайте в последнем аргументе метода db.addUser() значение true. Ниже со-

здаются два пользователя базы данных stocks. У первого будут все права, а у второго – только право чтения.

```
> use stocks
> db.addUser("trader", "moneyfornuthin")
> db.addUser("read-only-trader", "foobar", true)
```

Теперь к базе данных stocks могут получить доступ только три пользователя: boss, trader и read-only-trader. Чтобы получить список всех пользователей, имеющих доступ к базе данных, опросите коллекцию system.users:

```
> db.system.users.find()
{  "_id" : ObjectId("4d82100a6dfa7bb906bc4df7"),
   "user" : "trader", "readOnly" : false,
   "pwd" : "e9ee53b89ef976c7d48bb3d4ea4bffc1" }
{  "_id" : ObjectId("4d8210176dfa7bb906bc4df8"),
   "user" : "read-only-trader", "readOnly" : true,
   "pwd" : "c335fd71fb5143d39698baab3fdc2b31" }
```

Если удалить пользователя из этой коллекции, то его права доступа к базе данных будут отозваны. При желании можете воспользоваться эквивалентным вспомогательным методом оболочки db.removeUser().

Явно выполнять операцию завершения сеанса не требуется, достаточно просто закрыть соединение (выйти из оболочки). Однако команда выхода все же существует – на случай, если она вам понадобится.

```
> db.runCommand({logout: 1})
```

Естественно, все, о чем мы говорили, доступно также из драйверов. Подробности см. в документации по API драйвера.

Аутентификация при доступе к набору реплик

Наборы реплик поддерживают тот же API аутентификации, который был описан выше, но для его включения необходимо выполнить два дополнительных действия. Сначала создайте файл, содержащий не менее шести символов из множества, допустимого для кодировки Base64[6]. Эта строка будет служить паролем, который каждый член набора реплик применяет для аутентификации остальных членов. Например, можете создать файл secret.txt, содержащий такую строку:

```
tOps3cr3tpa55word
```

[6] В кодировке Base64 допустимы все строчные и заглавные буквы латинского алфавита, цифры от 0 до 9 и символы + и /.

Поместите этот файл на каждый член набора реплик и задайте разрешения так, чтобы он был доступен только владельцу:

```
sudo chmod 600 /home/mongodb/secret.txt
```

И, наконец, при запуске каждого члена набора реплик укажите местоположение файла с паролем с помощью параметра `--keyFile`:

```
mongod --keyFile /home/mongodb/secret.txt
```

Теперь для набора включена аутентификация. Пользователя-администратора следует создать заранее, как в предыдущем разделе.

Аутентификация при доступе к сегменту

Аутентификация сегментов является обобщением аутентификации набора реплик. Каждый набор реплик в кластере защищается, как описано выше, – с помощью ключевого файла. Тем же паролем пользуются все конфигурационные серверы и все экземпляры `mongos`. Запустите эти процессы с параметром `--keyFile`, указывающим на файл с паролем, который будет защищать весь кластер. Тем самым вы включите механизм аутентификации на кластере.

10.2. Мониторинг и диагностика

После того как сервер MongoDB развернут, за ним необходимо наблюдать. Если производительность медленно снижается или часто возникают ошибки, то вы не должны оставаться в неведении. Для этого и предназначены средства мониторинга. Начнем с простейшего из них: протоколирования. А затем рассмотрим встроенные команды, предоставляющие полную информацию о работе сервера; на их основе построена утилита `mongostat` и веб-консоль, которые также будет описаны ниже. Еще я дам две рекомендации касательно внешних средств мониторинга. И в конце этого раздела представлю две диагностические утилиты: `bsondump` и `mongosniff`.

10.2.1. Протоколирование

Протоколирование – это первый уровень мониторинга, и потому необходимо продумать, как будут храниться журналы[7]. Обычно это не составляет проблемы, потому что MongoDB требует задавать параметр `--logpath` при запуске в фоновом режиме. Однако есть два дополнительных параметра, о которых вам следует знать. Чтобы вклю-

[7] Ни в коем случае не перенаправляйте журналы в /dev/null или на stdout.

чить подробную диагностику, запускайте процесс mongod с флагом -vvvvv (чем больше букв v, тем подробнее диагностика). Например, это удобно, когда требуется отладить какой-то код или протоколировать все запросы. Но имейте в виду, что подробная диагностика влечет за собой рост размера журналов и отчасти снижение производительности сервера.

Далее, процесс mongod можно запускать с параметром --logappend. Тогда существующий журнал будет открываться в режиме дозаписи, а не перезаписи.

Наконец, если процесс MongoDB работает в течение длительного времени, то имеет смысл написать скрипт, который будет периодически ротировать журналы. Для этой цели MongoDB предоставляет команду logrotate. Из оболочки она вызывается следующим образом:

```
> use admin
> db.runCommand({logrotate: 1})
```

Если отправить процессу сигнал SIGUSR1, то также будет вызвана команда logrotate. Вот как этот сигнал отправляется процессу с номером 12345:

```
$ kill -SIGUSR1 12345
```

10.2.2. Средства мониторинга

В этом разделе я опишу команды мониторинга и утилиты, поставляемые в комплекте с MongoDB.

Команды базы данных

Существуют три команды, сообщающие о внутреннем состоянии сервера. Именно на них основаны все приложения для мониторинга MongoDB.

serverStatus

Команда serverStatus – подлинный кладезь информации. В числе прочего она возвращает статистику страничных отказов и частоты доступа к B-дереву, количество открытых соединений и общее число операций вставки, обновления, выборки и удаления. Ниже приведен сокращенный результат работы этой команды:

```
> use admin
> db.runCommand({serverStatus: 1})
{
```

```
"host" : "ubuntu",
"version" : "1.8.0",
"process" : "mongod",
"uptime" : 246562,
"localTime" : ISODate("2011-03-13T17:01:37.189Z"),
«globalLock» : {
   «totalTime» : 246561699894,
   «lockTime» : 243,
   «ratio» : 9.855545289656455e-10,
   "currentQueue" : {
      "total" : 0,
      "readers" : 0,
      "writers" : 0
   },
},
"mem" : {
   "bits" : 64,
   "resident" : 3580,
   "virtual" : 9000,
   "mapped" : 6591
}
"ok" : 1 }
```

Раздел `globalLock` интересен потому, что в нем показано, сколько всего времени сервер работал под защитой глобальной блокировки. Если значение поля `ratio` велико, то запись представляет собой узкое место. Раздел `currentQueue` (текущая очередь) дает, пожалуй, более точное представление об узких местах. Если в очереди стоит много операций чтения или записи, то следует подумать об оптимизации.

В разделе `mem` показано, как процесс `mongod` использует память. Поле `bits` говорит, что мы имеем дело с 64-разрядной машиной, поле `resident` – это объем физической памяти, занятый MongoDB, `virtual` – количество мегабайтов, спроецированных на виртуальную память, а `mapped` описывает подмножество `virtual`, относящееся только к файлам данных. В примере выше на виртуальную память спроецировано около 6,5 ГБ файлов данных, причем 3,5 ГБ из них находятся в физической памяти. Я неоднократно подчеркивал, что в идеале рабочий набор должен целиком умещаться в оперативной памяти. Раздел `mem` дает приблизительную оценку того, насколько текущая ситуация близка к идеалу.

Выдача команды `serverStatus` изменяется и улучшается в каждой новой версии MongoDB, поэтому документировать ее на таком быстро устаревающем носителе, как эта книга, не особенно полезно. Подробную и актуальную информацию можно найти на странице http://www.mongodb.org/display/DOCS/serverStatus.

top

Команда `top` выводит счетчики операций для каждой базы данных. Если в приложении используются несколько баз данных или вы хотите знать, сколько времени в среднем занимают операции, то эта команда будет полезна. Вот пример:

```
> use admin
> db.runCommand({top: 1}) {
   "totals" : { "cloud-docs" :
   { "total" : { "time" : 194470, "count" : 20 },
   "readLock" : { "time" : 324, "count" : 12 },
   "writeLock" : { "time" : 194146, "count" : 8 },
   "queries" : { "time" : 194470, "count" : 20 },
   "getmore" : { "time" : 0, "count" : 0 } },
   "ok" : 1}
```

Как видите, здесь сервер проводит много времени под защитой блокировки записи. Имеет смысл разобраться, нельзя ли как-то ускорить операции записи.

db.currentOp()

Часто полезно знать, что MongoDB делает именно *в данный момент*. Метод `db.currentOp()` выводит эту информацию в виде списка выполняемых и ожидающих операций. Ниже приведен пример выдачи этой команды для сегментированного кластера, созданного в предыдущей главе:

```
db.currentOp()
[ {
         "opid" : "shard-1-test-rs:1232866",
         "active" : true,
         "lockType" : "read",
         "waitingForLock" : false,
         "secs_running" : 11,
         "op" : "query",
         "ns" : "docs.foo",
         "query" : {
            "$where" : "this.n > 1000"
         },
         "client_s" : "127.0.0.1:38068",
         "desc" : "conn"
} ]
```

Здесь мы видим особо медленный запрос. Он выполняется уже 11 секунд и, как и любой запрос, захватил блокировку чтения. Если эта операция вызывает опасения, то поле `client` позволит узнать о ее источнике. Увы, это сегментированный кластер, поэтому источником

является процесс mongos, о чем свидетельствует имя поля client_s. Чтобы прервать операцию, передайте ее идентификатор opid методу db.killOp():

```
db.killOp("shard-1-test-rs:1232866")
{
   "op" : "shard-1-test-rs:1233339",
   "shard" : "shard-1-test-rs",
   "shardid" : 1233339
}
```

Чтобы получить подробный перечень всех операций, выполняемых текущим сервером MongoDB, выполните такую виртуальную команду:

```
db['$cmd.sys.inprog'].find({$all: 1})
```

Утилита *mongostat*

Метод db.currentOp() показывает только операции, которые в данный момент выполняются или находятся в очереди. Точно так же, команда serverStatus дает мгновенный снимок различных системных полей и счетчиков. Но иногда требуется наблюдать за поведением системы в режиме реального времени, и тут-то на помощь приходит утилита mongostat. Она построена по образцу iostat и других подобных программ, то есть опрашивает сервер с определенной периодичностью и выводит разнообразную статистику: количество операций вставки в секунду, объем занятой физической памяти, частоту непопадания в страницы B-дерева и т. д.

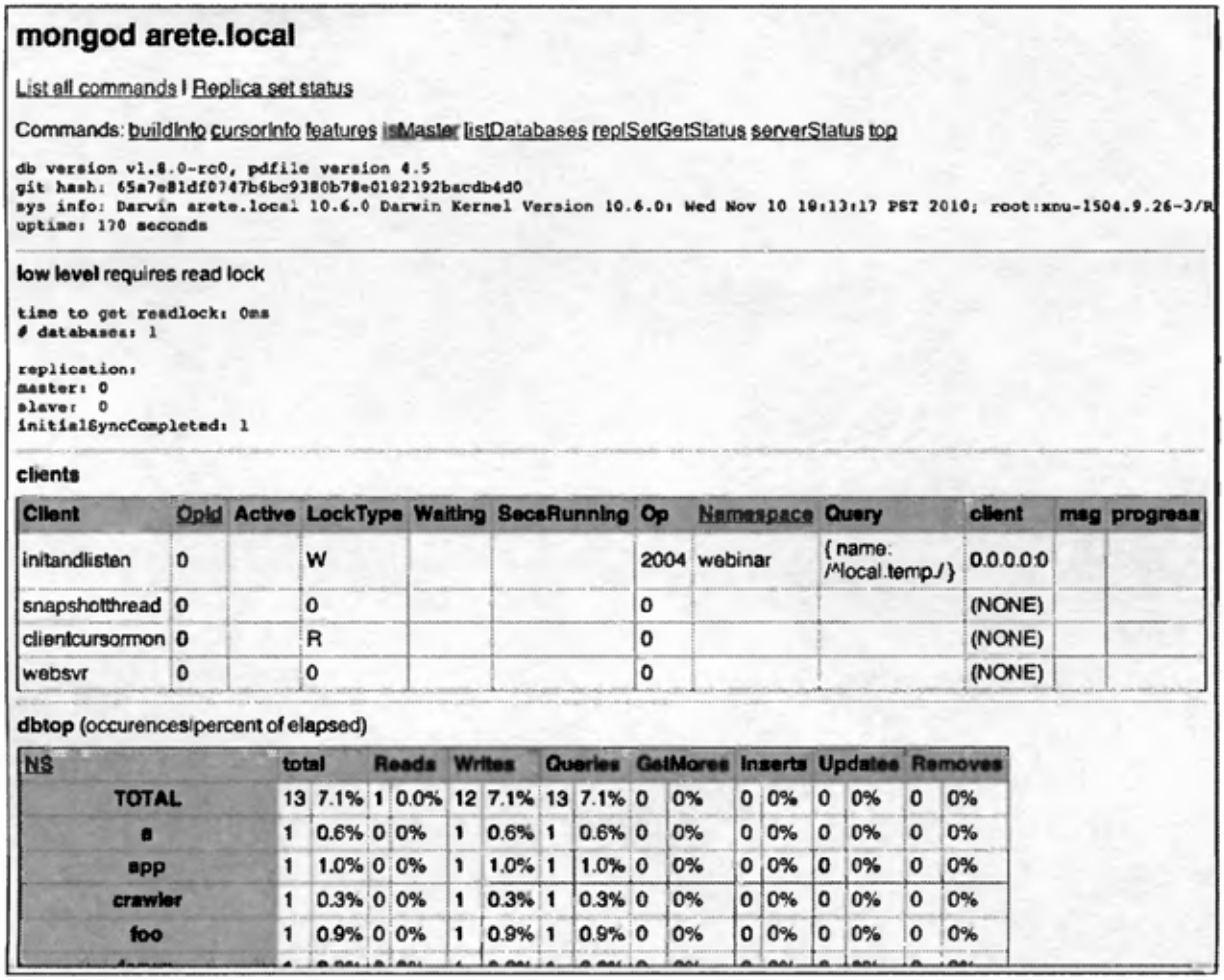
При вызове mongostat без параметров она опрашивает сервер localhost один раз в секунду:

```
$ mongostat
```

Но у этой команды множество параметров, а для получения справки нужно запустить ее с флагом --help. Одна из самых интересных возможностей – обнаружение кластера; если указать при запуске mongostat флаг --discover и какой-то один узел, то команда сама найдет все остальные узлы набора реплик или сегментированного кластера, а затем выведет сводную статистику по всему кластеру.

Веб-консоль

Веб-консоль дает чуть более наглядное представление о работе процесса mongod. Каждый экземпляр mongod прослушивает HTTP-запросы, поступающие в порт, номер которого на 1000 больше номера порта самого сервера. Иначе говоря, если mongod работает на порту



mongod arete.local

List all commands | Replica set status

Commands: buildInfo cursorInfo features isMaster listDatabases replSetGetStatus serverStatus top

```
db version v1.8.0-rc0, pdfile version 4.5
git hash: 65a7e81df0747b6bc9380b78e0102192bacdb4d0
sys info: Darwin arete.local 10.6.0 Darwin Kernel Version 10.6.0: Wed Nov 10 19:13:17 PST 2010; root:xnu-1504.9.26~3/R
uptime: 170 seconds
```

low level requires read lock

```
time to get readlock: 0ms
# databases: 1

replication:
master: 0
slave: 0
initialSyncCompleted: 1
```

clients

Client	Opid	Active	LockType	Waiting	SecsRunning	Op	Namespace	Query	client	msg	progress
initandlisten	0		W			2004	webinar	{ name: /^local.temp/ }	0.0.0.0:0		
snapshotthread	0		0			0			(NONE)		
clientcursormon	0		R			0			(NONE)		
websvr	0		0			0			(NONE)		

dbtop (occurences|percent of elapsed)

NS	total		Reads		Writes		Queries		GetMores		Inserts		Updates		Removes	
TOTAL	13	7.1%	1	0.0%	12	7.1%	13	7.1%	0	0%	0	0%	0	0%	0	0%
a	1	0.6%	0	0%	1	0.6%	1	0.6%	0	0%	0	0%	0	0%	0	0%
app	1	1.0%	0	0%	1	1.0%	1	1.0%	0	0%	0	0%	0	0%	0	0%
crawler	1	0.3%	0	0%	1	0.3%	1	0.3%	0	0%	0	0%	0	0%	0	0%
foo	1	0.9%	0	0%	1	0.9%	1	0.9%	0	0%	0	0%	0	0%	0	0%

Рис. 10.1. Веб-консоль MongoDB

27017, то веб-консоль будет доступна через порт 28017. Если браузер запущен на том же сервере, что и MongoDB, то введите в адресную строку URL http://localhost:28017 и увидите такую страницу, как на рис. 10.1.

Можно получить и более полную информацию о состоянии, активировав REST-интерфейс сервера. Если запустить `mongod` с флагом `--rest`, на начальной странице веб-консоли появятся ссылки на дополнительные команды.

10.2.3. Внешние приложения для мониторинга

В крупной системе не обойтись без внешних программ мониторинга. Часто для этой цели используются две популярные системы мониторинга с открытым кодом: Nagios и Munin. Подойдет любая из них, нужно только установить подключаемый модуль для MongoDB (тоже с открытым исходным кодом).

Но нетрудно написать подключаемый модуль для любой программы мониторинга. По существу, эта задача сводится к выполнению различных команд получения статистики от работающего сервера MongoDB. Команды `serverStatus`, `dbstats` и `collstats` обычно предоставляют всю необходимую информацию, и получить ее можно прямо через REST-интерфейс по протоколу HTTP, обойдясь без драйвера.

10.2.4. Диагностические средства (mongosniff, bsondump)

В состав MongoDB входят две диагностические утилиты. Первая, `mongosniff`, прослушивает все пакеты, передаваемые между клиентом и сервером MongoDB, и распечатывает их в удобном для восприятия виде. Если вы хотите написать новый драйвер или отладить странно ведущее себя соединение, то этот инструмент подойдет как нельзя лучше. Чтобы начать прослушивание локального сетевого интерфейса на подразумеваемом по умолчанию порту эту программу надо запустить следующим образом:

```
sudo mongosniff --source NET I0
```

После подключения любого клиента, например оболочки MongoDB, по экрану побежит поток сообщений о сетевых пакетах:

```
127.0.0.1:58022 -->> 127.0.0.1:27017 test.$cmd 61 bytes
id:89ac9c1d 2309790749 query: { isMaster: 1.0 } ntoreturn: -1
127.0.0.1:27017 <<-- 127.0.0.1:58022 87 bytes
reply n:1 cursorId: 0 { ismaster: true, ok: 1.0 }
```

Полную справку по параметрам `mongosniff` можно получить, запустив ее с флагом `--help`. Еще одна полезная утилита, `bsondump`, позволяет просматривать BSON-файлы. Такие файлы генерируются программой `mongodump` (рассматривается ниже) и в результате отката набора реплик[8]. Пусть, например, вы выгрузили коллекцию из одного документа в файл `users.bson`. Тогда просмотреть его можно следующим образом:

```
$ bsondump users.bson
{ "_id" : ObjectId( "4d82836dc3efdb9915012b91" ), "name" : "Kyle" }
```

Как видите, по умолчанию `bsondump` распечатывает BSON-данные в формате JSON. Но если вы занимаетесь серьезной отладкой, то,

[8] Существуют и другие ситуации, когда возникают BSON-файлы, но к самим файлам данных MongoDB это не относится, так что не пытайтесь просматривать их с помощью `bsondump`.

наверное, захотите увидеть реальные типы и размеры BSON-данных. Для этого программу нужно запустить в режиме отладки:

```
$ bsondump --type=debug users.bson
--- new object ---
size : 37
_id
type: 7 size: 17
name
type: 2 size: 15
```

Здесь мы видим общий размер объекта (37 байтов), типы обоих полей (7 и 2) и размеры этих полей.

10.3. Обслуживание

В этом разделе я расскажу о трех наиболее типичных задачах обслуживания MongoDB. Прежде всего, это резервное копирование. Как и в любой другой СУБД, благоразумие диктует регулярно снимать резервные копии. Затем я опишу сжатие, так как при некоторых, довольно редких, условиях может оказаться необходимым сжимать файлы данных. И, наконец, я коротко упомяну о процедуре перехода на новую версию, поскольку вам, безусловно, захочется работать с самой свежей стабильной версией MongoDB.

10.3.1. Резервное копирование и восстановление

Эксплуатируя производственную базу данных, нужно всегда быть готовым к авариям. И в этом плане резервное копирование играет важную роль. Когда гром грянет, хорошая резервная копия поможет спасти положение, и тогда вы поймете, насколько оправданы были затраты времени и усилий на проведение в жизнь политики регулярного резервного копирования. Однако до сих пор встречаются пользователи, считающие, что смогут прожить и без резервных копий. Когда окажется, что восстановить базу после аварии невозможно, винить им останется только самих себя. Не уподобляйтесь им.

Есть две общие стратегии резервного копирования базы данных MongoDB. Первая предполагает использование утилит `mongodump` и `mongorestore`. Вторая, пожалуй, более распространенная, заключается в копировании самих файлов данных.

Утилиты *mongodump* и *mongorestore*

Утилита `mongodump` выгружает содержимое базы данных в виде BSON-файлов. Утилита `mongorestore` читает и восстанавливает данные из этих файлов. Обе можно использовать для резервного копирования как отдельных коллекций и баз данных, так и всего сервера. Запускать их можно применительно к работающему серверу (его необязательно блокировать или останавливать), а можно указать некоторый набор файлов данных, но тогда сервер придется заблокировать или остановить. Простейший способ запуска `mongodump` выглядит так:

```
$ mongodump -h localhost --port 27017
```

Эта команда выгружает все базы данных вместе со всеми коллекциями с сервера, работающего на машине `localhost`, в каталог `dump`. После выгрузки в этом каталоге будут находиться все документы из всех коллекций, в том числе системных, в которых определены пользователи и индексы. Однако сами индексы в состав дампа не включаются. Это означает, что после восстановления данных индексы придется построить заново. Если набор данных очень велик или индексов много, то на это может уйти заметное время.

Для восстановления данных из BSON-файлов выполните команду `mongorestore`, указав на каталог `dump`:

```
$ mongorestore -h localhost --port 27017 dump
```

Отметим, что при восстановлении `mongorestore` по умолчанию не удаляет старые данные. Поэтому, запуская ее для существующей базы данных, не забудьте указать флаг `--drop`.

Резервное копирование файлов данных

Как правило, пользователи предпочитают копировать сами файлы данных в другое место. Этот подход часто оказывается быстрее `mongodump`, потому что ни при резервном копировании, ни при восстановлении не производятся никакие преобразования данных[9]. Единственная потенциальная проблема заключается в том, что базу данных необходимо заблокировать на время копирования, но обычно это можно сделать на вторичном узле, не прерывая работу приложения.

---

[9] И, в частности, копируются все индексы, так что перестраивать их после восстановления не придется.

Копирование файлов данных. Пользователи часто допускают ошибку, когда копируют файлы данных или делают снимок файловой системы, не заблокировав предварительно базу. Если выключено журналирование, то это приведет к повреждению скопированных файлов. При включенном журналировании делать снимок безопасно, но копировать сами файлы все равно не стоит, можно «запороть» базу.

Поэтому вне зависимости от того, включено журналирование или нет, я рекомендую всегда блокировать базу перед тем, как копировать файлы данных или делать снимок диска. Душевное спокойствие и гарантированная целостность данных стоят дороже небольшой задержки из-за блокирования.

Прежде чем копировать файлы данных, следует убедиться, что они находятся в согласованном состоянии. Поэтому либо остановите сервер, либо заблокируйте базу. Поскольку в некоторых системах останов сервера затруднен, то большинство пользователей отдают предпочтение блокированию. Вот как выглядит команда синхронизации с диском и последующего блокирования:

```
> use admin
> db.runCommand({fsync: 1, lock: true})
```

В этот момент все данные сброшены на диск и операции записи запрещены, а, значит, копировать файлы данных безопасно. Если файловая система или система хранения поддерживают снимки, то лучше сейчас сделать снимок, а скопировать файлы позже. Тогда вы сможете быстрее разблокировать базу.

Если снимок сделать нельзя, то базу придется оставить заблокированной на все время копирования файлов. Если файлы копируются на вторичном узле, то предварительно убедитесь, что вторичный узел синхронизирован с первичным и в его журнале операций достаточно места для хранения данных, которые поступят, пока файлы будут копироваться.

По завершении снятия снимка или резервного копирования базу данных можно разблокировать. Соответствующая команда выглядит следующим образом:

```
> db.$cmd.sys.unlock.findOne()
> { "ok" : 1, "info" : "unlock requested" }
```

Обратите внимание, что это лишь *запрос* на разблокировку; возможно, база не разблокируется немедленно. Убедиться, что база больше не заблокирована, можно с помощью метода `db.currentOp()`.

10.3.2. Сжатие и ремонт

В MongoDB имеется механизм ремонта базы данных. Чтобы отремонтировать все базы данных на сервере, выполните команду:

```
$ mongod --repair
```

А для ремонта одной базы данных – команду `repairDatabase`:

```
> use cloud-docs
> db.runCommand({repairDatabase: 1})
```

Во время ремонта база данных отвергает операции чтения и записи. В процессе ремонта все файлы данных читаются и записываются на новое место, причем поврежденные документы отбрасываются. Кроме того, перестраиваются все индексы. Следовательно, для ремонта базы на диске должно быть достаточно места для хранения копии всех данных. Сказать, что ремонт обходится дорого, будет сильным преуменьшением – ремонт очень большой базы данных может продолжаться несколько дней.

Механизм ремонта в MongoDB изначально задумывался как последнее средство при восстановлении поврежденной базы данных. В случае нештатного останова сервера, работающего без журналирования, ремонт – единственный способ вернуть файлы данных в согласованное состояние. Но никто не мешает вам развернуть систему с репликацией, запустить сервер хотя бы на одном узле в режиме журналирования и регулярно выполнять резервное копирование на удаленной площадке. И тогда вам не придется прибегать к ремонту для восстановления. Полагаться на ремонт как средство восстановления глупо. Не делайте этого.

Тогда для чего вообще нужен ремонт базы данных? Эта процедура сжимает файлы данных и перестраивает индексы. В версии MongoDB 2.0 сжатие данных поддерживается не очень хорошо. Поэтому после большого числа операций удаления, особенно мелких документов (меньше 4 КБ) может оказаться, что общий занимаемый файлами размер не уменьшается, а иногда даже увеличивается. Сжатие файлов позволяет избавиться от этого лишнего груза.

Если для полного ремонта нет времени или ресурсов, то существуют две возможности произвести его для отдельной коллекции: перестроить индексы или сжать коллекцию. Для перестройки индексов вызовите метод `reIndex()`:

```
> use cloud-docs
> db.spreadsheets.reIndex()
```

Возможно, это что-то даст, но, вообще говоря, место в индексе и так используется повторно, причем достаточно эффективно; основная проблема заключается в файлах данных. Поэтому лучше обратить взоры на команду `compact`, которая перезаписывает файлы данных и

перестраивает индексы для одной коллекции. Вот как она запускается из оболочки:

```
> db.runCommand({ compact: "spreadsheets" })
```

Эта команда предназначена для запуска на работающем вторичном узле, поэтому не приводит к простою системы. Закончив сжатие на всех вторичных узлах набора реплик, можете поменять первичный и вторичный узел ролями и произвести сжатие на прежнем первичном узле. Чтобы все-таки запустить команду `compact` на первичном узле, нужно при вызове добавить аргумент `{force: true}`. Но в этом случае команда заблокирует все операции записи на сервере:

```
> db.runCommand({ compact: "spreadsheets", force: true })
```

10.3.3. Модернизация

MongoDB – сравнительно молодой проект, а, значит, в новых версиях исправляются многие ошибки и вносятся улучшения в плане производительности. Поэтому по возможности следует работать с последней стабильной версией программы. До выхода версии 2.0 для перехода на новую версию было достаточно остановить старый процесс `mongod` и запустить новый со старыми файлами данных. В последующие версии MongoDB, вероятно, будут внесены небольшие изменения в форматы индексов и файлов данных, поэтому модернизация несколько усложнится. Обязательно читайте рекомендации в примечаниях к выпуску.

Разумеется, при переходе на новую версию MongoDB придется модернизировать весь реплицированный кластер. Для набора реплик общая стратегия заключается в том, чтобы модернизировать по одному узлу, начиная с вторичных.

10.4. Разрешение проблем, связанных с производительностью

В этом, последнем, разделе я хочу дать несколько основанных на опыте советов по диагностике и разрешению проблем, связанных с производительностью.

Большинство таких проблем в MongoDB возникают из-за одной и той же причины: жесткого диска. Чем больше нагрузка на диск, тем медленнее работает MongoDB. Поэтому цель почти всех усилий по оптимизации – разгрузить диск. Добиться этого можно разными спо-

собами, но перед тем как говорить о них, полезно было бы знать, как вообще оценивается производительность диска. В системах, производных от Unix, для этого предназначена утилита `iostat`. В следующем примере я запустил ее с флагом `-x`, чтобы вывести расширенную статистику, а число 2 говорит, что выводить ее нужно с интервалом 2 секунды[10]:

```
$ iostat -x 2
Device: rsec/s  wsec/s avgrq-sz avgqu-sz   await  svctm   %util
sdb      0.00  3101.12    10.09    32.83  101.39   1.34   29.36

Device: rsec/s  wsec/s avgrq-sz avgqu-sz   await  svctm   %util
sdb      0.00  2933.93     9.87    23.72  125.23   1.47   34.13
```

Подробное описание каждого поля см. на страницах руководства (`man`). Для быстрой диагностики наибольший интерес представляют последние три столбца. В столбце `await` показано среднее время обслуживания запросов ввода/вывода в миллисекундах. В него включается время ожидания в очереди и время, потраченное на выполнение запроса. В столбце `svctime` показано среднее время чистого обслуживания запросов. А в столбце `%util` – процентная доля времени, потраченного процессором на отправку запросов ввода/вывода диску.

В примере выше наблюдается умеренное использование диска. Среднее время ожидания запроса в очереди составляет около 100 мс (это много!), среднее время обслуживания – около 1 мс, а процент использования процессора – примерно 30 %. Заглянув в журналы MongoDB на этой машине, мы, скорее всего, увидели бы много медленных операций (выборки, вставки и т. д.). На самом деле, именно наличие медленных операций и должно было бы насторожить вас, а результаты `iostat` могут подтвердить, что проблема действительно имеется. Кстати, не такая уж редкость системы на основе MongoDB, в которых диск используется почти на 100 %; в таких случаях пользователи испытывают разочарование в MongoDB, хотя виновата, как правило, не только она. В следующих разделах я расскажу о способах оптимизации работы базы данных и снижения нагрузки на диск.

10.4.1. Проверка эффективности индексов и запросов

Когда производительность начинает падать, первое, на что нужно обратить внимание, – это индексы. По-видимому, приложение про-

[10] Это пример для Linux. В Mac OS X применяется команда iostat -w 2.

изводит много запросов и обновлений, поскольку именно эти операции преимущественно используют индексы[11]. В главе 7 говорилось о том, как выявлять и устранять медленные операции; для этого нужно включить профилировщик запросов и убедиться, что все операции выборки и обновления эффективно задействуют индексы. Основная цель в том, чтобы операция просматривала как можно меньше документов.

Важно также проверить, что нет лишних индексов, так как избыточный индекс занимает место на диске, потребляет память и приводит к дополнительной работе при каждой операции записи. В главе 7 рассказано о том, как избавляться от лишних индексов. А что дальше? После аудита индексов и запросов и исправления недостатков может оказаться, что проблема полностью исчезла. В журналах больше нет предупреждений о медленных запросах, и `iostat` показывает, что нагрузка на диск уменьшилась. Корректировка индексов решает проблему производительности чаще, чем вы думаете; это всегда должно быть самой первой мерой.

10.4.2. Добавление памяти

Но не всегда одного лишь изменения индексов достаточно. Даже при идеально оптимизированных запросах и прекрасно подобранных индексах использование диска может оставаться высоким. В таком случае нужно посмотреть на отношение суммарного размера индексов и рабочего набора к объему физической памяти. Для начала выполните команду `stats()` для каждой базы данных, используемой в приложении:

```
> use app
> db.stats()
{
   "db" : "app",
   "collections" : 5,
   "objects" : 3932487,
   "avgObjSize" : 543.012,
   "dataSize" : 2135390324,
   "storageSize" : 2419106304,
   "numExtents" : 38,
   "indexes" : 4,
   "indexSize" : 226608064,
   "fileSize" : 6373244928,
   "nsSizeMB" : 16,
   "ok" : 1
}
```

[11] А также некоторые другие команды, например, `count`.

Теперь посмотрите на размер данных и индексов. В данном случае размер данных чуть выше 2 ГБ, а размер индексов – около 230 МБ. В предположении, что в рабочий набор входят все данные, вам требуется 3 ГБ памяти, чтобы обращения к диску происходили не слишком часто. Если установлено только 1,5 ГБ памяти, то следует ожидать высокого использования диска.

При анализе статистики базы данных обращайте также внимание на разницу между `dataSize` и `storageSize`. Если `storageSize` превышает `dataSize` более чем в два раза, то производительность может снизиться из-за фрагментации диска. В результате может потребляться больше памяти, чем необходимо; в таком случае следует сначала сжать файлы, а только потом рассматривать вопрос о наращивании физической памяти. О том, как это делается, смотрите инструкции выше в этой главе.

10.4.3. Повышение производительности дисков

С добавлением памяти связано две проблемы. Во-первых, это не всегда возможно; например, при работе в облаке EC2 максимально доступный объем памяти в виртуальной машине составляет 68 ГБ. Во-вторых, наращивание памяти не всегда решает проблему ввода/вывода. Так, если приложение выполняет много операций записи, то фоновый сброс на диск или загрузка новых страниц данных в память все равно создают слишком большую нагрузку на диск. Поэтому, если построены эффективные индексы и памяти достаточно, а ввод/вывод тем не менее работает медленно, то имеет смысл подумать о повышении производительности дисковой подсистемы.

Решить эту задачу можно двумя способами. Первый – купить более быстрые диски. Накопитель с частотой вращения 15000 оборотов в минуту или твердотельный накопитель могут оправдать затраты. Альтернативно, или дополнительно, можно собрать из имеющихся дисков RAID-массив, который позволит увеличить пропускную способность по чтению и записи[12]. При правильной конфигурации RAID-массив способен расшить узкие места. Как уже отмечалось выше, выбор уровня RAID 10 на EBS-томах существенно повышает производительность чтения.

---

[12] У RAID есть и еще одно преимущество – при правильном выборе уровня обеспечивается избыточность.

10.4.4. Горизонтальное масштабирование

Следующая очевидная мера для решения проблемы производительности – горизонтальное масштабирование. Тут можно выбрать один из двух подходов. Если приложение в основном занято чтением, то не исключено, что один узел просто не в состоянии обслужить все посылаемые ему запросы, даже при условии, что все индексы оптимизированы и данные помещаются в оперативную память. В этом случае разумно будет распределить операции чтения по нескольким репликам. Официальные драйверы MongoDB поддерживают масштабирование на уровне членов реплик, и я рекомендую опробовать эту стратегию перед тем, как переходить к сегментированному кластеру.

Когда все прочие возможности исчерпаны, остается сегментирование. Организовывать кластер следует, когда выполняется хотя бы одно из следующих условий:

- рабочий набор не помещается в физическую память хотя бы на одной машине;
- хотя бы одна машина не справляется с нагрузкой на запись.

Если после настройки сегментированного кластера проблемы с производительностью не исчезли, то следует повторить все действия: проверить, оптимизированы ли индексы, убедиться, что данные помещаются в память, а диски работают эффективно. Чтобы в полной мере задействовать оборудование, возможно, понадобиться увеличить количество сегментов.

10.4.5. Обращение к профессионалам

Падение производительности может быть вызвано самыми разными, часто весьма специфическими, причинами – от плохо спроектированной схемы до коварных ошибок сервера. Если вы опробовали все возможные решения, а результата так и не добились, то стоит подумать о том, чтобы призвать на помощь специалистов по MongoDB. Конечно, книга способна направить на правильный путь, но с опытным экспертом она никогда не сравнится. Если вы исчерпали все идеи и терзаетесь сомнениями, обращайтесь к профессионалам. Иногда решение оказывается противоречащим интуиции.

10.5. Резюме

В этой главе мы обсудили наиболее важные вопросы развертывания MongoDB в производственной системе. Теперь вы знаете, как выби-

рать подходящее для MongoDB оборудование, как вести мониторинг работы системы и регулярно выполнять резервное копирование. Кроме того, вы получили некоторое представление о решении проблем, связанных с производительностью. Но настоящие знания приходят с опытом. Впрочем, MongoDB достаточно предсказуема и поддается оценке и наладке с помощью описанных здесь простых эвристических методов. Кроме тех случаев, когда не поддается. MongoDB стремится упростить вашу жизнь, но надо откровенно признать, что базы данных и их взаимодействие с приложениями очень сложны. Если изложенных в этой книге рекомендаций окажется недостаточно, прибегните к услугам опытного эксперта.

ПРИЛОЖЕНИЕ А. Установка

Из этого приложения вы узнаете, как установить MongoDB в Linux, Mac OS X и Windows, а также познакомитесь с наиболее употребительными параметрами настройки MongoDB. Для разработчиков я включил несколько замечаний о компилировании MongoDB из исходного кода. В конце главы есть ряд советов по установке Ruby и RubyGems; они будут полезны тем, кто захочет выполнить приведенные в книге примеры.

A.1. Установка

Перед тем как переходить к инструкциям по установке, уместно будет сделать замечание о схеме нумерации версий MongoDB. Всегда следует по возможности работать с последним стабильным выпуском для данной архитектуры. У стабильных выпусков четные номера дополнительной версии, то есть версии 1.8, 2.0 и 2.2 стабильные, а 1.9 и 2.1 находятся в разработке, поэтому в производственной системе их лучше не использовать. На странице загрузок на сайте http://mongodb.org имеются ссылки на статически скомпонованные исполняемые файлы для 32- и 64-разрядных систем. Такие файлы есть для последних стабильных выпусков, а также для ветвей разрабатываемых версий и ночных сборок последней ревизии. Скачать двоичные исполняемые файлы – самый простой способ установить MongoDB на большинство систем, в том числе Linux, Mac OS X, Windows и Solaris. Именно этому методу мы уделим основное внимание.

A.1.1. MongoDB в Linux

Есть три способа установить MongoDB в Linux: скачать откомпилированные двоичные файлы с сайта mongodb.org, воспользоваться менеджером пакетов или собрать вручную из исходного кода. В сле-

дующих разделах мы рассмотрим первые два способа, а в конце приложения я скажу пару слов о компилировании.

Установка откомпилированных двоичных файлов

Для начала перейдите на страницу http://www.mongodb.org/downloads. Там вы найдете таблицу ссылок на двоичные файлы для последних выпусков MongoDB. Выберите URL-адрес последней стабильной версии для своей архитектуры. В примерах ниже используется версия MongoDB 2.0, откомпилированная для 64-разрядной системы.

Скачайте архив с помощью браузера или утилиты `curl` и распакуйте его с помощью `tar`:

```
$ curl http://downloads.mongodb.org/linux/mongodb-linux-x86_64-2.0.0.tgz
      > mongo.tg
$ tar -xzvf mongo.tgz
```

Для работы MongoDB необходим каталог данных. По умолчанию демон `mongod` хранит файлы данных в каталоге `/data/db`. Создайте этот каталог и назначьте ему правильные разрешения:

```
$ sudo mkdir -p /data/db/
$ sudo chown `id -u` /data/db
```

Теперь все готово к запуску сервера. Перейдите в подкаталог `bin` установочного каталога MongoDB и запустите программу `mongod`:

```
cd mongodb-linux-x86_64-2.0.0/bin
./mongod
```

Если не возникнет ошибок, то на экране появится протокол запуска (ниже он приведен в сокращенном виде). Обратите внимание на последние строки, в которых говорится, что сервер прослушивает порт по умолчанию 27017:

```
Thu Mar 10 11:28:51 [initandlisten] MongoDB starting :
pid=1773 port=27017 dbpath=/data/db/ 64-bit
Thu Mar 10 11:28:51 [initandlisten] db version v2.0.0, pdfile version 4.5
...
Thu Mar 10 11:28:51 [initandlisten] waiting for connections on port 27017
Thu Mar 10 11:28:51 [websvr] web admin interface listening on port 28017
```

Если сервер неожиданно завершается, обратитесь к разделу А.5.

Установка с помощью менеджера пакетов

Менеджеры пакетов значительно упрощают установку MongoDB. Единственный их недостаток заключается в том, что составители па-

кетов часто не успевают за последними выпусками. Поскольку всегда желательно работать с последней стабильной версией, то, решив прибегнуть к менеджеру пакетов, проверяйте, какую версию вы устанавливаете.

При работе с дистрибутивами Debian, Ubuntu, CentOS или Fedora у вас всегда будет доступ к самым последним версиям, так как компания 10gen сама поддерживает и публикует свои пакеты для этих платформ. Дополнительные сведения об установке пакетов в этих системах можно найти на сайте mongodb.org. Инструкции для Debian и Ubuntu находятся на странице http://mng.bz/ZffG, а для CentOS и Fedora – на странице http://mng.bz/JSjC.

Имеются также пакеты для FreeBSD и ArchLinux. Детали см. на сайтах репозиториев пакетов.

A.1.2. MongoDB в Mac OS X

Есть три способа установить MongoDB в Mac OS X: скачать откомпилированные двоичные файлы с сайта mongodb.org, воспользоваться менеджером пакетов или собрать вручную из исходного кода. В следующих разделах мы рассмотрим первые два способа, а в конце приложения я скажу пару слов о компилировании.

Установка откомпилированных двоичных файлов

Для начала перейдите на страницу http://www.mongodb.org/downloads. Там вы найдете таблицу ссылок на двоичные файлы для последних выпусков MongoDB. Выберите URL-адрес последней стабильной версии для своей архитектуры. В примерах ниже используется версия MongoDB 2.0, откомпилированная для 64-разрядной системы.

Скачайте архив с помощью браузера или утилиты `curl` и распакуйте его с помощью `tar`:

```
$ curl http://downloads.mongodb.org/osx/mongodb-osx-x86_64-2.0.0.tgz >
        mongo.tgz
$ tar xzf mongo.tgz
```

Для работы MongoDB необходим каталог данных. По умолчанию демон `mongod` хранит файлы данных в каталоге `/data/db`. Создайте этот каталог:

```
$ mkdir -p /data/db/
```

Теперь все готово к запуску сервера. Перейдите в подкаталог bin установочного каталога MongoDB и запустите программу mongod:

```
$ cd mongodb-osx-x86_64-2.0.0/bin
$ ./mongod
```

Если не возникнет ошибок, то на экране появится протокол запуска (ниже он приведен в сокращенном виде). Обратите внимание на последние строки, в которых говорится, что сервер прослушивает порт по умолчанию 27017:

```
Thu Mar 10 11:28:51 [initandlisten] MongoDB starting :
pid=1773 port=27017 dbpath=/data/db/ 64-bit
Thu Mar 10 11:28:51 [initandlisten] db version v2.0.0, pdfile version 4.5
...
Thu Mar 10 11:28:51 [initandlisten] waiting for connections on port 27017
Thu Mar 10 11:28:51 [websvr] web admin interface listening on port 28017
```

Если сервер неожиданно завершается, обратитесь к разделу А.5.

Установка с помощью менеджера пакетов

Менеджеры пакетов значительно упрощают установку MongoDB. Единственный их недостаток заключается в том, что составители пакетов часто не успевают за последними выпусками. Поскольку всегда желательно работать с последней стабильной версией, то, решив прибегнуть к менеджеру пакетов, проверяйте, какую версию вы устанавливаете.

MacPorts (http://www.macports.org) и Homebrew (http://mxcl.github.com/homebrew/) – два менеджера пакетов для Mac OS X, которые заведомо поддерживают актуальные версии MongoDB. Для установки с помощью MacPorts выполните команду:

```
sudo port install mongodb
```

Отметим, что MacPorts собирает MongoDB и все ее зависимости с нуля. Поэтому, решив выбрать этот путь, будьте готовы к тому, что компилирование займет много времени.

Homebrew не компилирует, а просто скачивает последние двоичные файлы, поэтому установка производится гораздо быстрее, чем с помощью MacPorts. Для установки MongoDB посредством Homebrew выполните такую команду:

```
$ brew update
$ brew install mongodb
```

После установки Homebrew выведет инструкции о том, как запускать MongoDB с помощью агента пуска в Mac OS X.

A.1.3. MongoDB в Windows

Есть два способа установить MongoDB в Mac OS X. Самый простой – скачать откомпилированные двоичные файлы с сайта mongodb.org. Можно также вручную откомпилировать исходный код, но этот путь рекомендуется только для разработчиков и опытных пользователей. О компилировании исходного кода вы можете прочитать в следующем разделе.

Установка откомпилированных двоичных файлов

Для начала перейдите на страницу http://www.mongodb.org/downloads. Там вы найдете таблицу ссылок на двоичные файлы для последних выпусков MongoDB. Выберите URL-адрес последней стабильной версии для своей архитектуры. В примерах ниже используется версия MongoDB 2.0, откомпилированная для 64-разрядных версий Windows.

Скачайте подходящий дистрибутив и распакуйте его. Это можно сделать из Windows Explorer – найдите zip-архив MongoDB, щелкните по нему правой кнопкой мыши и выберите из контекстного меню пункт Extract All... (Извлечь все). После этого будет предложено выбрать папку, в которую распаковывать архив.

Вместо этого можно воспользоваться командной строкой. Перейдите в каталог Downloads (Загрузки) и вызовите утилиту `unzip` для распаковки архива:

```
C:\> cd \Users\kyle\Downloads
C:\> unzip mongodb-win32-x86_64-2.0.0.zip
```

Для работы MongoDB необходим каталог данных. По умолчанию демон `mongod` хранит файлы данных в каталоге `C:\data\db`. Откройте окно команд Windows и создайте этот каталог:

```
C:\> mkdir \data
C:\> mkdir \data\db
```

Теперь все готово к запуску сервера. Перейдите в подкаталог `bin` установочного каталога MongoDB и запустите программу `mongod`:

```
C:\> cd \Users\kyle\Downloads
C:\Users\kyle\Downloads> cd mongodb-win32-x86_64-2.0.0\bin
C:\Users\kyle\Downloads\mongodb-win32-x86_64-2.0.0\bin> mongod.exe
```

Если не возникнет ошибок, то на экране появится протокол запуска (ниже он приведен в сокращенном виде). Обратите внимание

на последние строки, в которых говорится, что сервер прослушивает порт по умолчанию 27017:

```
Thu Mar 10 11:28:51 [initandlisten] MongoDB starting :
pid=1773 port=27017 dbpath=/data/db/ 64-bit
Thu Mar 10 11:28:51 [initandlisten] db version v2.0.0, pdfile version 4.5
...
Thu Mar 10 11:28:51 [initandlisten] waiting for connections on port 27017
Thu Mar 10 11:28:51 [websvr] web admin interface listening on port 28017
```

Если сервер неожиданно завершается, обратитесь к разделу A.5.

Теперь можно запустить оболочку MongoDB. Для этого откройте второе окно команд и выполните программу `mongo.exe`:

```
C:\> cd \Users\kyle\Downloads\mongodb-win32-x86_64-2.0.0\bin
C:\Users\kyle\Downloads\mongodb-win32-x86_64-2.0.0\bin> mongo.exe
```

A.1.4. Компилирование MongoDB из исходного кода

Компилировать MongoDB из исходного кода рекомендуется только опытным пользователям и разработчикам. Если вы всего лишь хотите работать с самой последней версией, то компилировать необязательно, достаточно каждую ночь скачивать последние ревизии с сайта mongodb.org.

Тем не менее, откомпилировать исходный код самостоятельно вполне возможно. Самое сложное здесь – обеспечить необходимые зависимости, а именно библиотеки Boost, SpiderMonkey и PCRE. Актуальные инструкции по компилированию для каждой платформы имеются на странице http://www.mongodb.org/display/DOCS/Building.

A.1.5. Поиск и устранение неполадок

Процедура установки MongoDB несложна, но иногда возникают мелкие проблемы. Обычно о них свидетельствуют сообщения об ошибках, появляющиеся при попытке запуска демона `mongod`. Ниже описаны самые типичные ошибки и способы их устранения.

Выбрана не та архитектура

При попытке запустить в 32-разрядной системе файл, собранный для 64-разрядной, появится такое сообщение:

```
-bash: ./mongod: cannot execute binary file
```

В Windows 7 текст сообщения более осмысленный:

```
This version of
C:\Users\kyle\Downloads\mongodb-win32-x86_64-1.7.4\bin\mongod.exe
is not compatible with the version of Windows you're running.
Check your computer's system information to see whether you need
a x86 (32-bit) or x64 (64-bit) version of the program, and then
contact the software publisher.

Версия
C:\Users\kyle\Downloads\mongodb-win32-x86_64-1.7.4\bin\mongod.exe
несовместима с версией Windows, работающей на этом компьютере.
Проверьте сведения о системе, чтобы узнать, какая версия программы
x86 (32-разрядная) или x64 (64-разрядная) вам нужна, и получите
ее у поставщика программного обеспечения.
```

Решение в обоих случаях одно и то же: скачать и установить 32-разрядную версию. Двоичные исполняемые файлы для обеих архитектур имеются на странице загрузок на сайте MongoDB (http://www.mongodb.org/downloads).

Не существует каталог данных

MongoDB необходим каталог для хранения файлов данных. Если такого каталога нет, то будет напечатано такое сообщение об ошибке:

```
dbpath (/data/db/) does not exist, terminating
```

Для устранения ошибки нужно создать каталог. Выше, в инструкциях по установке, описано, как это делается в разных операционных системах.

Не хватает прав

При работе в ОС, производной от Unix, пользователь, от имени которого запускается `mongod`, должен иметь разрешение на запись в каталог данных. В противном случае появится сообщение:

```
Permission denied: "/data/db/mongod.lock", terminating
```

Или такое:

```
Unable to acquire lock for lockfilepath: /data/db/mongod.lock,
terminating
```

В том и в другом случае для решения проблемы нужно предоставить разрешения на работу с каталогом данных с помощью команды `chmod` или `chown`.

Невозможно привязаться к порту

По умолчанию MongoDB прослушивает порт 27017. Если другой демон `mongod` или еще какой-то процесс уже привязался к этому порту, то появится такое сообщение:

```
listen(): bind() failed errno:98
Address already in use for socket: 0.0.0.0:27017
```

Решить проблему можно двумя способами. Первый – найти процесс, который прослушивает процесс 27017 и остановить его. Второй – запустить `mongod` на другом порту, указав флаг `--port`. Ниже показано, как запустить MongoDB на порту 27018:

```
mongod --port 27018
```

A.2. Основные конфигурационные параметры

В этом разделе приведено краткое описание флагов, которые чаще всего используются при работе с MongoDB.

- `--dbpath` – путь к каталогу, в котором находятся файлы данных. По умолчанию `/data/db`.
- `--logpath` – полный путь к файлу, в который записывается журнал. По умолчанию журнал выводится на стандартный вывод (`stdout`).
- `--port` – номер порта, прослушиваемого MongoDB. Если не указан, то по умолчанию подразумевается 27017.
- `--rest` – активирует простой REST-интерфейс, расширяющий возможности веб-консоли. Веб-консоль всегда доступна на порту, номер которого на 1000 больше номера порта, прослушиваемого сервером. Следовательно, если сервер запущен на машине `localhost` и прослушивает порт 27017, то к веб-консоли можно обратиться по адресу `http://localhost:28017/`. Потратьте какое-то время на изучение веб-консоли, поскольку она может дать полезную информацию о работающем сервере MongoDB.
- `--fork` – отсоединяет процесс от терминала, оставляя его работать в режиме демона. Флаг `fork` работает только в версиях для Unix. В Windows для получения аналогичной функциональности ознакомьтесь с инструкциями по запуску MongoDB

в виде Windows-службы. Эти инструкции имеются на сайте mongodb.org.

Это наиболее важные флаги запуска MongoDB. Вот пример их использования в командной строке:

```
$ mongod --dbpath /var/local/mongodb --logpath /var/log/mongodb.log
--port 27018 --rest --fork
```

Отметим, что все эти параметры можно задать также в конфигурационном файле. Создайте текстовый файл (назовем его `mongodb.conf`) и включите в него вышеперечисленные параметры по одному в каждой строке:

```
dbpath=/var/local/mongodb
logpath=/var/log/mongodb.log
port=27018
rest=true
fork=true
```

Затем запустите `mongod`, указав конфигурационный файл с помощью флага `-f`:

```
$ mongod -f mongodb.conf
```

Чтобы узнать, с какими параметрами был запущен работающий экземпляр MongoDB, выполните команду `getCmdLineOpts`:

```
> use admin
> db.runCommand({getCmdLineOpts: 1})
```

A.3. Установка Ruby

Многие примеры в этой книге написаны на Ruby, поэтому для их выполнения необходимо установить интерпретатор Ruby и менеджер пакетов для него, RubyGems.

Следует использовать версию Ruby не ниже 1.8.7. На момент написания этой книги в производственных системах чаще всего использовались версии 1.8.7 и 1.9.3.

A.3.1. Linux и Mac OS X

В Mac OS X и многих дистрибутивах Linux Ruby установлен по умолчанию. Узнать номер версии можно с помощью команды

```
ruby -v
```

Если команда не найдена или установлена версия с номером меньше 1.8.7, то нужно будет установить или модернизировать Ruby.

Подробные инструкции по установке Ruby в Mac OS X и в различных вариантах Unix имеются на странице по адресу http://www.ruby-lang.org/en/downloads/ (возможно, придется прокрутить ее вниз). Многие менеджеры пакетов (в частности, MacPorts и Aptitude) также поддерживают последнюю версию Ruby, и вполне возможно, что именно этот путь установки окажется самым простым.

Помимо собственно интерпретатора Ruby, вам понадобится менеджер пакетов RubyGems, с помощью которого вы сможете установить драйвер MongoDB для Ruby. Узнать, установлена ли программа RubyGems, позволит команда `gem`:

```
gem -v
```

Установить RubyGems можно с помощью системного менеджера пакетов, но большинство пользователей предпочитает скачать последнюю версию и воспользоваться включенным в нее установщиком. Соответствующие инструкции можно найти на странице https://rubygems.org/pages/download.

A.3.2. Windows

Простейший способ установить Ruby и RubyGems в Windows – воспользоваться программой Windows Ruby Installer, которую можно скачать со страницы http://rubyinstaller.org/downloads. Эта программа представляет собой мастер, который проведет вас по всем шагам установки Ruby и RubyGems.

Помимо Ruby, можете установить также комплект программ Ruby DevKit, который упрощает компилирование расширений Ruby на языке C. Библиотека BSON, используемая в драйвере MongoDB для Ruby, может задействовать эти расширения (хотя способна работать и без них).

ПРИЛОЖЕНИЕ В. Паттерны проектирования

В.1. Паттерны

В начальных главах этой книги я неявно агитировал за использование некоторых паттернов проектирования. В этом приложении я коротко опишу их и еще несколько паттернов, не нашедших места в основном тексте.

В.1.1. Вложение или ссылка

Допустим, вы разрабатываете простое приложение MongoDB для ведения блога, которое позволяет хранить статьи и комментарии к ним. Как представить такие данные? Следует ли вкладывать комментарии в документы, описывающие статьи? Или лучше создать две коллекции – одну для статей, другую для комментариев – и связать комментарии со статьями с помощью ссылки на идентификатор объекта?

Это проблема выбора между вложением и ссылкой, которая часто смущает начинающих пользователей MongoDB. К счастью, существует простое эвристическое правило, помогающее при проектировании большинства схем. Применяйте вложение, если дочерние объекты всегда встречаются в контексте своего родителя. В противном случае, храните дочерние объекты в отдельной коллекции.

Что это означает применительно к статьям и комментариям в блоге? Зависит от приложения. Если комментарии отображаются только в составе статьи и никак не упорядочиваются (по дате, рейтингу и так далее), то лучше применить вложение. Но если, к примеру, вы хотите отображать последние комментарии вне зависимости от того, к какой статье они относятся, то необходимы ссылки. Вложение может дать небольшой выигрыш в производительности, но механизм ссылок гораздо более гибкий.

B.1.2. Связь один-ко-многим

В предыдущем разделе отмечалось, что связь один-ко-многим можно представить вложением или ссылкой. Вложение следует применять, когда объект со стороны *многие* естественно принадлежит своему родителю и редко изменяется. Хорошим примером может служить схема для приложения, которое выводит инструкции. В каждом руководстве шаги представлены в виде массива поддокументов, потому что они являются неотъемлемыми частями руководства и изменяются редко:

```
{  title:   "Как сварить яйцо всмятку",
           steps: [
           {  desc: "Налить в кастрюльку воду и довести до кипения.",
              materials: ["вода", "яйца"] },
           {  desc: "Осторожно опустить яйца и варить четыре минуты.",
              materials: ["яйцо таймер"]},
           {  desc: "Охладить яйца в проточной воде." },
           ]
}
```

Когда две связанные сущности могут появляться в приложении независимо, их лучше связывать ссылками. Во многих статьях по MongoDB высказывается мысль, что вложение комментариев в статьи блога – здравая идея. Однако ссылки обеспечивают гораздо большую гибкость. Во-первых, легко вывести список пользователей с комментариями каждого. Кроме того, можно показать недавние комментарии безотносительно к статье. Обе эти функции, считающиеся на большинстве сайтов в порядке вещей, было бы невозможно реализовать, если бы комментарии вкладывались[1]. Обычно документы связываются по идентификатору объекта. Вот как может выглядеть документ со статьей:

```
{  _id: ObjectId("4d650d4cf32639266022018d"),
   title: "Выращивание трав",
   text: "Травы нуждаются в периодическом поливе..."
}
```

А вот комментарий, связанный со статьей по полю `post_id`:

```
{  _id: ObjectId("4d650d4cf32639266022ac01"),
   post_id: ObjectId("4d650d4cf32639266022018d"),
   username: "zjones",
   text: "Да уж, базилик - ядреная травка!"
}
```

[1] Внесен запрос на реализацию *виртуальных коллекций*, которые могли бы соединить лучшее из обоих миров. Подробности см. на странице http://jira.mongodb.org/browse/SERVER-142.

Статьи и комментарии хранятся в разных коллекциях, для отображения статьи вместе с комментариями к ней необходимо два запроса. Поскольку комментарии запрашиваются по полю `post_id`, то понадобится индекс:

```
db.comments.ensureIndex({post_id: 1})
```

В главах 4, 5 и 6 мы часто применяли паттерн один-ко-многим; там вы найдете дополнительные примеры.

В.1.3. Связь многие-ко-многим

В реляционных СУБД для представления связи многие-ко-многим используются связующие таблицы; в MongoDB для этой цели служат массивы ключей. Пример такой техники встречался ранее при связывании товаров с категориями. В каждом документе о товаре хранится массив идентификаторов категорий, а сами товары и категории хранятся в разных коллекциях. Если есть такие два документа с описанием категорий:

```
{  _id: ObjectId("4d6574baa6b804ea563c132a"),
   title: "Эпифиты"
}
{  _id: ObjectId("4d6574baa6b804ea563c459d"),
   title: "Оранжерейные цветы"
}
```

то товар, принадлежащий той и другой, можно представить так:

```
{  _id: ObjectId("4d6574baa6b804ea563ca982"),
   name: "Ооpхидея Дракон",
   category_ids: [ ObjectId("4d6574baa6b804ea563c132a"),
   ObjectId("4d6574baa6b804ea563c459d") ]
}
```

Для эффективного поиска следует проиндексировать по массиву идентификаторов категорий:

```
db.products.ensureIndex({category_ids: 1})
```

Теперь, чтобы найти все товары в категории «Эпифиты», достаточно указать значение поля `category_id`:

```
db.products.find({category_id: ObjectId("4d6574baa6b804ea563c132a")})
```

А чтобы вернуть все категории, в которые входит орхидея Дракон, нужно сначала получить список идентификаторов категорий для товара:

```
product = db.products.findOne({_id: ObjectId("4d6574baa6b804ea563c132a")})
```

а затем опросить коллекцию categories с помощью оператора $in:

```
db.categories.find({_id: {$in: product['category_ids']}})
```

Как видите, для поиска категорий понадобилось два запроса, а для поиска товара – только один. Таким образом мы решили оптимизировать наиболее вероятный случай, поскольку поиск товаров, принадлежащих категории, – более частая операция, чем поиск категорий, включающих товар.

B.1.4. Деревья

Как и в большинстве РСУБД, в MongoDB нет встроенных средств для представления и обхода деревьев. Поэтому, если требуется такое поведение, то придется придумать собственное решение. Я предложил подход к представлению иерархии категорий в главах 5 и 6. Идея заключалась в том, чтобы хранить в каждом документе с описанием категорий копии текущих предков этой категории. При такой денормализации усложняется обновление, зато намного упрощается выборка.

К сожалению, денормализация предков не всегда является оптимальным решением. Другой распространенный пример использования деревьев – сетевой форум, в котором в одной ветке зачастую бывают сотни сообщений с большим уровнем вложенности. Из-за глубокой вложенности и большого объема данных хранение предков тут не подойдет. Неплохую альтернативу дает *материализованный путь*. При этом в каждом узле дерева имеется поле path, в котором хранится конкатенация идентификаторов предков этого узла. В узлах верхнего уровня поле path содержит null, так как у них нет предков. Давайте на примере посмотрим, как это работает. На рис. B.1 представлена ветвь комментариев с вопросами и ответами по истории Древней Греции.

Взгляните, как эти комментарии организованы в виде документов с материализованным путем. Сначала идет корневой комментарий, в котором path равно null:

```
{  _id: ObjectId("4d692b5d59e212384d95001"),
   depth: 0,
   path: null,
   created: ISODate("2011-02-26T17:18:01.251Z"),
   username: "plotinus",
   body: "Who was Alexander the Great's teacher?",
   thread_id: ObjectId("4d692b5d59e212384d95223a")
}
```

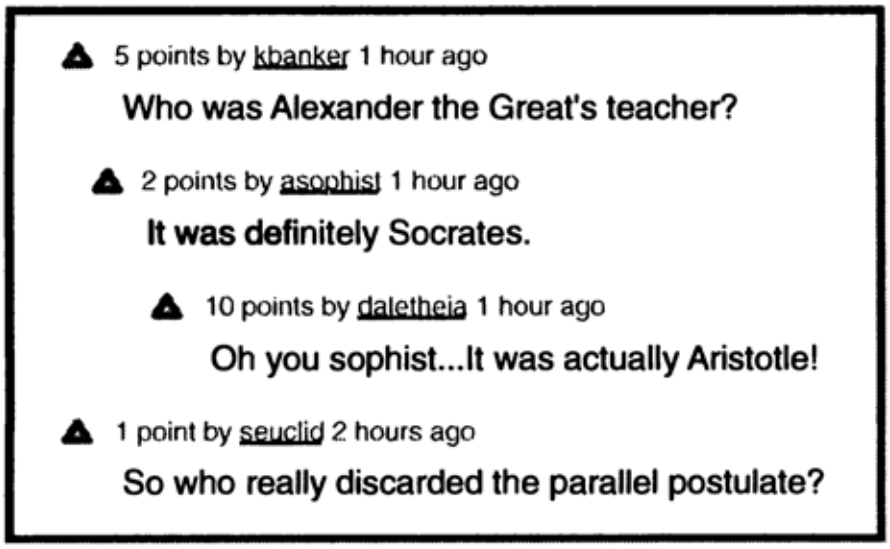


Рис. B.1. Вложенные комментарии в форуме

Второй вопрос корневого уровня, заданный пользователем `seuclid`, будет иметь такую же структуру. Интереснее, однако, ответы на вопрос, кто был учителем Александра Великого. В первом из них поле `path` содержит идентификатор `_id` непосредственного родителя:

```
{  _id: ObjectId("4d692b5d59e212384d951002"),
   depth: 1,
   path: "4d692b5d59e212384d95001",
   created: ISODate("2011-02-26T17:21:01.251Z"),
   username: "asophist",
   body: "It was definitely Socrates.",
   thread_id: ObjectId("4d692b5d59e212384d95223a")
}
```

В поле `path` ответа следующего уровня хранятся идентификаторы деда и родителя именно в этом порядке, разделенные двоеточием:

```
{  _id: ObjectId("4d692b5d59e212384d95003"),
   depth: 2,
   path: "4d692b5d59e212384d95001:4d692b5d59e212384d951002",
   created: ISODate("2011-02-26T17:21:01.251Z"),
   username: "daletheia",
   body: "Oh you sophist...It was actually Aristotle!",
   thread_id: ObjectId("4d692b5d59e212384d95223a")
}
```

Как минимум, нам понадобятся индексы по полям `thread_id` и `path`, поскольку мы всегда будем предъявлять запросы ровно по одному из этих полей:

```
db.comments.ensureIndex({thread_id: 1})
db.comments.ensureIndex({path: 1})
```

Теперь встает вопрос, как опрашивать и отображать дерево. Одно из достоинств материализованного пути заключается в том, что необходим только один запрос к базе данных вне зависимости от того, хотим мы отобразить всю ветвь комментариев или только ее часть. Запрос для получения первой ветви целиком тривиален:

```
db.comments.find({thread_id: ObjectId("4d692b5d59e212384d95223a")})
```

Запрос для получения части ветви более хитрый, так как является префиксным:

```
db.comments.find({path: /^4d692b5d59e212384d95001/})
```

В ответ мы получим все комментарии, для которых путь `path` начинается с указанной строки. Эта строка содержит `_id` комментария от пользователя `plotinus`, и, взглянув на поле `path` в каждом дочернем комментарии, легко убедиться, что все они удовлетворяют запросу. При этом префиксные запросы выполняются быстро, так как сервер может воспользоваться индексом по полю `path`.

Получить список комментариев просто, потому что требуется только один запрос к базе. А вот отобразить их сложнее, так как необходимо сохранить порядок узлов в дереве. Для этого придется написать код на стороне клиента. Его вариант на Ruby приведен ниже[2]. Первый метод, `threaded_list`, строит список всех комментариев верхнего уровня и словарь, отображающий идентификаторы родителей на списки дочерних узлов:

```
def threaded_list(cursor, opts={})
   list        = []
   child_map   = {}
   start_depth = opts[:start_depth] || 0

   cursor.each do |comment|
      if comment['depth'] == start_depth
         list.push(comment)
      else
         matches = comment['path'].match(/([d|w]+)$/)
         immediate_parent_id = matches[1]
         if immediate_parent_id
            child_map[immediate_parent_id] ||= []
            child_map[immediate_parent_id] << comment
         end
      end
   end

   assemble(list, child_map)
end
```

[2] В исходном коде, прилагаемом к книге, имеется полный пример обработки иерархии комментариев с применением материализованных путей. В него включены описанные ниже методы отображения.

Метод assemble принимает список узлов верхнего уровня и словарь дочерних узлов и строит новый список в порядке отображения:

```
def assemble(comments, map)
   list = []
   comments.each do |comment|
      list.push(comment)
      child_comments = map[comment['_id'].to_s]
      if child_comments
         list.concat(assemble(child_comments, map))
      end
   end

   list
end
```

Чтобы напечатать комментарии, нужно просто обойти этот список, формируя отступы в соответствии с полем depth каждого комментария:

```
def print_threaded_list(cursor, opts={})
   threaded_list(cursor, opts).each do |item|
      indent = " " * item['depth']
      puts indent + item['body'] + " #{item['path']}"
   end
end
```

Запрос и распечатка комментариев теперь реализуются тривиально:

```
cursor = @comments.find.sort("created")
print_threaded_list(cursor)
```

B.1.5. Очереди

Реализовать очереди в MongoDB можно с помощью стандартных или ограниченных коллекций. В обоих случаях команда findAndModify позволяет обрабатывать записи очередей атомарно.

В каждой записи очереди должны быть служебные поля state и timestamp, а также прочие – содержащие полезную нагрузку. Состояние state можно представлять строками, но целые числа более эффективны с точки зрения занимаемого места. Мы будем представлять состояния *processed* (обработана) и *unprocessed* (не обработана) числами 0 и 1 соответственно. В поле временной метки timestamp хранится дата в стандартном формате BSON. В роли полезной нагрузки будет выступать простое текстовое сообщение, но в принципе это может быть что угодно:

```
{  state: 0,
   created: ISODate("2011-02-24T16:29:36.697Z"),
   message: "hello world" }
```

Необходимо построить индекс для эффективной выборки последних необработанных записей (в порядке FIFO). Этой цели вполне отвечает составной индекс по полям `state` и `created`:

```
db.queue.ensureIndex({state: 1, created: 1})
```

Теперь с помощью команды `findAndModify` мы можем вернуть очередную запись и пометить, что она обработана:

```
q = {state: 0}
s = {created: 1}
u = {$set: {state: 1}}
db.queue.findAndModify({query: q, sort: s, update: u})
```

Если используется стандартная коллекция, то не забывайте удалять старые записи из очереди. Это можно делать по ходу обработки, указав при вызове команды `findAndModify` параметр `{remove: true}`. Но иногда требуется отложить удаление до момента, когда обработка будет полностью завершена.

В качестве основы для реализации очередей можно использовать также ограниченные коллекции. Так как над такой коллекцией не строится обычный индекс по полю `_id`, то вставка в нее производится чуть быстрее, однако для большинства приложений разница пренебрежимо мала. Еще одно потенциальное преимущество – автоматическое удаление. Однако это палка о двух концах – необходимо позаботиться о том, чтобы коллекция была достаточно велика, иначе могут оказаться затерты еще не обработанные записи. Так что при использовании ограниченной коллекции задавайте размер с запасом. Идеальный размер коллекции зависит от частоты добавления в очередь и от среднего размера полезной нагрузки.

После того как размер ограниченной коллекции определен, схема, индекс и порядок применения команды `findAndModify` остаются такими же, как для стандартной коллекции.

B.1.6. Динамические атрибуты

Модель документа в MongoDB полезна для представления сущностей с переменным набором атрибутов. Канонический пример дают товары, и ранее в этой книге были продемонстрированы некоторые способы моделирования их атрибутов. Один из возможных подходов – помещать их в отдельные поддокументы. Тогда в одной коллек-

ции products можно будет хранить товары разных типов, например наушники:

```
{  _id: ObjectId("4d669c225d3a52568ce07646")
   sku: "ebd-123"
   name: "Hi-Fi Earbuds",
   type: "Headphone",
   attrs: { color: "silver",
                   freq_low: 20,
                   freq_hi: 22000,
                   weight: 0.5
   }
}
```

и твердотельные накопители:

```
{  _id: ObjectId("4d669c225d3a52568ce07646")
   sku: "ssd-456"
   name: "Mini SSD Drive",
   type: "Hard Drive",
   attrs: { interface: "SATA",
                   capacity: 1.2 * 1024 * 1024 * 1024,
                   rotation: 7200,
                   form_factor: 2.5
   }
}
```

Если запросы по таким атрибутам предъявляются часто, то можно построить по ним разреженные индексы. Вот, например, как оптимизировать запросы по диапазону частотной характеристики наушников:

```
db.products.ensureIndex({"attrs.freq_low": 1, "attrs.freq_hi": 1},
   {sparse: true})
```

А обеспечить эффективный поиск жесткого диска с заданной частотой вращения поможет следующий индекс:

```
db.products.ensureIndex({"attrs.rotation": 1}, {sparse: true})
```

Общая стратегия такова: помещать атрибуты в отдельные поддокументы, так чтобы их было легко идентифицировать, и строить разреженные индексы, чтобы исключить пустые значения.

Если атрибуты совершенно непредсказуемы, то построить отдельный индекс для каждого невозможно. Тогда придется применить другую стратегию, которую мы продемонстрируем на примере следующего документа:

```
{  _id: ObjectId("4d669c225d3a52568ce07646")
   sku: "ebd-123"
```

```
    name: "Hi-Fi Earbuds",
    type: "Headphone",
    attrs: [ {n: "color", v: "silver"},
                    {n: "freq_low", v: 20},
                    {n: "freq_hi", v: 22000},
                    {n: "weight", v: 0.5}
                  ]
}
```

Здесь ключ `attrs` указывает на массив поддокументов, в каждом из которых хранятся два значения – `n` и `v` – определяющие имя и значение динамического атрибута. Такое нормализованное представление позволяет построить над этими атрибутами составной индекс:

```
db.products.ensureIndex({"attrs.n": 1, "attrs.v": 1})
```

Теперь атрибуты можно опрашивать, применяя оператор запроса `$elemMatch`:

```
db.products.find({attrs: {$elemMatch: {n: "color", v: "silver"}}})
```

Такой подход влечет за собой заметные издержки на хранение в индексе имен ключей. Прежде чем применять его в реальной системе, необходимо провести тщательное тестирование производительности на репрезентативном наборе данных.

B.1.7. Транзакции

MongoDB не дает гарантий ACID (атомарность, непротиворечивость, изолированность, долговечность) для последовательности операций и не располагает эквивалентами команд `BEGIN`, `COMMIT` и `ROLLBACK`, имеющихся в реляционных СУБД. Если вам это необходимо, используйте какую-нибудь другую базу данных (только для данных с транзакционной семантикой или для всего приложения). Тем не менее MongoDB поддерживает атомарное долговечное обновление на уровне отдельного документа и согласованное чтение. С помощью этих функций, пусть примитивных, можно реализовать поведение, напоминающее транзакции.

Развернутый пример этой техники был представлен в главе 6 при обсуждении авторизации заказов и управления запасами. А реализацию очереди, рассмотренную в этом приложении выше, легко можно модифицировать для поддержки транзакций. В обоих случаях основой транзакционного поведения является необычайно гибкая команда `findAndModify`, применяемая для атомарного манипулирования полем `state` в одном или нескольких документах.

Стратегию реализацию транзакционной семантики во всех случаях можно назвать *компенсационной*[3]. Абстрактно компенсационный процесс работает следующим образом:

1. Атомарно изменить состояние документа.
2. Выполнить какое-то действие, возможно, включающее атомарные модификации других документов.
3. Убедиться, что система в целом (все так или иначе связанные документы) находится в согласованном состоянии. Если это так, пометить транзакцию как завершенную, в противном случае вернуть все документы в состояние, предшествующее началу транзакции.

Следует отметить, что компенсационная стратегия абсолютно необходима для протяженных и многошаговых транзакций. Примером может служить процедура авторизации, отгрузки и отмены заказа. В таких случаях даже РСУБД, полностью поддерживающая транзакционную семантику, вынуждена реализовывать подобную стратегию.

Не всегда возможно обойти некоторые требования приложения к ACID-транзакциям с участием нескольких объектов. Но за счет правильного проектирования приложения MongoDB, иногда удается реализовать транзакционную семантику, необходимую приложению.

B.1.8. Локальность и предвычисления

MongoDB часто позиционируют как базу данных для аналитических расчетов, и очень многие пользователи хранят в ней аналитические данные. Сочетание атомарного инкремента и развитой структуры документов, похоже, именно то, что надо. Например, ниже приведен документ, в котором хранится число просмотров страниц за каждый день месяца и за месяц в целом. Для краткости мы включили только итоги за первые пять дней месяца:

```
{  base: "org.mongodb", path: "/",
   total: 99234,
   days: {
      "1": 4500,
      "2": 4324,
      "3": 2700,
```

[3] По компенсационному механизму транзакций можно порекомендовать две интересные работы: оригинальную статью Garcia-Molina and Salem «Sagas» (http://mng.bz/73is) и менее формальную, но более занимательную статью Gregor Hohpe «Your Coffee Shop Doesn't Use Two-Phase Commit» (http://mng.bz/kpAq).

```
        "4": 2300,
        "5": 0
    }
}
```

Для изменения итогов за день и за месяц достаточно простого направленного обновления с применением оператора `$inc`:

```
use stats-2011
db.sites-nov.update({ base: "org.mongodb", path: "/" },
      $inc: {total: 1, "days.5": 1 });
```

Обратите внимание на имена коллекции и базы данных. В коллекции `sitesnov` хранятся данные за один месяц, а в базе данных `stats-2011` – данные за один год. Таким образом, мы обеспечиваем локальность данных приложения. Для запроса сведений о недавних посещениях всегда используется одна коллекция, размер которой невелик по сравнению со всей историей аналитических данных. Если какие-то данные больше не нужны, то достаточно удалить всю коллекцию за соответствующий период времени, а не часть документов из более крупной коллекции. Последняя операция может привести к фрагментации диска.

Здесь же применяется еще один принцип – *предвычисление*. В конце каждого месяца вставляется пустой документ-шаблон с нулевыми значениями для каждого дня последующего месяца. Поэтому при увеличении счетчиков размер документа не будет изменяться, потому что новые поля не добавляются, а лишь модифицируются их значения на месте. Это важно, потому что позволяет избежать перемещения документа в другое место на диске в результате записи. Перемещение – медленная операция, которая часто приводит к фрагментации.

B.2. Антипаттерны

В MongoDB отсутствуют ограничения (constraints), что может привести к плохой организации данных. Ниже описаны некоторые проблемы, часто встречающиеся в производственных системах.

B.2.1. Непродуманное индексирование

Нередко причиной проблем с производительностью являются неиспользуемые или неэффективные индексы. Наиболее эффективный набор индексов получается на основе анализа запросов, предъявляе-

мых приложением. Рационально применяйте способы оптимизации, описанные в главе 7.

B.2.2. Смешанные типы

Следите за тем, чтобы в разных документах коллекции одним и тем же ключам соответствовали значения одного типа. Например, номер телефона всегда следует хранить одинаково – либо в виде строки, либо в виде числа, не смешивая одно с другим. Смешение типов одного ключа усложняет логику приложения и затрудняет разбор BSON-документов в некоторых строго типизированных языках.

B.2.3. Коллекции-свалки

Коллекции следует использовать для хранения сущностей одного типа; не помещайте в одну коллекции товары и пользователей. Поскольку создавать коллекции очень просто, то под каждый тип следует отводить свою коллекцию.

B.2.4. Большие документы с глубокой вложенностью

Существует два неправильных мнения о документной модели данных MongoDB. Одно гласит, что между коллекциями вообще не должно быть связей, а все взаимосвязанные данные следует помещать в один документ. Зачастую это приводит к полной неразберихе, но иногда пользователи все же пытаются идти этим путем. Второе недопонимание проистекает из слишком буквальной интерпретации слова *документ*. По мнению некоторых пользователей, документ – это единая сущность, аналогичная реальному физическому документу. В результате получаются гигантские документы, которые трудно запрашивать и обновлять, не говоря уж о том, как нелегко в них разобраться.

Подведем итог. Документы должны быть небольшими (намного меньше 100 КБ, если только в них не хранятся двоичные данные) с умеренным уровнем вложенности. Чем меньше документ, тем дешевле обходится его обновление, а, если возникнет необходимость переместить документ на другое место, то нуждающихся в перезаписи данных будет меньше. Дополнительное преимущество заключается в том, что документ остается понятным, что упрощает жизнь разработчикам, желающим разобраться в модели данных.

B.2.5. Одна коллекция на каждого пользователя

Редко имеет смысл создавать по одной коллекции на каждого пользователя. Проблема в том, что по умолчанию размер пространства имен (количество коллекций и индексов) ограничен 24 000. По достижении этого порога придется создавать новую базу данных. Кроме того, каждая коллекция со всеми своими индексами – это дополнительные накладные расходы. Поэтому описанная стратегия – не что иное как расточительство места.

B.2.6. Несегментируемые коллекции

Если ожидается, что коллекция рано или поздно потребует сегментирования, то позаботьтесь об этом заранее. Коллекция считается сегментируемой, если для нее можно определить эффективный сегментный ключ. Рекомендации по выбору сегментного ключа приведены в главе 9.

ПРИЛОЖЕНИЕ С. Двоичные данные и GridFS

Во многих приложениях для хранения изображений, миниатюр, аудио и прочих двоичных файлов используется исключительно файловая система. Но обеспечивая быстрый доступ к файлам, файловая система может стать причиной организационной неразберихи. Имейте в виду, что в большинстве файловых систем количество файлов в одном каталоге ограничено. Если файлов миллионы, то придется разработать стратегию распределения файлов по нескольким каталогам. Другая сложность – хранение метаданных. Поскольку метаданные хранятся в базе, то согласованное резервное копирование файлов вместе с их метаданными может оказаться невероятно сложным делом.

В некоторых случаях имеет смысл хранить файлы в самой базе данных, поскольку это упрощает организацию и резервное копирование. В MongoDB имеется BSON-тип `binary`, позволяющий хранить произвольные двоичные данные. Он соответствует типу *BLOB* (binary large object) в РСУБД и является основой двух вариантов хранения двоичных объектов в MongoDB.

Первый предполагает хранение каждого файла в отдельном документе, он оптимален для небольших двоичных объектов. Если требуется каталогизировать много изображений-миниатюр или MD5-сверток, то такой «однодокументный» подход здорово облегчит вашу жизнь. С другой стороны, иногда возникает необходимость хранить большие изображения или аудиофайлы. В таких случаях лучше прибегнуть к GridFS – предлагаемому MongoDB API для хранения двоичных объектов произвольного размера. В следующих двух разделах будут приведены примеры обоих подходов к хранению.

C.1. Хранение простых двоичных объектов

В формате BSON предусмотрен полноценный тип для представления двоичных данных. С его помощью можно хранить двоичные объекты прямо в документах MongoDB. Единственное ограничение – размер самого документа, который в версии MongoDB 2.0 не должен превышать 16 МБ. Поскольку такие большие документы могут истощить системные ресурсы, то для хранения двоичных объектов размером больше 1 МБ рекомендуется использовать GridFS.

Мы рассмотрим два примера, когда двоичный объект разумно хранить в отдельном документе: миниатюры изображений и MD5-свертки.

C.1.1. Хранение миниатюр

Пусть требуется хранить коллекцию миниатюр изображений. Код не вызывает затруднений. Сначала получаем имя файла изображения, `canyon-thumb.jpg`, а затем считываем данные в локальную переменную. Далее оборачиваем двоичные данные BSON-объектом типа `binary`, применяя конструктор `BSON::Binary`, предоставляемый драйвером для Ruby:

```
require 'rubygems'
require 'mongo'

image_filename = File.join(File.dirname(__FILE__), "canyon-thumb.jpg")
image_data = File.open(image_filename).read

bson_image_data = BSON::Binary.new(image_data)
```

Осталось только построить простой документ, содержащий двоичные данные, и поместить его в базу:

```
doc = {  "name" => "monument-thumb.jpg",
         "data" => bson_image_data }

@con = Mongo::Connection.new
@thumbnails = @con['images']['thumbnails']
@image_id = @thumbnails.insert(doc)
```

Чтобы извлечь двоичные данные, запросим этот документ. В Ruby метод `to_s` преобразует данные в двоичную строку, которую можно использовать для сравнения сохраненных данных с оригиналом:

```
doc = @thumbnails.find_one({"_id" => @image_id})
   if image_data == doc[„data"].to_s
   puts "Сохраненное изображение совпадает с исходным файлом!"
end
```

Выполнив этот скрипт, вы увидите сообщение, подтверждающее, что оба файла идентичны.

C.1.2. Хранение MD5-свертки

Очень часто требуется сохранить контрольную сумму в виде двоичных данных, и здесь снова приходит на помощь BSON-тип `binary`. Вот как можно сгенерировать MD5-свертку миниатюры и сохранить ее в том же документе, что и выше:

```
require 'md5'
md5 = Digest::MD5.file(image_filename).digest
bson_md5 = BSON::Binary.new(md5, BSON::Binary::SUBTYPE_MD5)

@thumbnails.update({:_id => @image_id}, {"$set" => {:md5 => bson_md5}})
```

Отметим, что при создании двоичного BSON-объекта мы пометили данные тегом `SUBTYPE_MD5`. Подтип – это дополнительное поле в BSON-типе `binary`, обозначающее вид хранимых двоичных данных. Впрочем, оно факультативно и никак не отражается на хранении или интерпретации данных базой[1].

Запросить сохраненный документ просто, но обратите внимание, что мы исключили поле `data`, чтобы возвращенный документ был небольшим и понятным:

```
> use images
> db.thumbnails.findOne({}, {data: 0})
{
   "_id" : ObjectId("4d608614238d3b4ade000001"),
   "md5" : BinData(5,"K1ud3EUjT49wdMdkOGjbDg=="),
   "name" : "monument-thumb.jpg"
}
```

Как видите поле MD5 помечено как содержащее двоичные данные, и для него отображается подтип и полезная нагрузка.

---

[1] Так было не всегда. Ныне нерекомендуемый подтип 2 означал, что присоединенные двоичные данные включают дополнительные четыре байта размера, и это учитывалось некоторыми командами базы данных. В современных версиях по умолчанию подразумевается подтип 0, и при любом подтипе полезная двоичная нагрузка хранится одинаково. Поэтому подтипы можно рассматривать как признаки, которые могут использовать в своих целях разработчики приложений.

C.2. GridFS

GridFS – это соглашение о хранении файлов произвольного размера в MongoDB. Спецификация GridFS реализована во всех официальных драйверах и в инструменте `mongofiles`, так чтобы гарантировать согласованность на разных платформах. GridFS полезна для хранения больших двоичных объектов в базе данных. Зачастую она обеспечивает и достаточно быстрое чтение, причем метод хранения приспособлен для потокового доступа.

Сам термин *GridFS* нередко приводит к недоразумениям, поэтому сразу сделаем необходимые пояснения. Во-первых, GridFS – не внутренняя функция MongoDB. Как уже было сказано, это *соглашение*, которое поддерживают все официальные драйверы (и некоторые инструменты), чтобы иметь возможность управлять хранением больших двоичных объектов в базе данных. Во-вторых, важно понимать, что GridFS не обладает развитой семантикой настоящих файловых систем. Например, не существует протокола обеспечения блокировки и одновременного доступа, так что интерфейс GridFS ограничен только простейшими операциями put, get и delete. Следовательно, чтобы обновить файл, его необходимо сначала удалить, а потом записать новую версию.

Принцип работы GridFS основан на разбиении большого файла на блоки размером 256 КБ и сохранении каждого блока в отдельном документе. По умолчанию блоки хранятся в коллекции `fs.chunks`. После записи блоков метаданные файла сохраняются в одном документе в другой коллекции – `fs.files`. На рис. C.1 показана упрощенная схема этого процесса в применении к гипотетическому файлу `canyon.jpg` размером 1 МБ. Но довольно теории. Далее мы покажем, как GridFS применяется на практике – на примере Ruby GridFS API и утилиты `mongofiles`.

C.2.1. GridFS в Ruby

Выше мы сохраняли небольшое изображение – миниатюру. При размере всего 10 КБ ее вполне можно сохранить в одном документе. Но исходное-то изображение занимало почти 2 МБ, поэтому для его хранения GridFS будет более уместным решением. Сейчас мы сохраним исходное изображение с помощью интерфейса к GridFS API из Ruby. Сначала подключимся к базе данных, а затем инициализируем объект `Grid`, передав его конструктору ссылку на базу данных, в которой будет храниться GridFS-файл.

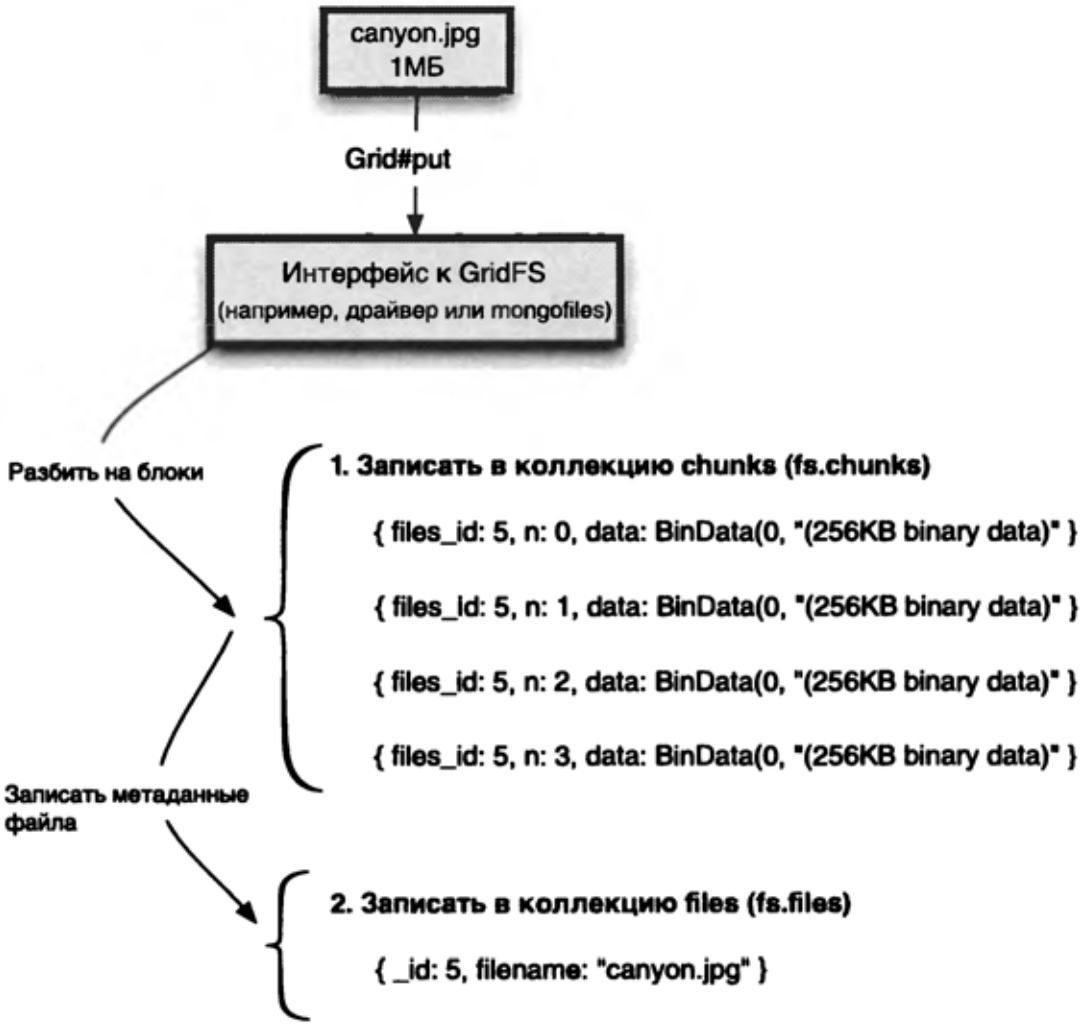


Рис. C.1. Сохранение файла с помощью GridFS

Далее откроем файл исходного изображения, `canyon.jpg`, для чтения. Простейшие методы интерфейса к GridFS – помещение и извлечение файла. В данном случае мы пользуемся методом `Grid#put`, который принимает либо строку двоичных данных, либо объект `IO`, играющий роль указателя на файл. Получив указатель, объект `Grid` запишет данные в базу.

Этот метод возвращает уникальный идентификатор объекта для сохраненного файла:

```
@con  = Mongo::Connection.new
@db   = @con["images"]

@grid = Mongo::Grid.new(@db)

filename = File.join(File.dirname(__FILE__), "canyon.jpg")
file = File.open(filename, "r")

file_id = @grid.put(file, :filename => "canyon.jpg")
```

Как уже отмечалось, в GridFS используются две коллекции для хранения данных файла. В первой (обычно она называется `fs.files`), хранятся метаданные файлов. Во второй, `fs.chunks`, хранятся блоки двоичных данных файлов. Исследуем структуру этих коллекций из оболочки.

Переключитесь на базу данных `images` и запросите первую запись в коллекции `fs.files`. Будут выведены метаданные только что сохраненного файла:

```
> use images
> db.fs.files.findOne()
{
   "_id" : ObjectId("4d606588238d3b4471000001"),
   "filename" : "canyon.jpg",
   "contentType" : "binary/octet-stream",
   "length" : 2004828,
   "chunkSize" : 262144,
   "uploadDate" : ISODate("2011-02-20T00:51:21.191Z"),
   "md5" : "9725ad463b646ccbd287be87cb9b1f6e"
}
```

Это минимально необходимый набор атрибутов для любого GridFS-файла. По большей части они не требуют пояснений. Здесь мы видим, что размер файла составляет примерно 2 МБ, и он разбит на блоки по 256 КБ каждый. Хранится также MD5-свертка. Спецификация GridFS настаивает на хранении контрольной суммы, позволяющей убедиться, что сохраненный файл совпадает с оригиналом.

В каждом блоке хранится идентификатор объекта файла-владельца – в поле `files_id`. Таким образом, можно легко подсчитать количество блоков, принадлежащих файлу:

```
> db.fs.chunks.count({"files_id" : ObjectId("4d606588238d3b4471000001")})
8
```

При данном размере блока и общем размере файла неудивительно, что получилось именно восемь блоков. Содержимое блоков просмотреть тоже просто. Как и раньше, мы исключаем поле `data`, чтобы результат можно было прочитать на экране. Этот запрос возвращает первый из восьми блоков, на что указывает его порядковый номер *n*:

```
> db.fs.chunks.findOne({files_id: ObjectId("4d606588238d3b4471000001")},
                       {data: 0})
{
   "_id" : ObjectId("4d606588238d3b4471000002"),
   "n" : 0,
```

```
   "files_id" : ObjectId("4d606588238d3b4471000001")
}
```

Читать GridFS-файлы не сложнее, чем записывать. В примере ниже мы пользуемся методом `Grid#get` для возврата `IO`-подобного объекта `GridIO`, представляющего файл. Теперь можно потоком записать GridFS-файл обратно в файловую систему. В данном случае мы читаем по 256 КБ и записываем копию оригинального файла:

```
image_io = @grid.get(file_id)

copy_filename = File.join(File.dirname(__FILE__), "canyon-copy.jpg")
copy = File.open(copy_filename, "w")

while !image_io.eof? do
   copy.write(image_io.read(256 * 1024))
end

copy.close
```

Можете сами проверить, что файлы идентичны[2]:

```
$ diff -s canyon.jpg canyon-copy.jpg
Files canyon.jpg and canyon-copy.jpg are identical
```

Итак, мы рассмотрели основы чтения и записи GridFS-файлов с помощью драйвера. Различные GridFS API слегка различаются, но, располагая этими примерами и знанием принципов работы GridFS, вы без труда разберетесь в том, что написано в документации по драйверу.

C.2.2. Доступ к GridFS с помощью mongofiles

В дистрибутив MongoDB включена удобная утилита `mongofiles` для перечисления, добавления, получения и удаления GridFS-файлов из командной строки. Например, вот как можно получить список GridFS-файлов в базе данных `images`:

```
$ mongofiles -d images list
connected to: 127.0.0.1
canyon.jpg 2004828
```

Добавлять файлы столь же просто. В примере ниже мы добавляем еще одну копию изображения, которое раньше добавили из скрипта на Ruby:

---

[2] Предполагается, что в системе установлена утилита `diff`.

```
$ mongofiles -d images put canyon-copy.jpg
connected to: 127.0.0.1
added file: { _id: ObjectId('4d61783326758d4e6727228f'),
filename: "canyon-copy.jpg",
chunkSize: 262144, uploadDate: new Date(1298233395296),
md5: "9725ad463b646ccbd287be87cb9b1f6e", length: 2004828 }
```

Еще раз выведем список файлов, чтобы убедиться в том, что копия действительно добавлена:

```
$ mongofiles -d images list
connected to: 127.0.0.1
canyon.jpg 2004828
canyon-copy.jpg 2004828
```

У программы `mongofiles` есть много параметров, получить справку по ним можно, запустив ее с флагом `--help`:

```
$ mongofiles --help
```

ПРИЛОЖЕНИЕ D. MongoDB на PHP, Java и C++

В этой книге мы рассматривали MongoDB сквозь призму языков JavaScript и Ruby. Но есть и много других способов взаимодействия с MongoDB, и в этом приложении я для полноты картины представлю три из них. Начну с PHP, потому что это популярный скриптовый язык программирования. Язык Java я включил, так как он все еще, наверное, чаще других применяется при разработке систем масштаба предприятия и важен для многих читателей. К тому же, API драйвера для Java существенно отличается от используемого в скриптовых языках. И, наконец, я представлю драйвер для C++, поскольку именно на этом языке написано ядро MongoDB, и он будет интересен тем разработчикам, которые пишут высокопроизводительные автономные приложения.

В каждом разделе сначала описывается, как конструировать документы и устанавливать соединения, а затем приводится пример полной программы, которая вставляет, обновляет, выбирает и удаляет документы. Все программы выполняют одни и те же операции и порождают одни и те же результаты, поэтому их легко сравнивать. В качестве образца фигурирует документ, который мог бы хранить простой веб-паук; ниже для справки он приведен в формате JSON:

```
{  url: "org.mongodb",
   tags: ["database", "open-source"],
   attrs: { "last-visit" : ISODate("2011-02-22T05:18:28.740Z"),
      "pingtime" : 20
   }
}
```

D.1. PHP

Сообщество PHP с энтузиазмом восприняло MongoDB, в немалой степени благодаря качеству драйвера. Пример кода выглядит почти так же, как на Ruby.

D.1.1. Документы

В PHP массивы реализованы в виде упорядоченных словарей, поэтому они отлично ложатся на BSON-документы. Для создания простого документа можно использовать массивы-литералы PHP:

```
$basic = array( "username" => "jones", "zip" => 10011 );
```

Массивы в PHP могут быть вложенными. Следующий сложный документ включает массив тегов и поддокумент, содержащий дату последнего доступа `last_access` и целочисленное время отклика `pingtime`. Отметим, что для представления даты необходимо использовать специальный класс `MongoDate`:

```
$doc = array( "url" => "org.mongodb",
   "tags" => array( "database", "open-source"),
   "attrs" => array( "last_access" => new MongoDate(),
      "pingtime" => 20
   )
);
```

D.1.2. Подключение

Для подключения к одному узлу применяется конструктор `Mongo`:

```
$conn = new Mongo( "localhost", 27017 );
```

Чтобы подключиться к набору реплик, передайте конструктору `Mongo` URI со схемой MongoDB. Необходимо также задать аргумент `array("replicaSet" => true)`:

```
$repl_conn = new Mongo( "mongo://localhost:30000,localhost:30001",
array( "replicaSet" => true ));
```

Схема URI для MongoDB. URI-адрес со схемой для MongoDB – стандартный способ задания параметров соединения, применяемый в разных драйверах. Большинство драйверов принимают URI соединения, так как это позволяет упростить конфигурирование гетерогенной системы с сервером MongoDB. Актуальную спецификацию формата URI можно найти в официальной документации по MongoDB в сети.

PHP-приложения обычно работают гораздо эффективнее, когда устанавливается постоянное подключение. Для этого задайте аргумент `array( "persistent" => "x" )`, где `"x"` – уникальный идентификатор создаваемого постоянного соединения:

```
$conn = new Mongo( "localhost", 27017, array( "persist" => "x" ) );
```

D.1.3. Пример программы

На примере следующей программы на PHP демонстрируется вставка, обновление, выборка и удаление документа. Здесь же показано несколько представлений BSON-документов на PHP:

```
<?php
   $m = new Mongo( "localhost", 27017 );
   $db = $m->crawler;
   $coll = $db->sites;
   $doc = array("url" => "org.mongodb",
                "tags" => array( "database",
                                 "open-source"),
                "attrs" => array( "last_access" =>
                                  new MongoDate(),
                "pingtime" => 20
                                 )
               );

   $coll->insert( $doc );

   print "Начальный документ:\n";
   print print_r( $doc );

   print "Обновляется pingtime...\n";
   $coll->update(
      array( "_id" => $doc["_id"] ),
      array( '$set' => array( 'attrs.pingtime' => 30 ) ) )
   );

   print "После обновления:\n";
   $cursor = $coll->find();
   print print_r( $cursor->getNext() );

   print "\nКоличество документов на сайте: " . $coll->count() . "n";
   print "Удаляются документы...\n";
   $coll->remove();
?>
```

D.2. Java

Из всех драйверов MongoDB Java, пожалуй, используется в производственных системах чаще всего. Он применяется не только в приложениях на чистом Java, но и лежит в основе драйверов для языков, построенных на базе JVM, например: Scala, Clojure и JRuby. Из-за отсутствия в Java словарей-литералов конструирование BSON-документов выглядит несколько многословно, но в целом работать с драйвером просто.

D.2.1. Документы

Чтобы сконструировать BSON-документ, нужно инициализировать объект `BasicBSONObject`, который реализует интерфейс `Map` и тем самым предоставляет простой API, в центре которого находятся операции `get()` и `put()`.

Конструктор `BasicBSONObject` принимает необязательную начальную пару ключ-значение. Простой документ создается следующим образом:

```
DBObject simple = new BasicDBObject( "username", "Jones" );
simple.put( "zip", 10011 );
```

Для добавления поддокумента нужно создать дополнительный объект `BasicBSONObject`. Массивы документов представляются стандартным массивом Java:

```
DBObject doc = new BasicDBObject();
String[] tags = { "database", "open-source" };

doc.put("url", "org.mongodb");
doc.put("tags", tags);

DBObject attrs = new BasicDBObject();
attrs.put( "lastAccess", new Date() );
attrs.put( "pingtime", 20 );

doc.put( "attrs", attrs );

System.out.println( doc.toString() );
```

Наконец, для просмотра документа следует воспользоваться его методом `toString()`.

D.2.2. Подключение

Подключиться к одному узлу совсем просто, нужно только поместить вызов внутрь блока try:

```
try {
   Mongo conn = new Mongo("localhost", 27017);
} catch (Exception e) {
   throw new RuntimeException(e);
}
```

Для подключения к набору реплик следует построить список объектов ServerAddress и передать его конструктору класса Mongo:

```
List servers = new ArrayList();
servers.add( new ServerAddress( "localhost" , 30000 ) );
servers.add( new ServerAddress( "localhost" , 30001 ) );

try {
   Mongo replConn = new Mongo( servers );
} catch (Exception e) {
   throw new RuntimeException(e);
}
```

В драйвер для Java включена гибкая поддержка механизма гарантирования записи. Можно задавать различные параметры на уровне объектов Mongo, DB и DBCollection, а также в любом методе записи объекта DBCollection. В примере ниже задается глобальная гарантия записи для установленного соединения с помощью конфигурационного класса WriteConcern:

```
WriteConcern w = new WriteConcern( 1, 2000 );
conn.setWriteConcern( w );
```

D.2.3. Пример программы

Следующая программа является результатом трансляции на Java написанного ранее варианта на PHP. Она понятна без пояснений.

```
import com.mongodb.Mongo;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;
import com.mongodb.DBCursor;
import com.mongodb.WriteConcern;
import java.util.Date;
```

```
public class Sample {

   public static void main(String[] args) {

      Mongo conn;
      try {
         conn = new Mongo("localhost", 27017);
      } catch (Exception e) {
         throw new RuntimeException(e);
      }

      WriteConcern w = new WriteConcern( 1, 2000 );
      conn.setWriteConcern( w );

      DB db = conn.getDB( "crawler" );
      DBCollection coll = db.getCollection( "sites" );

      DBObject doc = new BasicDBObject();
      String[] tags = { "database", "open-source" };

      doc.put("url", "org.mongodb");
      doc.put("tags", tags);

      DBObject attrs = new BasicDBObject();
      attrs.put( "lastAccess", new Date() );
      attrs.put( "pingtime", 20 );

      doc.put( "attrs", attrs );

      coll.insert(doc);

      System.out.println( "Начальный документ:\n" );
      System.out.println( doc.toString() );

      System.out.println( "Обновляется pingtime...\n" );
      coll.update( new BasicDBObject( "_id", doc.get("_id") ),
         new BasicDBObject( "$set", new BasicDBObject( "pingtime", 30 ) ) );

      DBCursor cursor = coll.find();
      System.out.println( "После обновления\n" );
      System.out.println( cursor.next().toString() );

      System.out.println( "Количество документов на сайте: " + coll.count() );

      System.out.println( "Удаляются документы...\n" );
      coll.remove( new BasicDBObject() );
   }
}
```

D.3. C++

Драйвер C++ имеет смысл рекомендовать за его быстродействие и близость к ядру сервера. Трудно найти более быстрый драйвер, а если вам интересно, как MongoDB устроена изнутри, то изучение исходного кода драйвера для C++ станет неплохой отправной точкой. Это не столько автономный драйвер, сколько неотъемлемая составная часть MongoDB API, поэтому он тесно переплетен с кодом сервера. Однако его можно использовать и как независимую библиотеку.

D.3.1. Документы

Существует два способа создать BSON-документ на C++. Можно использовать несколько громоздкий объект `BSONObjBuilder` или положиться на обертывающие его макросы. Я продемонстрирую оба способа.

Начнем с простого документа:

```
BSONObjBuilder simple;
simple.genOID().append("username", "Jones").append( "zip", 10011 );
BSONObj doc = simple.obj();

cout << doc.jsonString();
```

Обратите внимание, что идентификатор объекта генерируется явно путем обращения к функции `genOID()`. В C++ BSON-объекты статические, то есть функция вставки не может модифицировать их, как в других драйверах. Если вы хотите иметь ссылку на идентификатор объекта после вставки, то должны сгенерировать его самостоятельно.

Отметим также, что `BSONObjBuilder` необходимо перед использованием преобразовать в `BSONObj`. Для этого вызывается метод `obj()` объекта `BSONObjBuilder`. Теперь сгенерируем тот же самый документ с помощью вспомогательных макросов `BSON` и `GENOID`; при этом код получается более лаконичным:

```
BSONObj o = BSON( GENOID << "username" << "Jones" << "zip" << 10011 );
cout << o.jsonString();
```

Более сложные документы конструируются примерно так же, как в Java, то есть каждый подобъект создается отдельно. Массив можно построить с помощью стандартного класса `BSONObjBuilder`, только числовые индексы 0 и 1 задаются в виде строк. На самом деле, именно так массивы хранятся в BSON:

```
BSONObjBuilder site;
site.genOID().append("url", "org.mongodb");
BSONObjBuilder tags;
tags.append("0", "database");
tags.append(„1", „open-source");
site.appendArray( „tags", tags.obj() );

BSONObjBuilder attrs;
time_t now = time(0);
attrs.appendTimeT( „lastVisited", now );
attrs.append( „pingtime", 20 );
site.append( „attrs", attrs.obj() );

BSONObj site_obj = site.obj();
```

Как и раньше, макросы позволяют сократить размер кода. Обратите особое внимание на макросы `BSON_ARRAY` и `DATENOW` и на то, вместо каких конструкций в версии с `BSONObjBuilder` они подставляются:

```
BSONObj site_concise = BSON( GENOID << "url" << "org.mongodb"
      << "tags" << BSON_ARRAY( "database" << "open-source" )
      << "attrs" << BSON( "lastVisited" << DATENOW << "pingtime" << 20 ) );
```

Уникальным для C++ является требование явно помечать BSON-документы, которые будут использоваться в качестве селекторов запроса. Сделать это можно, например, с помощью конструктора `Query()`:

```
BSONObj selector = BSON( "_id" << 1 );
Query * q1 = new Query( selector );
cout << q1->toString() << "n";
```

И на этот раз удобнее воспользоваться макросом `QUERY`:

```
Query q2 = QUERY( "pingtime" << LT << 20 );
cout << q2.toString() << "n";
```

D.3.2. Подключение

Для подключения к одному узлу нужно просто создать экземпляр класса `DBClientConnection`. Не забывайте помещать вызов конструктора в блок `try`:

```
DBClientConnection conn;
try {
   conn.connect("localhost:27017");
}
catch( DBException &e ) {
```

```
   cout << "caught " << e.what() << endl;
}
```

Чтобы подключиться к набору реплик, нужно сначала построить вектор объектов `HostAndPort`, а затем передать имя набора реплик и этот вектор конструктору класса `DBClientReplicaSet`. Для просмотра содержимого объекта следует вызвать метод `toString()`:

```
std::vector<HostAndPort> seeds (2);
seeds.push_back( HostAndPort( "localhost", 30000 ) );
seeds.push_back( HostAndPort( "localhost", 30001 ) );

DBClientReplicaSet repl_conn( "myset", seeds );
try {
   repl_conn.connect();
catch( DBException &e ) {
   cout << "caught " << e.what() << endl;
}

cout << repl_conn.toString();
```

D.3.3. Пример программы

Главное, на что нужно обратить внимание в примере программы на C++, – это отсутствие явных классов, абстрагирующих базы данных и коллекции. Все операции вставки, обновления, выборки и удаления выполняются от имени самого объекта соединения. Имя базы данных и коллекции задаются в первом аргументе метода в виде пространства имен (`crawler.sites`):

```
#include <iostream>
#include <ctime>
#include "client/dbclient.h"

using namespace mongo;

int main() {
    DBClientConnection conn;

    try {
      conn.connect("localhost:27017");
    }
    catch( DBException &e ) {
        cout << "caught " << e.what() << endl;
    }

    BSONObj doc = BSON( GENOID << "url" << "org.mongodb"
      << "tags" << BSON_ARRAY( "database" << "open-source" )
```

```
    << "attrs" << BSON( "lastVisited" << DATENOW << "pingtime" << 20 ) );

    cout << "Начальный документ:\n" << doc.jsonString() << "\n";
    conn.insert( "crawler.sites", doc );

    cout << "Обновляется pingtime...\n";
    BSONObj update = BSON( "$set" << BSON( "attrs.pingtime" << 30) );
    conn.update( «crawler.sites», QUERY(«_id» << doc[«_id»]), update);

    cout << «После обновления:\n";
    auto_ptr<DBClientCursor> cursor;
    cursor = conn.query( "crawler.sites", QUERY( "_id" << doc["_id"]) );
    cout << cursor->next().jsonString() << "\n";

    cout << "Количество документов на сайте: " <<
      conn.count( "crawler.sites" ) << "\n";

    cout << "Удаляются документы...\n";
    conn.remove( "crawler.sites", BSONObj() );

    return 0;
}
```

ПРИЛОЖЕНИЕ Е. Пространственные индексы

По мере распространения интеллектуальных мобильных устройств постоянно растет спрос на службы с привязкой к местоположению пользователя. Для разработки таких приложений необходима база данных, умеющая индексировать и запрашивать пространственные данные. Эта возможность была включена в план развития MongoDB еще на ранних этапах, и теперь средства пространственного индексирования MongoDB, хотя и не такие развитые, как, скажем, в PostGIS, лежат в основе географических запросов на многих популярных сайтах[1].

Как явствует из самого названия, пространственные индексы оптимизированы для описания географического местоположения. В MongoDB эти данные обычно представляются в виде широты и долготы в географической системе координат. Пространственный индекс над такими данными позволяет выполнять запросы с учетом местоположения пользователя. Например, в приложении может быть коллекция, содержащая меню и координаты всех ресторанов в Нью-Йорке. Построив индекс по местоположению ресторанов, вы сможете запросить у базы ближайшее к Бруклинскому мосту заведение, в котором подают черную икру.

Более того, индексатор достаточно общий и применим не только к Земле. В частности, можно проиндексировать местоположения, заданные двумерными координатами на плоскости или, скажем, на Марсе[2]. Но в любом случае построение и запросы к пространственно-

[1] Самым известным является сайт Foursquare (http://foursquare.com). О том, как MongoDB применяется в Foursquare, можно прочесть на странице http://mng.bz/rh4n.

[2] Блестящий пример первого варианта – похожая на Scrabble игра WordSquared (http://wordsquared.com), в которой пространственные индексы MongoDB применяются для поиска плиток на игровом поле.

му индексу – дело сравнительно простое. Ниже я опишу, как строятся такие индексы и какие запросы к ним возможны.

E.1. Основы пространственного индексирования

Для демонстрации пространственного индексирования в MongoDB мы будем пользоваться базой данных почтовых индексов США. Эти данные можно скачать со страницы http://mng.bz/dOpd. После распаковки архива вы получите JSON-файл, который можно импортировать с помощью `mongoimport`:

```
$ mongoimport -d geo -c zips zips.json
```

Сначала познакомимся со структурой документа, описывающего почтовый индекс. Если вы следовали инструкциям по импортированию, то для выборки документа нужно выполнить такие команды:

```
> use geo
> db.zips.findOne({zip: 10011})
{
   "_id" : ObjectId("4d291187888cec7267e55d24"),
   "city" : "New York City",
   "loc" : {
      "lon" : -73.9996
      "lat" : 40.7402,
   },
   "state" : "New York",
   "zip" : 10011
}
```

Помимо города, штата и почтового индекса, в документе присутствует четвертое поле, `loc`, в котором хранятся географические координаты центра региона с данным почтовым индексом. Именно по этому полю мы хотим построить индекс и в дальнейшем предъявлять запросы. Пространственные индексы можно строить только по полям, содержащим значения координат. Однако конкретная форма этих полей не фиксирована. Никто не мешает назвать ключи координат по-другому:

```
{ "loc" : { "x" : -73.9996, "y" : 40.7402 } }
```

Или использовать массив из двух элементов:

```
{ "loc" : [ -73.9996, 40.7402 ] }
```

При условии, что координаты представлены двумя значениями –

все равно, в поддокумент или в массиве, – над полем может быть построен пространственный индекс.

При создании индекса необходимо задать его тип – 2d:

```
> use geo
> db.zips.ensureIndex({loc: '2d'})
```

Эти команда строит индекс по полю `loc`. Индексируются только документы, в которых значения координат правильно отформатированы, поэтому пространственные индексы всегда являются разреженными. По умолчанию минимальное и максимальное значение координат равны соответственно –180 и 180.

Это естественный диапазон географических координат, но если требуется построить индекс для другой предметной области, то можно явно задать значения `min` и `max` следующим образом:

```
> use games
> db.moves.ensureIndex({loc: '2d'}, {min: 0, max: 64})
```

Имея пространственный индекс, можно предъявлять пространственные запросы[3]. Простейший и самый распространенный тип пространственного запроса – `$near` (рядом). В сочетании с ограничением `limit` запрос `$near` находит *n* ближайших к указанной географической точке мест. Например, следующий запрос находит три почтовых индекса, ближайших к Центральному вокзалу Нью-Йорка:

```
> db.zips.find({'loc': {$near: [ -73.977842, 40.752315 ]}}).limit(3)
{  "_id" : ObjectId("4d291187888cec7267e55d8d"), "city" : "New York City",
   "loc" : { "lon" : -73.9768, "lat" : 40.7519 },
   "state" : "New York", "zip" : 10168 }
{  "_id" : ObjectId("4d291187888cec7267e55d97"), "city" : "New York City",
   "loc" : { "lon" : -73.9785, "lat" : 40.7514 },
   "state" : "New York", "zip" : 10178 }
{  "_id" : ObjectId("4d291187888cec7267e55d8a"), "city" : "New York City",
   "loc" : { "lon" : -73.9791, "lat" : 40.7524 },
   "state" : "New York", "zip" : 10165 }
```

Чтобы получить ответ быстро, задавайте разумное ограничение. Если ограничение вообще не задано, то по умолчанию предполагается 100, иначе запрос вернул бы весь набор данных. Чтобы получить больше 100 результатов, укажите в методе `limit` нужное числовое значение:

```
> db.zips.find({'loc': {$near: [ -73.977842, 40.752315 ]}}).limit(500)
```

---

[3] Обратите внимание на отличие от непространственных запросов, которые можно предъявлять вне зависимости от того, есть подходящий индекс или нет.

E.2. Более сложные запросы

Хотя запросов `$near` в большинстве случаев достаточно, ими арсенал доступных средств не исчерпывается. Можно также выполнить специальную команду `geoNear`, которая возвращает расстояния до ближайших объектов и статистические данные о самом запросе:

```
> db.runCommand({'geoNear': 'zips', near: [-73.977842, 40.752315], num: 2})
{
   "ns" : "geo.zips",
   "near" : "0110000111011010011111010110010011001111111011011100",
   "results" : [
      {
         "dis" : 0.001121663764459287,
         "obj" : {
            "_id" : ObjectId("4d291187888cec7267e55d8d"),
            "city" : "New York City",
            "loc" : {
               "lon" : -73.9768,
               "lat" : 40.7519
            },
            "state" : "New York",
            "zip" : 10168
         }
      },
      {
         "dis" : 0.001126847051610947,
         "obj" : {
            "_id" : ObjectId("4d291187888cec7267e55d97"),
            "city" : "New York City",
            "loc" : {
               "lon" : -73.9785,
               "lat" : 40.7514
            },
            "state" : "New York",
            "zip" : 10178
         }
      }
   ],
   "stats" : {
      "time" : 0,
      "btreelocs" : 4,
      "nscanned" : 3,
      "objectsLoaded" : 2,
      "avgDistance" : 0.001124255408035117,
      "maxDistance" : 0.001126847051610947
   },
   "ok" : 1
}
```

Поле dist, включенное в каждый документ, – это мера расстояния от данного объекта до центральной точки. В данном случае расстояние измеряется в градусах.

Другой запрос позволяет искать объекты в пределах указанной области с помощью оператора $within. Например, чтобы найти все почтовые индексы, находящиеся в области радиусом 0,011 градусов от Центрального вокзала, нужно выполнить следующий запрос:

```
> center = [-73.977842, 40.752315]
> radius = 0.011
> db.zips.find({loc: {$within: {$center: [ center, radius ] }}}).count()
26
```

Теоретически это эквивалентно запросу $near с дополнительным параметром $maxDistance. Оба запроса возвращают все объекты, находящиеся не дальше указанного расстояния от центральной точки.

```
> db.zips.find({'loc': {$near: [-73.977842, 40.752315],
$maxDistance: 0.011}}).count()
26
```

Помимо операции $center, можно использовать оператор $box, который возвращает объекты, находящиеся внутри заданного ограничивающего прямоугольника. Например, чтобы вернуть все почтовые индексы в прямоугольнике, ограниченном Центральным вокзалом и аэропортом Ла Гардия, можно задать такой запрос:

```
> lower_left = [-73.977842, 40.752315]
> upper_right = [-73.923649, 40.762925]
> db.zips.find({loc: {$within:
{$box: [ lower_left, upper_right ] }}}).count()
15
```

Отметим, что оператор $box принимает массив из двух элементов, причем первый задает координаты левого нижнего, а второй – координаты правого верхнего угла прямоугольника.

E.3. Составные пространственные индексы

Можно строить составные пространственные индексы, но только если координата стоит на первом месте. Такие индексы можно использовать для обслуживания запросов, включающих местоположение и еще какие-то метаданные. Пусть, например, магазин садовых принадлежностей, который мы рассматривали в книге ранее, имеет

пункты розничных продаж в разных местах, причем каждый пункт предлагает свой набор услуг. Например, в базе могли бы существовать такие документы:

```
{loc: [-74.2, 40.3], services: ['nursery', 'rentals']}
{loc: [-75.2, 39.3], services: ['rentals']}
```

Тогда для эффективной обработки запросов по местоположению и услугам можно было бы построить такой индекс:

```
> db.locations.ensureIndex({loc: '2d', services: 1})
```

Теперь найти все розничные магазины, в которых имеется комната для детей, тривиально:

```
> db.locations.find({loc: [-73.977842, 40.752315], services: 'nursery'})
```

Больше о составных пространственных индексах сказать, пожалуй, нечего. Если вы не уверены, насколько эффективным будет их применение в вашем приложении, выполните для соответствующих запросов команду `explain()`.

E.4. Сферическая геометрия

Во всех приведенных выше пространственных запросах для вычисления расстояний использовалась плоская модель Земли. В частности, для нахождения расстояния между двумя точками применяется евклидова метрика[4]. В большинстве случаев, в том числе для нахождения n ближайших к данной точке объектов, этого вполне достаточно, а благодаря простоте математических вычислений результат возвращается очень быстро.

Но в действительности форма Земли близка к сферической[5]. А следовательно, по мере увеличения расстояния между точками евклидова метрика перестает быть точной. Поэтому MongoDB поддерживает также вычисление расстояния на основе двумерной сферической модели. Результат при этом получается более точным, но за счет небольшого снижения быстродействия.

Чтобы можно было воспользоваться сферической геометрией, необходимо хранить координаты в порядке «долгота-широта» и выражать угловые расстояния в радианах. Тогда во всех представленных выше запросах можно будет использовать сферические эквиваленты.

---

[4] http://ru.wikipedia.org/wiki/Евклидова_метрика

[5] На самом деле это сплющенный сфероид, выпученный в области экватора.

Например, оператор `$nearSphere` – сферический эквивалент `$near` и записывается в таком виде:

```
> db.zips.find({'loc': {$nearSphere: [ -73.977842, 40.752315 ]}}).limit(3)
```

Команда `geoNear` также поддерживает сферические вычисления, нужно только добавить параметр `{ spherical: true }`:

```
> db.runCommand({'geoNear': 'zips',
near: [-73.977842, 40.752315], num: 2, spherical: true})
```

Наконец, оператор `$centerSphere` можно использовать для поиска объектов внутри окружности, когда требуется вычислять расстояния на сфере. Только не забывайте, что угловое расстояние должно быть выражено в радианах:

```
center = [-73.977842, 40.752315]
radius_in_degrees = 0.11
radius_in_radians = radius_in_degrees * (Math.PI / 180);
db.zips.find({loc: {$within:
   {$centerSphere: [center, radius_in_radians ] }}})
```

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

T

U

V

W

X

А

Б

В

Г

Д

К

Л

М

Н

О

П

Р

С

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 Internet-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. **(499) 725-54-09, 725-50-27**;

Электронный адрес **books@alians-kniga.ru**.

Кайл Бэнкер

MongoDB в действии

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Перевод с английского *Слинкина А. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 20.02.2012. Формат 60×90 $^{1}/_{16}$.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 24,13. Тираж 200 экз.
заказ №

Web-сайт издательства: www.dmk-press.ru