Маленькая книга о MongoDB

Карл Сегуин

Об этой книге

Лицензия

Об авторе

Карл Сегуин --- разработчик, имеющий опыт работы в различных областях и технологиях. Он --- профессиональный разработчик на .NET и Ruby. Он --- участник нескольких открытых проектов, технический писатель и участник конференций. Касательно MongoDB, он принимает участие в разработке NoRM --- библиотеки на C#, создал интерактивный курс mongly, а также Mongo Web Admin. Его бесплатный сервис для разработчиков казуальных игр mogade.com также работает на MongoDB.

Карл также написал The Little Redis Book (перевод этой книги на русский выполнил Андрей Кондратович).

Его блог можно найти по адресу http://openmymind.net, его твиттер --- @karlseguin

Благодарности

Я особо благодарен Perry Neal за то, что он позволил мне воспользоваться его глазами, разумом и увлеченностью. Твоя помощь неоценима. Спасибо тебе.

Актуальная версия

Актуальная версия исходного кода книги доступна по адресу:

http://github.com/karlseguin/the-little-mongodb-book.

О переводе

Перевод подготовил Дмитрий Григорьев. Актуальную версию исходников перевода можно получить по адресу http://github.com/at8eqeq3/the-little-mongodb-book.

Введение

> Я не виноват, что главы получились такими короткими. Это все от того, что MongoDB очень легко изучать.

Часто говорят, что технология развивается немыслимыми темпами. Да, список новых технологий и техник непрерывно растет. Однако, мне всегда казалось, что фундаментальные технологии, используемые программистами, движутся значительно медленнее. Можно потратить годы на изучение того, что еще осталось актуальным. Удивляет то с какой скоростью привычные технологии заменяются новыми. Нечто, казавшееся таким привычным, внезапно оказывается под угрозой быть забытым совсем.

Самым, пожалуй, примечательным образцом таких изменений может служить рост популярности NoSQL-технологий по сравнению с реляционными базами данных. Кажется, что еще вчера весь интернет держался на нескольких СУРБД, а сегодня уже пять, или около того, NoSQL-решений показывают себя весьма полезными.

И хотя кажется, что такие вещи происходят мгновенно, в реальности могут пройти годы, пока эти технологии станут широко распространены. Начальный энтузиазм подогревается относительно небольшими группами разработчиков и компаний. Оттачиваются решения, усваиваются уроки, и вот, видя, что технологии становятся более зрелыми, все остальные постепенно начинают пробовать их самостоятельно. Частично это так применительно к NoSQL-решениям, которые не способны полностью заменить традиционные хранилища, но предназначены для решения определенного круга задач в дополнении к ним.

Кроме сказанного выше, мы должны объяснить, что мы имеем в виду, говоря «NoSQL». Это очень неконкретное название, и для разных людей оно может означать разные вещи. Лично я часто использую его, чтобы назвать некую систему, играющую роль в хранении данных. С другой стороны, NoSQL (опять же, для меня) --- это вера в то, что хранение данных не обязательно происходит в одной-единственной системе. Разработчики реляционных баз данных исторически пытаются представить свои продукты универсальными, NoSQL склоняется к небольшим зонам ответственности, где для каждой задачи может быть найден лучший инструмент. Таким образом, ваш NoSQL-стек может по-прежнему использовать реляционную базу (допустим, MySQL), но также иметь Redis в качестве persistence lookup(???) определенных частей системы, и еще Hadoop для обработки больших объемов данных. Проще говоря, NoSQL --- это быть открытым и знать об альтернативах, существующих и новых способах и инструментах для управления данными.

Вы, наверное, задумались, как MongoDB относится к этому всему. Являясь документ-ориентированной базой данных, Mongo --- это более общее NoSQL-решение. Её можно рассматривать как альтернативу реляционным базам. И, как реляционные базы, она тоже может выиграть, будучи связанной с другими, более специализированными NoSQL-решениями. MongoDB имеет свои достоинства и недостатки, о которых мы расскажем чуть дальше в этой книге.

Как вы уже, возможно, заметили, слова «MongoDB» и «Mongo» означают одно и то же.

Начало работы

Большая часть этой книги описывает основной функционал MongoDB. Поэтому мы будем работать с оболочкой MongoDB. Оболочка полезна для обучения, и, ко всему прочему, является неплохим инструментом для управления, однако при написании кода вы будете использовать MongoDB-драйверы.

Вот и первое, что мы узнаем о MongoDB --- её драйверы. У MongoDB есть множество официальных драйверов для различных языков. Эти драйверы не сильно отличаются от возможно известных вам драйверов для других БД. На основе этих драйверов сообществом разработаны более специфичные для некоторых языков и фреймворков библиотеки. Например, NoRM --- библиотека для C#, использующая LINQ, и MongoMapper --- библиотека на Ruby, совместимая с ActiveRecord. Выбрать для работы низкоуровневый драйвер или более высокоуровневую библиотеку --- дело исключительно ваше. Я говорю об этом лишь из-за того, что некоторые люди при знакомстве с MongoDB не понимают, почему есть официальные драйверы и библиотеки, разработанные сообществом; первые предназначены для низкоуровневого «общения» с БД, вторые --- более «заточенные» под конкретный язык или фреймворк.

Я настоятельно советую при чтении этой книги воспроизводить мои действия самостоятельно, а также исследовать возможности для решения собственных вопросов. MongoDB достаточно просто установить, так что давайте потратим пару минут на подготовку всего необходимого.

1. Обратитесь к официальной странице загрузок и скачайте бинарный дистрибутив из первой строчки (рекомендованную стабильную версию) для своей операционной системы. Для нужд разработки пойдет и 32-битная, и 64-битная.
2. Распакуйте архив (куда вам удобно) и перейдите в каталог `bin`. Пока не нужно ничего запускать, но запомните, что `mongod` --- это серверный процесс, а `mongo` --- это клиентская оболочка. С этими двумя программами мы и будем проводить большую часть нашего времени.
3. Создайте в `bin` текстовый файл с названием `mongodb.config`.
4. Добавьте в него всего одну строчку: `dbpath=PATH_TO_WHERE_YOU_WANT_TO_STORE_YOUR_DATABASE_FILES`. Например, под Windows это может быть `dbpath=c:\mongodb\data`, а под Linux --- `dbpath=/etc/mongodb/data`.
5. Убедитесь, что указанный в `dbpath` путь существует.
6. Запустите `mongod` с параметром `--config /path/to/your/mongodb.config`

Например, для пользователя Windows, если он распаковал архив в `c:\mongodb\` и создал `c:\mongodb\data\`, то в `c:\mongodb\bin\mongodb.config` ему нужно указать `dbpath=c:\mongodb\data\`, а запускать `mongod` из командной строки, выполнив `c:\mongodb\bin\mongod --config c:\mongodb\bin\mongodb.config`.

Чтобы сэкономить буквы, можно добавить каталог `bin` в переменную окружения `PATH`. Действия для пользователей Linux и MacOS примерно такие же. Отличаться будут только пути.

Надеюсь, что все прошло успешно и MongoDB работает. Если же вы получаете ошибку, то прочтите внимательно вывод: сервер обычно очень хорошо объясняет, что с ним не так.

Теперь вы можете запустить `mongo` (без *d*), который запустит оболочку к вашему серверу. Попробуйте ввести `db.version()`, чтобы убедиться, что все работает как надо. В идеале, вы должны увидеть номер установленной вами версии.

Глава 1 --- Основы

Мы начнём наше путешествие с исследования основных действий при работе с MongoDB. Очевидно, это есть ключ к пониманию сути MongoDB, но это также поможет найти ответы на более высокоуровневые вопросы применения MongoDB.

Для начала, вот вам 6 простых концепций, которые нужно понять:

1. MongoDB вкладывает в понятие «база данных» уже привычное вам значение (оракловоды называют это «схема»). В каждом экземпляре MongoDB может быть 0 или более баз данных, каждая из которых является контейнером для всего остального.
2. База данных может содержать 0 или более «коллекций». Коллекции во многом похожи на таблицы, и представлять их именно такими не будет ошибкой.
3. Коллекции состоят из 0 или более «документов». Документ, как уже можно догадаться, очень похож на строку.
4. Документ состоит и 1 или более «полей», которые, внезапно, можно сравнить со столбцами.
5. «Индексы» в MongoDB работают примерно так же, как и их тезки в СУРБД.
6. «Курсоры» отличаются от первых пяти концепций, но они не менее важны, хотя про них и часто забывают. Я думаю, они заслуживают отдельного упоминания. Самый важный момент касательно курсоров --- это то, что когда вы запрашиваете данные у MongoDB, она возвращает курсор, с которым мы и работаем --- считаем количество документов, листаем вперед-назад, а не сами данные.

Еще раз, MongoDB сделана из **баз данных**, которые содержат **коллекции**. **Коллекция** состоит из **документов**. Каждый **документ** имеет **поля**. **Коллекции** могут быть **проиндексированы** для повышения производительности поиска и сортировки. И, наконец, когда мы запрашиваем данные у MongoDB, мы делаем это через **курсор**, а реальное выполнение запросов откладывается, пока данные не станут необходимы.

Вам, вероятно, интересно, зачем использовать новую терминологию (коллекции вместо таблиц, документы вместо строк, поля вместо столбцов). Чтобы всё усложнить? На самом деле, соль в том, что хотя эти понятия и похожи на своих реляционных «коллег», они не идентичны. Основное различие в том, что реляционные базы определяют **столбцы** на уровне **таблиц**, тогда как документ-ориентированная база определяет **поля** на уровне **документов**. Таким образом, каждый **документ** в **коллекции** может иметь свой собственный уникальный набор **полей**. По сути, **коллекция** --- это всего лишь контейнер по сравнению с **таблицей**, а **документ** хранит намного больше информации, чем **строка**.

Хотя это и очень важно понимать, не переживайте, если пока еще не все абсолютно ясно. После пары insert-ов будет понятнее. По большому счёту, суть в том, что коллекции не принци-

пиально, что происходит внутри неё (она бессхемна). Поля принадлежат каждому отдельному документу. Достоинства и недостатки такого подхода мы изучим в одной из следующих глав.

Предлагаю немного повозиться. Если вы еще не запустили сервер и оболочку, самое время сделать это. Оболочка выполняет JavaScript. В ней есть несколько глобальных команд, таких как `help` или `exit`. Команды, выполняемые на текущей базе данных, выполняются на объекте db, что-то в духе `db.help()` или `db.stats()`. Команды на определенной коллекции (которые нам нужны чаще всех), выполняются на объекте `db.COLLECTION_NAME` --- `db.unicorns.help()` или `db.unicorns.count()`.

Введите в оболочке команду `db.help()`, и вы получите список команд, которые можно выполнить на объекте db.

Небольшое отступление. Поскольку в оболочке у нас JavaScript, если вы выполните метод и забудете про скобки `()`, вам вернется тело метода, а не результат его выполнения. Я говорю об этом, чтобы когда вы в первый раз сделаете так и увидите ответ, начинающийся с `function(...){`, вы не очень пугались. Например, если вы напишете `db.help` (без скобок), вы получите код метода `help`.

Для начала мы воспользуемся глобальным методом `use` для переключения базы данных. Введите `use learn`. Не важно, что упомянутая база данных на самом деле не существует: как только мы создадим в ней первую коллекцию, мы одновременно создадим и БД `learn`. Теперь, находясь внутри базы данных, мы можем выполнять специфичные для базы команды, такие как `db.getCollectionNames`. Сделав это, вы получите пустой массив (`[ ]`). Поскольку коллекции бессхемны, нам не нужно явно создавать их. Мы можем просто вставить документ в новую коллекцию. Для этого у нас есть команда `insert`, которой мы передаем создаваемый документ:

```
db.unicorns.insert({name: 'Aurora', gender: 'f', weight: 450})
```

Этой строкой мы осуществляем вставку (`insert`) на коллекции `unicorns`, передавая методу единственный аргумент. Внутри MongoDB использует двоичный сериализованный JSON. Внешне же это означает, что нам придется часто использовать JSON, как в случае с нашим параметром. Если мы теперь выполним `db.getCollectionNames()`, мы получим целых две коллекции --- `unicorns` и `system.indexes`. `system.indexes` создается единожды в каждой базе данных и хранит сведения о ее индексах.

Теперь на `unicorns` можно выполнить команду `find` и получить список документов:

```
db.unicorns.find()
```

Обратите внимание, что кроме введенных вами данных, есть еще поле `_id`. Каждый документ должен иметь уникальное поле `_id`. Вы можете генерировать его самостоятельно или позволить MongoDB создавать ObjectId за вас. Чаще всего вы будете доверять это MongoDB.

По умолчанию, поле `_id` проиндексировано --- это и объясняет появление коллекции `system.indexes`. Можно посмотреть и на неё:

```
db.system.indexes.find()
```

Вы получите имя индекса, базу данных и коллекцию, для которой он создан, а также поля, входящие в индекс.

Теперь мы вернемся к обсуждению бессхемных коллекций. Давайте, вставим совсем другой документ в коллекцию `unicorns`, как-то так:

```
db.unicorns.insert({name: 'Leto', gender: 'm', home: 'Arrakeen', worm: false})
```

Попробуйте снова выполнить `find`, чтобы перечислить документы. Когда мы узнаем чуть больше, мы обсудим это необычное поведение MongoDB, но, надеюсь, вы уже начинаете понимать, почему традиционная терминология не очень подходит нам.

Создание селекторов

В дополнение к уже исследованным 6 понятиям, есть ещё один практический аспект MongoDB, который важно понять прежде, чем переходить к более сложным вещам: селекторы запросов. Селекторы запросов в MongoDB похожи на `where` в SQL. Они используются для поиска, подсчёта, обновления и удаления документов из коллекций. Селектор есть JSON-объект, в своём простейшем виде выглядящий как `{}` и совпадающий со всеми документами (`null` тоже работает). Если мы хотим найти всех единорогов женского пола, мы можем использовать `{gender: 'f'}`.

Прежде, чем окончательно зарыться в селекторы, давайте подготовим немного данных, с которыми нам предстоит возиться. Для начала удалите все, что мы уже насоздавали в коллекции `unicorns` с помощью `db.unicorns.remove()` (поскольку мы не передали никакого селектора, этот вызов уничтожит все документы). Теперь выполните следующие insert-ы, чтобы ввести данные для упражнений (копирование и вставка поможет):

```
db.unicorns.insert({name: 'Horny', dob: new Date(1992,2,13,7,47), loves: ['
    carrot','papaya'], weight: 600, gender: 'm', vampires: 63});
db.unicorns.insert({name: 'Aurora', dob: new Date(1991, 0, 24, 13, 0), loves:
    ['carrot', 'grape'], weight: 450, gender: 'f', vampires: 43});
db.unicorns.insert({name: 'Unicrom', dob: new Date(1973, 1, 9, 22, 10), loves:
     ['energon', 'redbull'], weight: 984, gender: 'm', vampires: 182});
db.unicorns.insert({name: 'Roooooodles', dob: new Date(1979, 7, 18, 18, 44),
    loves: ['apple'], weight: 575, gender: 'm', vampires: 99});
db.unicorns.insert({name: 'Solnara', dob: new Date(1985, 6, 4, 2, 1), loves:['
    apple', 'carrot', 'chocolate'], weight:550, gender:'f', vampires:80});
db.unicorns.insert({name:'Ayna', dob: new Date(1998, 2, 7, 8, 30), loves: ['
    strawberry', 'lemon'], weight: 733, gender: 'f', vampires: 40});
db.unicorns.insert({name:'Kenny', dob: new Date(1997, 6, 1, 10, 42), loves: ['
    grape', 'lemon'], weight: 690,  gender: 'm', vampires: 39});
```

```
db.unicorns.insert({name: 'Raleigh', dob: new Date(2005, 4, 3, 0, 57), loves:
   ['apple', 'sugar'], weight: 421, gender: 'm', vampires: 2});
db.unicorns.insert({name: 'Leia', dob: new Date(2001, 9, 8, 14, 53), loves: ['
   apple', 'watermelon'], weight: 601, gender: 'f', vampires: 33});
db.unicorns.insert({name: 'Pilot', dob: new Date(1997, 2, 1, 5, 3), loves: ['
   apple', 'watermelon'], weight: 650, gender: 'm', vampires: 54});
db.unicorns.insert({name: 'Nimue', dob: new Date(1999, 11, 20, 16, 15), loves:
    ['grape', 'carrot'], weight: 540, gender: 'f'});
db.unicorns.insert({name: 'Dunx', dob: new Date(1976, 6, 18, 18, 18), loves: [
   'grape', 'watermelon'], weight: 704, gender: 'm', vampires: 165});
```

Теперь у нас есть данные, и мы можем начать создавать селекторы. `{field: value}` предназначен для поиска документов, где значение поля `field` совпадает с `value`. `{field1: value1, field2: value2}` --- это как бы and. Операторы `$lt`, `$lte`, `$gt`, `$gte` и `$ne` --- это «меньше», «меньше или равно», «больше», «больше или равно», «не равно» соответственно. Например, чтобы найти всех самцов-единорогов, весящих более 700 фунтов, мы можем выполнить:

```
db.unicorns.find({gender: 'm', weight: {$gt: 700}})
```

или (не совсем то же самое, но для демонстрации подойдёт):

```
db.unicorns.find({gender: {$ne: 'f'}, weight: {$gte: 701}})
```

Оператор `$exists` для поиска существующего или несуществующего поля:

```
db.unicorns.find({vampires: {$exists: false}})
```

Должно вернуть один документ. Если мы хотим объединять условия по ИЛИ, а не по И, мы используем оператор `$or`, присваивая ему массив значений, которые мы хотим объединить:

```
db.unicorns.find({gender: 'f', $or: [{loves: 'apple'}, {loves: 'orange'}, {
   weight: {$lt: 500}}]})
```

Это должно вернуть всех самок единорогов, которые либо любят яблоки, либо апельсины, либо весят менее 500 фунтов.

В примере выше есть еще один примечательный момент. Возможно, вы уже заметили, что поле `loves` --- это массив. MongoDB считает массивы первоклассными объектами. Это невероятно удобная штука. Однажды начав ею пользоваться, вы будете удивляться, как раньше вы могли жить без неё. Ещё интереснее то, насколько легка выборка значений на основе массивов: `{loves: 'watermelon'}` вернёт все документы, где `watermelon` есть среди значений `loves`.

На самом деле, операторов много больше, чем мы тут обсудили. Самый гибкий из них --- это `$where`, который позволяет передать JavaScript-код, выполняемый на сервере. Все операторы

описаны на странице Advanced Queries на сайте MongoDB. Мы же рассмотрели лишь основы, необходимые для старта. Но одновременно это и самые востребованные операторы.

Мы видели, как селекторы могут использоваться в команде `find`. Точно так же они могут использоваться с `remove`, которую мы уже видели, `count`, которую мы не рассматривали, но вы, возможно, уже заметили, и `update`, с которой мы вплотную познакомимся чуть позже.

`ObjectId`, который MongoDB генерирует для наших `_id`, можно выбрать с помощью:

```
db.unicorns.find({_id: ObjectId("TheObjectId")})
```

В этой главе

Мы не рассматривали ещё команду `update` и пропустили несколько интересных фишек с `find`. Однако мы установили и запустили MongoDB, коротко познакомились с командами `insert` и `remove` (на самом деле, нам про них уже почти всё известно). Мы рассмотрели команду `find` и увидели, как работают селекторы в MongoDB. Мы неплохо начали и обзавелись неплохой основой для наших будущих исследований. Хотите --- верьте, хотите --- нет, но вы уже знаете большую часть того, что вам нужно знать о MongoDB --- её действительно легко изучить и просто использовать. Я настоятельно советую вам поиграть с вашей базой данных перед тем, как двинуться дальше. Вставьте разных документов, возможно --- в другие коллекции, и познакомьтесь поближе с различными селекторами. Используйте `find`, `count` и `remove`. Через некоторое время вещи, которые казались странными и непонятными, уложатся в картинку.

Глава 2 --- Обновление

В первой главе мы познакомились с тремя из четырёх CRUD-операций (create, read, update и delete). Эта глава посвящена оставшейся --- `update`. У операции `update` есть несколько неожиданных особенностей, поэтому ей и посвщена целая глава.

Обновление: Замена против $set

В своей простейшей форме `update` принимает два аргумента: селектор (where) для нужных документов и какое поле нужно обновить. Если Roooooodles немного набрал вес, мы выполним:

```
db.unicorns.update({name: 'Roooooodles'}, {weight: 590})
```

(если вы играли с коллекцией `unicorns` и она уже не содержит нужных данных, просто снесите из неё все записи и выполните заново код из первой главы).

В реальном коде вы, возможно, использовали бы `_id` для выбора полей, но поскольку я не знаю, какие `_id` сгенерировала для вас MongoDB, мы будем использовать `name`. Давайте же посмотрим на обновлённую запись:

```
db.unicorns.find({name: 'Roooooodles'})
```

И вот вам сразу первый сюрприз от `update`. Ничего не нашлось, потому что второй параметр **заменил** исходный документ. Другими словами, `update` нашла документ по полю `name` и с чистой совестью заменила его целиком новым документом из второго параметра. Да, это не похоже на поведение `update` из SQL. В некоторых ситуациях это может быть полезным. Однако если вы хотите лишь изменить значение одного или нескольких полей, вам нужно использовать модификатор `$set`:

```
db.unicorns.update({weight: 590}, {$set: {name: 'Roooooodles', dob: new Date
    (1979, 7, 18, 18, 44), loves: ['apple'], gender: 'm', vampires: 99}})
```

Это вернет потерянным полям прежние значениея. А поскольку мы не указали `weight`, оно затронуто не будет. Теперь выполним:

```
db.unicorns.find({name: 'Roooooodles'})
```

Теперь мы получим то, что ожидали. Таким образом, правильный способ обновления веса выглядит так:

```
db.unicorns.update({name: 'Roooooodles'}, {$set: {weight: 590}})
```

Модификаторы update

В довесок к `$set`, у нас есть еще несколько модификаторов для разных интересных вещей. Эти модификаторы работают с полями, так что весь документ они не стирают. Например, модификатор `$inc` предназначен для изменения значения поля на какое-либо положительное

или отрицательное число. Например, если Pilot-у ошибочно засчитали пару убитых вампиров, мы можем исправить это:

```
db.unicorns.update({name: 'Pilot'}, {$inc: {vampires: -2}})
```

Если Aurora вдруг стала сладкоежкой, мы можем добавить значение в её список `loves` с помощью `$push`:

```
db.unicorns.update({name: 'Aurora'}, {$push: {loves: 'sugar'}})
```

Раздел Updating сайта MongoDB содержит больше информации об этих и других модификаторах

Upsert-ы

Один из самых приятных сюрпризов `update` --- это поддержка т.н. `upserts`. Это обновление документа если он существует или вставка нового в противном случае. Upsert-ы удобным в некоторых ситуациях, и вы это непременно заметите. Чтобы использовать эту фишку, нужно лишь добавить третьим параметром `true`.

Житейский пример --- счётчик посещений на сайте. Если мы хотим хранить количество посещений, нам нужно посмотреть, есть ли уже запись для нужной странички, и решить, одновить её или создать новую. Без третьего параметра (или если он установлен в `false`), следующий код не сделает ничего полезного:

```
db.hits.update({page: 'unicorns'}, {$inc: {hits: 1}});
db.hits.find();
```

Однако, с использованием upsert-ов, результат будет иным:

```
db.hits.update({page: 'unicorns'}, {$inc: {hits: 1}}, true);
db.hits.find();
```

Поскольку до сих пор не существовало документа, у которого `page` было бы равно `unicorns`, будет создан новый. Если мы выполним то же самое ещё раз, `hits` у существующего документа увеличится до 2.

```
db.hits.update({page: 'unicorns'}, {$inc: {hits: 1}}, true);
db.hits.find();
```

Множественные обновления

И последний сюрприз, приготовленный нам `update`, заключается в том, что по умолчанию обновляется только один документ. Для предыдущих примеров это выглядело логичным. Однако, если вы выполните что-то в духе:

```
db.unicorns.update({}, {$set: {vaccinated: true }});
db.unicorns.find({vaccinated: true});
```

Вы, вероятно, ожидаете увидеть всех ваших драгоценных единорогов привитыми. На самом деле, для этого нужно установить в `true` четвертый параметр:

```
db.unicorns.update({}, {$set: {vaccinated: true }}, false, true);
db.unicorns.find({vaccinated: true});
```

В этой главе

Эта глава завершает наше знакомство с CRUD-операциями, которые можно выполнять с коллекциями. Мы поближе познакомились с `update` и рассмотрели три интересных возможности. Во-первых, в отличие от обновлений в SQL, `update` в MongoDB заменяет исходный документ. Из-за этого модификатор `$set` оказывается оень полезным. Во-вторых, `update` дает нам возможность использовать upsert-ы, что особенно удобно вместе с модификатором `$inc`. И, наконец, по умолчанию `update` обновляет только первый из найденных документов.

Не забывайте, что сейчас мы рассматриваем MongoDB с точки зрения работы через оболочку. Используемые вами драйверы и библиотеки могут иметь другие умолчания или даже предоставлять иные API. Например, дрыйвер для Ruby объединяет последние два параметра в хэш: `{:upsert => false, :multi => false}`.

Глава 3 --- Поиск документов

В первой главе мы уже кратко взглянули на команду `find`. Однако, её возможности не ограничиваются селекторами. Мы уже упоминали, что результатом работы `find` является курсор. Давайте же посмотрим более детально, что к чему.

Выбор полей

Прежде, чем вплотную заняться курсорами, вам следует знать, что `find` принимает второй аргумент. Например, мы можем найти только имена всех единорогов:

```
db.unicorns.find(null, {name: 1});
```

По умолчанию, в ответе всегда будет поле `_id`. Но мы можем явно исключить его, указав `{name:1, _id: 0}`.

Кроме поля `_id` смешивать включение и исключение других полей нельзя. Если вы думали об этом, то это важно. Вам следует явно либо выбирать, либо исключать поля.

Сортировка

Я уже неоднократно упоминал, что `find` возвращает курсор, реальное выполнение которого откладывается до победного. Однако, как вы, несомненно заметили, работая с оболочкой, `find` выполняется сразу же. Такое поведение характерно только для оболочки. Истинное поведение курсоров можно увидеть, взглянув на методы, которые можно сцепить с `find`. Первый из них --- это `sort`. Он работает подобно выбору полей из предыдущего раздела. Мы указываем поля, по которым сортировать, и 1 для сортировки по возрастанию либо --1 --- по убыванию. Например:

```
//heaviest unicorns first
db.unicorns.find().sort({weight: -1})

//by vampire name then vampire kills:
db.unicorns.find().sort({name: 1, vampires: -1})
```

Как и реляционные базы данных, MongoDB может использовать индексы при сортировке. С индексами мы познакомимся чуть позже. Однако, важно знать, что MongoDB ограничивает размер сортировки без индекса. Таким образом, при попытке отсортировать большую выборку без индекса, мы получим ошибку. Некоторые считают это ограничением. По правде говоря, я бы хотел, чтобы у всех СУБД была возможность запрещать выполнение неоптимизированных запросов (я не пытаюсь превратить каждый недостаток MongoDB в достоинство, но я достаточно насмотрелся на плохо оптимизированные БД и искренне мечтаю, чтобы в них была какая-либо строгость).

Разбиение на страницы

Разбиение результата на страницы выполняется с помощью методов `limit` и `skip` курсора. Чтобы выбрать второго и третьего по весу единорога, сделаем:

```
db.unicorns.find().sort({weight: -1}).limit(2).skip(1)
```

Использование `limit` в связке с `sort` --- хороший способ избежать проблем при сортировке по неиндексированным полям

Подсчёт

Оболочка позвлоляет выполнить `count` прямо на коллекции, как-то так:

```
db.unicorns.count({vampires: {$gt: 50}})
```

В реальности же `count` --- это метод курсора, в оболочку же просто встроен костыль для этого. Драйверы, не имеющие такой плюшки, требуют такого подхода (в оболочке тоже работает):

```
db.unicorns.find({vampires: {$gt: 50}}).count()
```

В этой главе

Использование `find` и курсоров --- это явное преимущество. Есть еще несколько дополнительных команд, которые мы либо рассмотрим в дальнейших главах, либо которые предназначены для специфичных задач. Сейчас же вы уже должны достаточно комфортно чувствовать себя в оболочке mongo и понимать основы MongoDB/

Глава 4 --- Моделирование данных

Предлагаю перейти к немного более абстрактной теме, касающейся MongoDB. Объяснить несколько новых терминов и немного нового синтаксиса --- не самая сложная задача. Рассмотреть особенности моделирования с новой парадигмой --- на порядок сложнее. На самом деле, многие из нас продолжают методом проб и ошибок искать правильный подход к моделированию в этих новых технологиях. Мы лишь начнем этот разговор, но большую часть этой темы вам придется раскрыть самостоятельно, работая с реальным кодом.

По сравнению с другими NoSQL-решениями, документ-ориентированные БД, возможно, меньше всего отличаются от реляционных в плане моделирования. Разница не велика, но это не значит, что ею можно пренебречь.

Никаких объединений

Первое и самое важное отличие, которое нужно усвоить для комфортной работы с MongoDB --- это отсутствие джойнов. Я не знаю точной причины, почему некоторые возможности джойнов не поддерживаются в MongoDB, но я точно знаю, что джойны масштабируются примерно никак. Таким образом, однажды начав разделять свои данные горизонтально, вы дойдете до выполнения джойнов на клиенте (т.е. в сервере приложений). Независимо от причин, остается фактом то, что данные *являются* реляционными, а MongoDB не поддерживает джойны.

Не имея ничего другого и живя в мире без объединений, мы вынуждены выполнять их самостоятельно, из кода нашего приложения. Мы всего лишь должны выполнить еще один `find` за нужными данными. Отличий от объявления внешнего ключа не много. Давайте немного отвлечемся от единорогов и взглянем на сотрудников. Для начала мы создадим одного (я явно указываю `_id`, чтобы вы могли использовать дальнейшие примеры без изменений):

```
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d730"), name: 'Leto'})
```

Давайте добавим еще сотрудников, и назначим Leto их руководителем:

```
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d731"), name: 'Duncan'
    , manager: ObjectId("4d85c7039ab0fd70a117d730")});
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d732"), name: 'Moneo',
     manager: ObjectId("4d85c7039ab0fd70a117d730")});
```

(Не будет лишним напомнить, что значения `_id` должны быть уникальными. Поскольку вы, скорее всего, будете использовать `ObjectId` в своих приложениях, здесь тоже будет он)

Конечно же, чтобы найти всех подчинённых Leto, можно просто выполнить:

```
db.employees.find({manager: ObjectId("4d85c7039ab0fd70a117d730")})
```

Вот и всё, никакой магии. В худшем случае, отсутствие соединений будет требовать лишь дополнительного запроса (индексированного, пожалуй).

Массивы и внедрённые документы

То, что MongoDB не умеет джойны, ещё не означает, что у неё нет других хитростей. Помните, мы немного говорили о том, что MongoDB поддерживает массивы как первоклассные объекты? Это же невероятно удобно, когда нам надо организовать отношения многие-к-одному и многие-ко-многим. В качестве простого примера, если у сотрудника может оказаться два руководителя, мы просто положим их в массив:

```
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d733"), name: 'Siona',
    manager: [ObjectId("4d85c7039ab0fd70a117d730"), ObjectId("4
   d85c7039ab0fd70a117d732")] })
```

В качестве дополнительной плюшки, `manager` для некоторых документов может быть скалярным значением, а для некоторых --- массивом. Наш `find` прекрасно отработает в обоих случаях:

```
db.employees.find({manager: ObjectId("4d85c7039ab0fd70a117d730")})
```

Полагаю, вы быстро обнаружите, что массивы значений гораздо более удобны, чем эти ваши таблицы с джойнами.

Помимо массивов, MongoDB также поддерживает внедрённые документы. Попробуйте создать документ с внедрённым документом, вот так:

```
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d734"), name: 'Ghanima
   ', family: {mother: 'Chani', father: 'Paul', brother: ObjectId("4
   d85c7039ab0fd70a117d730")}})
```

И, как вы, вероятно, уже подумали, внедрённый документ можно запросить, используя точечную нотацию:

```
db.employees.find({'family.mother': 'Chani'})
```

Давайте кратко побеседуем о том, в каких случаях пригодятся внедрённые документы и как их использовать.

DBRef

MongoDB поддерживает нечто, именуемое `DBRef`, и это соглашение поддерживают многие драйверы. Когда драйвер встречает `DBRef`, он автоматически подтягивает ссылаемый документ. `DBRef` включает в себя коллекцию и идентификатор ссылаемого документа. Это, в основном, служит одной специфичной цели: когда документы из одной коллекции ссылаются на документы из другой, и наоборот. Таким образом, `DBRef` для документа document1 может указывать на документ в `managers`, а `DBRef` из document2 может указывать на документ из `employees`.

Денормализация

Ещё одной альтернативой джойнам является денормализация данных. Исторически, денормализация применялась лишь для весьма чувствительного к производительности кода, или когда данные нужно было действительно дублировать (как в журналах аудита). Однако, с ростом популярности NoSQL, многие из которых не умеют джойны, денормализация всё чаще становится частью обычного процесса моделирования. Это не значит, что нужно дублировать вообще все данные в каждом документе. Но, вместо того, чтобы бояться дублирования в своих моделях, просто обращайте внимение на то, где и что хранить.

Для примера допустим, что вы разрабатываете движок форума. Традиционный способ связать определенного пользователя с постом --- это добавить в `posts` колонку `user_id`. В такой модели вы не можете отобразить посты, не запрашивая (т.е., без объединения) пользователей. Возможной альтернативой будет хранение имени вместе с `user_id` в каждом посте. Можно даже использовать внедрённый документ для этого: `user: {id: ObjectId('Something'), name: 'Leto'}`. Разумеется, если вы разрешаете юзерам менять свои имена, вам придётся в таких случаях обновлять все посты (что есть еще один дополнительный запрос).

Иногда приспособиться к такому подходу бывает непросто. В некоторых случаях будет даже непонятно, нужен ли он. Не бойтесь экспериментировать. Он не только подходит для некоторых ситуаций, но и является единственно верным.

Что же выбрать?

Массивы идентификаторов --- это всегда полезная стратегия, когда приходится работать с отношениями один-ко-многим и многие-ко-многим. Можно даже сказать, что `DBRef` используется довольно редко, хотя вы, конечно же, можете повозиться и с ними. Этот момент обычно оставляет начинающих разработчиков в непонятках, использовать ли внедрённые документы или просто ссылаться.

Первое, что нужно знать --- это то, что каждый отдельный документ ограничен в размерах до 4 мегабайт. Знание, что документы имеют ограничения, пусть и достаточно большие, подсказывает нам, как их лучше использовать. Сейчас будет казаться, что большинство разработчиков делают больший упор на ссылки для своих отношений. Внедренные документы тоже часто используются, но лишь для хранения небольших кусочков данных, которые непременно надо таскать вмете с объемлющим документом. В качестве живого примера я использую документов `accounts` вместе с каждым пользователем, как-то так:

```
db.users.insert({name: 'leto', email: 'leto@dune.gov', account: {
   allowed_gholas: 5, spice_ration: 10}})
```

Это не означает, что нужно недооценивать мощь внедрённых документов и использовать их как что-то менее полезное. Проецирование модели данных на объекты реального мира упрощает принятие решений и часто избавляет от нужды в джойнах. Это будет особенно

верно, когда вы увидите, что MongoDB позволяет запрашивать и индексировать поля внедрённых документов.

Мало или много коллекций

Зная, что коллекции не имеют никакой схемы, возможно построить систему с единственной коллекцией, хранящей совсем разношёрстные документы. Но по моим личным наблюдениям, системы в MongoDB чаще строятся аналогично реляционным системам: то, что в РБД было бы таблицей, будет коллекцией в MongoDB (таблицы для отношений многие-ко-многим --- это важное исключение).

Беседа становится интереснее, когда вы переходите к внедрённым документам. Часто вспоминаемый при этом пример --- это блог. Следует ли нам иметь коллекцию `posts` и коллекцию `comments`, или посты должны иметь массив внедрённых в них комментов? Памятуя об ограничении в 4 мегабайта (весь «Гамлет» весит меньше 200 килобайт, напишут ли вам столько комментов?), большинство разработчиков предпочтут разделить коллекции. Это более ясно и чётко.

Жёстких правил не существует (ну, кроме всё тех же 4 МБ). Попробуйте применять различные подходы, и вы поймёте, что в вашем случае хорошо, а что --- нет.

В этой главе

Нашей целью в этой главе было предоставить несколько полезных идей для моделирования ваших данных в MongoDB. Отправная точка, ежели угодно. Моделирование в документ-ориентированной системе отличается, но не радикально, от реляционного мира. Вы получаете большую гибкость и одно ограничение, но для новых систем всё просто идеально. Единственный ошибочный подход --- это не пытаться делать что-либо.

Глава 5 --- Когда использовать MongoDB

К этому моменту у вас уже должно было сложиться хорошее понимание того, как вы сможете использовать MongoDB в ваших существующих системах. От количества новых технологий хранения даже голова может закружиться.

Для меня лично самый важный урок --- это то, что больше не нужно завязываться на единственное решение для хранения данных. Несомненно, единственное хранилище имеет неоспоримые преимущества, и для множества (пожалуй, даже большинства) проектов оно будет оптимальным подходом. Идея не в том, что вы должны использовать различные технологии, но в том, что вы можете это сделать. Только вы можете решить, когда стоит добавлять еще одно решение.

С учётом всего сказанного, я надеюсь, что вы видите MongoDB как решение общего назначения. Мы уже упоминали несколько раз, что документ-ориентированные БД имеют много общего с реляционными. Вместо того, чтобы играть словами, давайте просто договоримся, что MongoDB можно рассматривать как прямую альтернативу реляционным базам. Там, где кто-то видит Lucene как улучшение для полнотекстовых индексов, или Redis как персистентное хранилище ключ-значение, MongoDB будет центральным хранилищем ваших данных.

Прошу заметить, что я назвал MongoDB не *заменой* реляционных БД, но *альтернативой* им. Это инструмент, который может делать то же самое, что и все остальные. Что-то MongoDB делает лучше, что-то --- хуже. Давайте разберёмся, что именно.

Отсутствие схемы

Одним из самых часто упоминаемых преимуществ документ-ориентированных баз является отсутствие схемы. Это делает их много более гибкими, чем традиционные таблицы. Я согласен, что бессхемность --- это удобная штука, но совсем не по той причине, которая многими упоминается.

Люди говорят о бессхемности как если бы вы вдруг стали хранить чумовую мешанину данных. Существуют такие области и наборы данных, которые действительно трудно смоделировать для реляционных БД, но это же только крайние случаи. Бессхемность --- это здорово, но большая часть ваших данных так или иначе структурирована. Согласен, иметь отдельные расхождения удобно, особенно при добавлении нового функционала, но в реальности нет ничего, что бы нельзя было решить с помощью столбцов, допускающих пустые значения.

Для меня, реальными преимуществами бессхемности являются более простая настройка и лёгкость связывания с ООП. Это особенно верно, когда вы работаете со статическими языками. Я работал с MongoDB в C# и Ruby, и разница впечатляет. Динамизм Ruby и его популярная библиотека ActiveRecord сами по себе уменьшают трудности связи данных и объектов. Я не хочу сказать, что MongoDB не подходит для Ruby, ещё как подходит. Многие Ruby-разработчики увидят в MongoDB просто улучшение, тогда как программисты на C# или Java заметят серьёзные различия в подходе к работе с данными.

Подумайте об этом с точки зрения разработчика драйвера. Вы хотите сохранить объект? Сериализуйте его в JSON (на самом деле, BSON, но разница не велика) и отправьте его MongoDB. Не нужно никаких соответствия свойств и типов. Эта прямолинейность, определённо, влияет на вас, разработчиков.

Операции записи

Есть одна задача, для которой MongoDB очень хорошо подходит --- логирование. У неё есть две хитрости, делающие запись очень быстрой. Во-первых, команда на запись выполняется мгновенно, а не заставляет нас ждать, пока запись действительно произойдёт. Во-вторых, с представлением журналирования в версии 1.8 и улучшениями в 2.0, записью можно управлять для достижения нужной надёжности данных. Такие настройки, в дополнение к указанию, сколько серверов должны получить данные, прежде чем они будут считаться записанными успешно, можно указывать для каждой операции записи, что даёт нам возможность контролировать производительность записи и надёжность данных.

В довесок к высокой производительности, логирование также выигрывает от бессхемности коллекций. И, в конце концов, у MongoDB есть плюшка под названием capped collection. По умолчанию все коллекции создаются обычными, мы можем сделать их capped, явно указав соответствующий флаг в `db.createCollection`:

```
//limit our capped collection to 1 megabyte
db.createCollection('logs', {capped: true, size: 1048576})
```

Когда наша коллекция дорастёт до 1 мегабайта, старые документы будут автоматически удалены. Можно также установить ограничение на количество документов, а не на объём, для этого есть `max`. У ограниченных коллекций есть ряд интересных свойств. Например, можно обновлять документ, но ему будет нельзя увеличиваться в размерах. Также, нам доступен порядок создания документов, поэтому нам не нужен индекс для сортировки по времени создания

Не будет лишним сказать, что если вам нужно знать, не случилось ли ошибок при записи (по умолчаниютони не выводятся), вы можете выполнить команду `db.getLastError()`. Многие драйверы добавляют метод для *безопасной записи*, указывая параметр `{:safe => true}` вторым параметром `insert`.

Надёжность

До версии 1.8 в MongoDB не было обеспечения надёжности для единственного сервера. То есть, при падении сервера данные, скорее всего, терялись. Выходом из этой ситуации был запуск множества серверов (MongoDB умеет репликацию). Важной фишкой, добавленной в 1.8, стало журналирование. Чтобы включить его, добавьте строку `journal=true` в ваш `mongodb.config` который мы создали при установке (и не забудьте перезапустить сервер, чтобы он подхватил изменения). Скорее всего, вы захотите иметь журналирование включенным (в будущих релизах это будет по умолчанию). Однако, в некоторых обстоятельствах отключе-

ние журналирования может быть вполне оправданным риском (ведь некоторым приложениям не страшно терять данные).

Надёжность упоминается здесь лишь для того, чтобы рассказать, как много было сделано для преодоления ненадёжности единственного сервера. Информация о ненадежности, несомненно, ещё не раз попадётся вам на глаза. Просто учтите, что она протухла.

Полнотекстовый поиск

Настоящий полнотекстовый поиск --- это то, чего все ждут с надеждою от одного из будущих релизов MongoDB. Однако, благодаря поддержке массивов, простой полнотекстовый поиск можно легко собрать на коленке. Для чего-то более продвинутого придётся использовать сторонние решения, такие как Lucene или Solr. Многие реляционные базы, впрочем, тоже этим грешат.

Транзакции

Транзакций в MongoDB нет. Но есть целых две альтернативы, одна из которых хороша, но ограничена, а вторая похуже, но более гибкая.

Первая --- это наличие нескольких атомарных операций. В большинстве случаев их вполне достаточно. Некоторые из них мы уже видели --- это `$inc` и `$set`. Есть ещё команды наподобие `findAndModify`, которые обновляют объекты и тут же их возвращают.

Вторая используется, когда атомарных операций недостаточно, и называется она «двухфазная запись». Двухфазная запись по отношению к настоящим транзакциям --- это как расстановка ссылок вручную по отношению к джойнам. На сайте MongoDB есть пример, иллюстрирующий довольно распространённый сценарий (перевод денег). Основная идея в том, что мы храним в документе состояние транзакции и выполняем начало-ожидание-подтверждение/откат ручками.

Поддержка вложенных документов и бессхемный дизайн делают двухфазную запись менее болезненной, но это всё равно не верх приятности, особенно для тех, кто впервые с этим сталкивается.

Обработка данных

Для большинства задач обработки данных MongoDB использует MapReduce. Кое-что она умеет «из коробки», но для чего-то серьёзного без MapReduce не обойтись. В следующей главе мы рассмотрим MapReduce поближе. Пока же просто думайте об этом как о мощном и слегка необычном способе выполнить `group by`. Одной из сильных сторон MapReduce является его умение распараллеливать обработку больших объёмов данных. Однако сама MongoDB работает в основном на JavaScript, который однопоточен. Выход? Для обработки больших наборов нужно обратиться к чему-то другому, например, к Hadoop. Благодаря тому, что системы задуманы как дополняющие друг друга, у нас есть адаптер MongoDB-Hadoop.

Разумеется, параллельная обработка данных не является сильной стороной и большинства

реляционных СУБД. В планах на будущие релизы MongoDB есть и улучшение работы с большими наборами данных.

Geospatial

Особенно мощная штука, имеющаяся в MongoDB --- это поддержка гео-индексов. Они позволяют нам хранить координаты (x и y) в документе, а потом искать то, что рядом (`$near`) с нужными нам координатами или внутри (`$within`) определенного круга или квадрата. Это всё лучше объяснять наглядно, поэтому я приглашаю вас посетить 5 minute geospatial interactive tutorial, если вас эта тема заинтересовала.

Инструменты и зрелость

Возможно, вы и так уже всё знаете: MongoDB значительно моложе большинства реляционных СУБД. Это нельзя отрицать. Насколько это важно --- зависит от того, что вы делаете и как вы это делаете. Но, по-хорошему, нельзя игнорировать то, что MongoDB весьма молода, и инструментов для неё пока немного (впрочем, у некоторых зрелых СУРБД дела обстоят не лучше). Например, отсутствие поддержки десятичных чисел с плавающей точкой может стать проблемой (не факт, что непреодолимой), когда системе надо работать с деньгами.

С другой стороны, существуют драйверы для очень многих языков, протокол соверменен и прост, а разработка ведётся с космической скоростью. НИПАНИМАТ Я ВАШ АНГЛИЙСКИЙ, ПЕРЕВЕДИТЕ КТО-НИБУДЬ ЭТУ ФРАЗУ, А?

В этой главе

Эта глава стремится донести мысль о том, что MongoDB в большинстве случаев может заменить реляционные базы данных. Она более проста и прямолинейна, она быстрее и накладывает меньше ограничений на разработчиков. Отсутствие транзакций может оказаться серьёзноым недостатком. Однако, когда кто-либо спрашивает: *«где находится MongoDB в современном мире хранения данных?»*, ответ будет прост: **«точно посередине»**.

Глава 6 --- MapReduce

MapReduce --- это подход к обработке данных, имеющий два важных преимущества перед традиционными решениями. Первое (и самое важное) --- он был разработан для высокой производительности. Теоретически, MapReduce может распараллеливаться, позволяя обрабатывать большие объёмы данных на нескольких ядрах/процессорах/машинах. Как мы уже и говорили, MongoDB пока не справляется с этим. Второе преимущество MapReduce заключается в том, что для обработки вы можете писать настоящий код. По сравнению с возможностями SQL, MapReduce даёт гораздо больше возможностей, и значительно отодвигает тот момент, когда вам придётся обратиться к более специализированным решениям.

MapReduce --- это набирающий популярность паттерн, и его можно использовать почти где угодно. Существуют реализации для C#, Ruby, Java, Python, и многих других языков. Хочу предупредить, что поначалу всё будет казаться непривычным и сложным. Не пугайтесь, найдите время, чтобы повозиться с этим самостоятельно. Это полезно понимать независимо от тог, используете ли вы MongoDB или нет.

Смесь из теории и практики

MapReduce --- это процесс, состоящий из двух этапов. Сначала вы распределяете (map), а затем --- сворачиваете (reduce). При распределении входные документы разбираются на пары ключ=>значение (и то, и другое может быть сложным). Свёртка берёт ключ и массив значений для этого ключа, и вычисляет конечный результат. Мы рассмотрим каждый шаг и то, что выводится на каждом шаге.

Для нашего примера мы будем генерировать отчёт о количестве посещений страницы вебсайта за один день. Это такой *hello world* для MapReduce. Для нашей задачи мы возьмём коллекцию `hits`, имеющую два поля: `resource` и `date`. Мы хотим, чтобы выдача делилась на ресурс, год, месяц, число и количество.

Пусть в `hits` у нас будут следующие данные:

```
resource     date
index        Jan 20 2010 4:30
index        Jan 20 2010 5:30
about        Jan 20 2010 6:00
index        Jan 20 2010 7:00
about        Jan 21 2010 8:00
about        Jan 21 2010 8:30
index        Jan 21 2010 8:30
about        Jan 21 2010 9:00
index        Jan 21 2010 9:30
index        Jan 22 2010 5:00
```

А получить мы хотим вот такой результат:

```
resource  year   month   day   count
```

```
index     2010   1       20    3
about     2010   1       20    1
about     2010   1       21    3
index     2010   1       21    2
index     2010   1       22    1
```

(У этого подхода есть один приятный бонус: когда мы храним вывод, отчёты генерируются быстрее, а увеличение объема данных контролируется: для каждого отслеживаемого ресурса будет создаваться не более 1 документа в день).

Давайте пока сосредоточимся на понимании принципа. В конце главы я дам вам данные и код, с которыми вы сможете поиграть самостоятельно.

Сначала посмотрим на map-функцию. Её задача --- выдавать значение, которое потом может быть свёрнуто. Она может выдавать результат 0 и более раз. В нашем случае (и в большинстве других) она выдаёт одно значение. Представьте себе, что map пробегает по всем документам в коллекции. Мы хотим выдавать для каждого документа ключ, состоящий из ресурса, года, месяца и дня, а значение его всегда должно быть 1:

```
function() {
    var key = {
        resource: this.resource,
        year: this.date.getFullYear(),
        month: this.date.getMonth(),
        day: this.date.getDate()
    };
    emit(key, {count: 1});
}
```

`this` означает текущий обрабатываемый документ. Надеюсь, что изучение вывода поможет вам понять, что здесь происходит. Для данных, предложенных выше, вывод будет выглядеть вот так:

```
{resource: 'index', year: 2010, month: 0, day: 20} => [{count: 1}, {count: 1},
    {count:1}]
{resource: 'about', year: 2010, month: 0, day: 20} => [{count: 1}]
{resource: 'about', year: 2010, month: 0, day: 21} => [{count: 1}, {count: 1},
    {count:1}]
{resource: 'index', year: 2010, month: 0, day: 21} => [{count: 1}, {count: 1}]
{resource: 'index', year: 2010, month: 0, day: 22} => [{count: 1}]
```

Понимание этого промежуточного шага --- это ключ к пониманию MapReduce. Значения из выдачи сгруппированы в массивы по ключу. Программисты .NET и Java могут думать о них как о `IDictionary<object, IList<object>>` (для .NET) или `HashMap<Object, ArrayList>` (для Java).

Давайте слегка изменим нашу map-функцию:

```
function() {
    var key = {resource: this.resource, year: this.date.getFullYear(), month:
        this.date.getMonth(), day: this.date.getDate()};
    if (this.resource == 'index' && this.date.getHours() == 4) {
        emit(key, {count: 5});
    } else {
        emit(key, {count: 1});
    }
}
```

И первый промежуточный вывод станет таким:

```
{resource: 'index', year: 2010, month: 0, day: 20} => [{count: 5}, {count: 1},
    {count:1}]
```

Обратите внимание, как emit генерирует новые значения, которыегруппируются по ключу.

Функция свёртки принимает каждый из этих промежуточных результатов и выводит конечный результат. Наша будет выглядеть вот так:

```
function(key, values) {
    var sum = 0;
    values.forEach(function(value) {
        sum += value['count'];
    });
    return {count: sum};
};
```

И выведет она:

```
{resource: 'index', year: 2010, month: 0, day: 20} => {count: 3}
{resource: 'about', year: 2010, month: 0, day: 20} => {count: 1}
{resource: 'about', year: 2010, month: 0, day: 21} => {count: 3}
{resource: 'index', year: 2010, month: 0, day: 21} => {count: 2}
{resource: 'index', year: 2010, month: 0, day: 22} => {count: 1}
```

Технически, вывод в MongoDB будет таким:

```
_id: {resource: 'home', year: 2010, month: 0, day: 20}, value: {count: 3}
```

Думаю, вы уже заметили, что мы добились того, чего хотели.

Если вы были очень внимательны, то вы, возможно, спросите себя: *«а почему нельзя просто использовать* `sum = values.length`*?»*. Это может показаться более эффективным, особенно

при сложении массива из единичек. Вот только reduce не всегда вызывается на полном и правильном наборе промежуточных данных. Например, вместо:

```
{resource: 'home', year: 2010, month: 0, day: 20} => [{count: 1}, {count: 1},
   {count:1}]
```

reduce может получить:

```
{resource: 'home', year: 2010, month: 0, day: 20} => [{count: 1}, {count: 1}]
{resource: 'home', year: 2010, month: 0, day: 20} => [{count: 2}, {count: 1}]
```

В конце мы всё равно получим 3, но несколько иным путём. Таким образом, reduce всегда должна быть идемпотентна. То есть, сколько бы мы ни вызывали её, она всегда должна возвращать тот же результат.

Мы не будем раскрывать эту тему здесь, но часто встречаются ситуации, когда reduce-методы выполняются цепочкой для выполнения более сложного анализа.

Только практика

Работая с MongoDB, мы выполняем функцию `mapReduce` на коллекции. `mapReduce` принимает аргументами map-функцию, reduce-функцию и директиву вывода. В оболочке мы можем создать JavaScript-функцию и передать её аргументом. Для большинства библиотек придётся писать в аргументе функцию в виде строки (выглядит ужасно). Однако же, давайте введём наш набор данных

```
db.hits.insert({resource: 'index', date: new Date(2010, 0, 20, 4, 30)});
db.hits.insert({resource: 'index', date: new Date(2010, 0, 20, 5, 30)});
db.hits.insert({resource: 'about', date: new Date(2010, 0, 20, 6, 0)});
db.hits.insert({resource: 'index', date: new Date(2010, 0, 20, 7, 0)});
db.hits.insert({resource: 'about', date: new Date(2010, 0, 21, 8, 0)});
db.hits.insert({resource: 'about', date: new Date(2010, 0, 21, 8, 30)});
db.hits.insert({resource: 'index', date: new Date(2010, 0, 21, 8, 30)});
db.hits.insert({resource: 'about', date: new Date(2010, 0, 21, 9, 0)});
db.hits.insert({resource: 'index', date: new Date(2010, 0, 21, 9, 30)});
db.hits.insert({resource: 'index', date: new Date(2010, 0, 22, 5, 0)});
```

Теперь создадим наши map- и reduce-функции (оболочка MongoDB принимает многострочные выражения, вы увидите ... после нажатия Enter, если она ожидает ещё чего-то):

```
var map = function() {
    var key = {resource: this.resource, year: this.date.getFullYear(), month:
       this.date.getMonth(), day: this.date.getDate()};
    emit(key, {count: 1});
};
```

```
var reduce = function(key, values) {
    var sum = 0;
    values.forEach(function(value) {
        sum += value['count'];
    });
    return {count: sum};
};
```

И теперь мы можем использовать их в нашей команде `mapReduce` на коллекции `hits`, вот так:

```
db.hits.mapReduce(map, reduce, {out: {inline:1}})
```

Когда вы выполните это, вы увидите желаемый вывод. Установка `out` в `inline` означает, что вывод `mapReduce` будет отправлен сразу вам. Так можно вывести только результат, не превышающий 16 мегабайт. Вместо этого мы можем сказать `{out: 'hit_stats'}`, и результат будет сохранён в коллекции `hit_stats`:

```
db.hits.mapReduce(map, reduce, {out: 'hit_stats'});
db.hit_stats.find();
```

Когда вы так сделаете, все существующие в `hit_stats` данные будут уничтожены. Если же указать `{out: {merge: 'hit_stats'}}`, значения существующих ключей будут заменены новыми, а новые ключи будут вставлены как новые документы. И наконец, мы можем выводить данные из `reduce`, чтобы осуществлять разные хитрости (наподобие upsert-ов).

В третьем параметре можно передать разные дополнительные опции. Например, мы можем отбирать, сортировать и лимитировать документы которые мы хотим анализировать. Также можно передать метод `finalize`, который будет выполнен на результирующем наборе после `reduce`.

В этой главе

Это первая глава, в которой мы открыли что-то действительно необычное. Если вам это кажется неудобным, помните, что вы всегда можете воспользоваться другими возможностями агрегации данных в MongoDB для более простых сценариев. Как бы то ни было, MapReduce --- это одна из самых серьёзных возможностей MongoDB. Ключ к пониманию того, как писать свои функции распределения и свёртки --- это представлять и понимать, как промежуточные данные должны выглядеть на пути из `map` в `reduce`.

Глава 7 --- Производительность и инструменты

В последней главе мы рассмотрим вопросы производительности, а также несколько полезных инструментов для разработчкиов MongoDB. Мы не будем слишком сильно углубляться в эти темы, но изучим самые важные их части.

Индексы

В самом начале мы встретились с особой коллекцией `system.indexes`, хранящей информацию обо всех индексах нашей базы данных. Индексы в MongoDB работают подобно индексам в РБД: они помогают повысить производительность запросов и сортировки. Индексы создаются с помощью `ensureIndex`:

```
db.unicorns.ensureIndex({name: 1});
```

А удаляются с помощью `dropIndex`:

```
db.unicorns.dropIndex({name: 1});
```

Уникальный индекс будет создан, если во втором параметре выставить `unique` значение `true`:

```
db.unicorns.ensureIndex({name: 1}, {unique: true});
```

Индексировать можно так же и внедрённые поля (используя точечную нотацию) и поля-массивы. Мы можем даже создавать сложные индексы:

```
db.unicorns.ensureIndex({name: 1, vampires: -1});
```

Страница о индексах может рассказать о них ещё кое-что.

Объяснение (Explain)

Чтобы увидеть, используют ли ваши запросы индекс, вы можете использовать `explain` на курсоре:

```
db.unicorns.find().explain()
```

Вывод сообщает нам, что был использован `BasicCursor` (то есть, не индексированный), было просмотрено 12 объектов, сколько времени это заняло, какой индекс (если он есть), и другие полезные сведения.

Если мы изменим запрос так, что будет задействован индекс, мы увидим, что используется `BtreeCursor`, и что индекс действительно использовался:

```
db.unicorns.find({name: 'Pilot'}).explain()
```

Запись в стиле «выстрелил и забыл»

Мы уже упоминали ранее, что по умолчанию в MongoDB запись работает по принципу «выстрелил и забыл». Это неплохо повышает производительность ценой риска потерять данные в случае аварии. Интересным побочным эффектом является так же то, что попытка сохранения записи с неуникальным ключом не возвращает ошибку. Чтобы узнать о том, что что-то пошло не так, следует вызывать `db.getLastError()` после вставки. Многие драйверы скрывают эти шаги и позволяют выполнить безопасную запись, как правило, с помощью лишь дополнительного параметра.

К сожалению, оболочка автоматически осуществляет безопасную вставку, поэтому увидеть это поведение в действии непросто

Разбиение

MongoDB поддерживает разбиение. Это такой подход к масштабированию, когда данные распределяются по множеству серверов. Наивная реализация может складывать данные пользователей с именами, начинающимися на буквы от A до M на один сервер, а данные остальных --- на второй. К счастью, MongoDB использует значительно более умный алгоритм. Изучение этой темы выходит за рамки книги, но вам следует знать, что оно существует и потребуется его изучать, когда вам станет тесно на одном сервере.

Репликация

Репликация в MongoDB работает почти так же, как и в реляционных базах данных. Запись производится на одном сервере (который мастер), который затем синхронизирует себя с одним или более серверами (слейвами). Вы можете управлять тем, будут ли данные читаться со слейвов или нет --- это поможет распределить нагрузку ценой риска получить неактуальные данные. Если мастер перестаёт функционировать, слейв можно делать новым мастером. Опять-таки, репликация --- это тема отдельной книги.

Хотя репликация и может увеличить производительнось (распределяя операции чтения), её основное предназначение --- это увеличение надёжности. Часто репликацию комбинируют с разделением. Например, каждый шард может состоять из мастера и слейва (технически, для этого ещё нужен арбитр, решающий, кто из двух слейвов может стать мастером, но арбитр потребляет мало ресурсов и поэтому он может быть один на несколько шардов).

Статистика

Статистику по базе данных можно собрать, выполнив `db.stats()`. Большая часть информации будет касаться размеров базы данных. Статистику можно так же получить и по коллекции: например, для `unicorns` мы выполним `db.unicorns.stats()`. Опять же, информация будет в основном о размерах коллекции.

Веб-интерфейс

Информация, выводимая MongoDB при старте, содержит ссылку на веб-интерфейс администрирования (вы можете найти её, прокрутив вверх окно терминала до того места, где вы

выполнили mongod). На него можно зайти, введя в браузере http://localhost:28017/. Чтобы сделать его максимально полезным, добавьте строку rest=true в конфигурационный файл и перезапустите mongod. Веб-интерфейс отображает массу полезной информации о текущем состоянии сервера.

Профилировщик

Профилировщик MongoDB можно включить, выполнив:

```
db.setProfilingLevel(2);
```

После этого мы можем выполнить команду:

```
db.unicorns.find({weight: {$gt: 600}});
```

А потом посмотреть в профилировщик:

```
db.system.profile.find()
```

Вывод расскажет нам, что было выполнено и когда, сколько документов было просмотрено и сколько данных возвращено.

Отключается профилировщик вызовом setProfileLevel ещё раз, но с аргументом 0. Другой вариант --- указать 1, тогда профилироваться будут только запросы, выполняющиеся дольше 100 миллисекунд. Либо вы можете указать минимальное вермя, в миллисекундах, во втором параметре:

```
//profile anything that takes more than 1 second
db.setProfilingLevel(1, 1000);
```

Резервное копирование и восстановление

В каталоге bin есть исполняемый файл mongodump. Будучи вызванным без аргументов, он подключится к localhost и скопирует все базы данных в каталог dump. Чтобы увидеть возможные параметры выполните mongodump --help. Чаще всего используются --db DBNAME для копирования определённой базы данных и --collection COLLECTIONNAME для копирования определённой коллекции. После этого можно использовать mongorestore, лежащую всё в той же bin для восстановления сохранённой ранее копии. И снова опции --db и --collection могут использоваться для указания конкретных баз данных и коллекций.

Например, для создания резервной копии базы данных learn в каталог backup, мы выполним (это обычный исполняемый файл, его надо выполнять в обычном терминале, а не в оболочке MongoDB):

```
mongodump --db learn --out backup
```

Чтобы восстановить только коллекцию unicorns, мы потом можем сделать:

```
mongorestore --collection unicorns backup/learn/unicorns.bson
```

Будет так же полезно сказать, что программы `mongoexport` и `mongoimport` позволяют экспортировать и импортировать данные в форматах JSON и CSV. Например, мы можем получить JSON, сделав так:

```
mongoexport --db learn -collection unicorns
```

А CSV --- вот так:

```
mongoexport --db learn -collection unicorns --csv -fields name,weight,vampires
```

Обратите внимание, что `mongoexport` и `mongoimport` в некоторых случаях не смогут корректно представить данные. Для бэкапов пригодны лишь `mongodump` and `mongorestore`.

В этой главе

В этой главе мы рассмотрели различные команды, инструменты и детали производительности MongoDB. Конечно же, это далеко не всё, мы коснулись только самого важного. Индексирование в MongoDB похоже на таковое в реляционных базах, то же самое можно сказать о большинстве инструментов. Однако, в MongoDB многие из них очень полезны и просты в использовании.

Заключение

Вы получили достаточно информации, чтобы начать применять MongoDB в реальных проектах. Мы раскрыли далеко не всё, что касается MongoDB, но следующими приоритетными задачами для вас должны быть собирание всего изученного в кучку и знакомство с драйвером, который вам предстоит использовать. На сайте MongoDB есть очень много полезной информации. А лучшим местом для задавания вопросов является официальная группа пользователей MongoDB

NoSQL появился на свет не только из-за необходимости, но также и из интереса испытать новые подходы. Это доказывает, что наша область интересов расширяется, и если мы не будем пытаться (и иногда ошибаться), мы никогда не достигнем успеха. Это, я считаю, хороший путь для профессионального развития.