Figure 1: The Little MongoDB Book, By Karl Seguin

О книге

Лицензия

Об авторе

Karl Seguin - разработчик с опытом во многих областях и технологиях. Он является .NET- и Ruby-разработчиком с большим опытом работы. Также участвует в open-source проектах, является техническим писателем и нерегулярным докладчик на конференциях. Применительно к MongoDB, он был разработчиком ядра C# MongoDB библиотеки NoRM, написал интерактивный учебник mongly и Mongo Web Admin. Его бесплатный сервис для разработчиков казуальных игр, mogade.com, работает на MongoDB.

Карл также написал книгу The Little Redis Book

Вот его блог: http://openmymind.net, и твиттер: @karlseguin

Благодарности

Особая благодарность Perry Neal за ум, внимание и энтузиазм. Ты оказал мне неоценимую помощь. Спасибо.

Последняя версия

Свежие исходники книги доступны по адресу:

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

О переводе

Перевёл на русский язык @jsmarkus (хабраюзер markpnk).

Корректоры:

- Денис Веселов (хабраюзер progrik).
- Павел Вирский (хабраюзер Paaashka).

Введение

Не моя вина, что книга такая короткая, просто MongoDB легка в освоении.

Технологии развиваются семимильными шагами. Список новых технологий и методологий постоянно растёт. Однако, я всегда придерживался мнения, что фундаментальные технологии, используемые программистами, развиваются не столь стремительно. Можно долгое время обладать актуальными знаниями, не пополняя их. Однако зачастую устоявшиеся технологии заменяются другими с потрясающей скоростью. Внезапные скачки разработок иногда ставят под угрозу устоявшиеся старые технологии.

Яркий пример того - прогресс NoSQL-технологий, приходящих на замену давно известным реляционным базам данных. Вчера еще веб базировался на нескольких известных СУРБД, однако уже сегодня появилось около пяти NoSQL-решений, достойно зарекомендовших себя.

Несмотря на скачкообразность таких изменений, на деле могут понадобиться годы, чтобы они вошли в общепринятую практику. Начальный энтузиазм, как правило, охватывает небольшое число разработчиков и компаний. Решения оттачиваются, извлекаются уроки, - и, видя, что новая технология развивается, остальные пробуют применять её для своих нужд. Опять же, это касается сферы NoSQL, где множество технологий являются не столько прямой заменой более традиционным механизмам хранения, сколько являются решениями специальных проблем, в дополнение к тому, что можно ожидать от традиционных систем.

Принимая во внимание всё вышеизложенное, мы должны уяснить, чем же является NoSQL. Это широкий термин, в который означает разное для разных людей. Лично я использую его в широком смысле, чтобы обозначить систему, участвующую в хранении данных. С другой стороны NoSQL для меня означает убежденность в том, что задача хранения данных не возлагается на одну большую систему. В то время, как производители большинства баз данных исторически пытались позиционировать свой софт, как решение ``всё в одном'', NoSQL стремится к меньшему уровню ответственности - когда для определенных задач может быть выбран такой инструмент, который бы решал именно эту задачу наилучшим образом. К примеру, ваш NoSQL-стек может эффективно использовать реляционные базы данных, как например MySQL, однако он также может включать в себя Redis - для организации хранения записей key-value или Hadoop - для интенсивной обработки данных. Проще говоря, NoSQL - это открытая технология, состоящая из альтернативных, существующих и дополнительных шаблонов управления данными.

Удивительно, но MongoDB подходит под все эти определения. Как документ-ориентированная СУБД, Mongo - это довольно-таки обобщенное NoSQL решение. Ее можно рассматривать, как альтернативу реляционным СУБД. Подобно реляционным СУБД, она также может выигрышно дополняться более специализированными NoSQL решениями. У MongoDB есть как достоинства, так и недостатки, о них мы поговорим в следующих частях книги.

Как вы уже заметили, термины MongoDB и Mongo используются как синонимы.

Приступая к работе

Большая часть книги освещает базовые возможности MongoDB. Поэтому нам понадобится консоль MongoDB. Консоль будет использоваться для учебных и административных задач, а в коде мы будем пользоваться драйвером MongoDB.

Мы подошли к первому, что надо знать о MongoDB: к её драйверам. У MongoDB есть множество официальных драйверов для различных языков. Их можно рассматривать как драйверы уже привычных реляционных БД. На их основе сообщество разработчиков построило множество высокоуровневых драйверов - для определенных языков и фреймворков. Например, NoRM это библиотека для C#, реализующая LINQ, а MongoMapper для Ruby, с поддержкой ActiveRecord. Программировать напрямую, используя низкоуровневые драйверы MongoDB, или же с применением высокоуровневых библиотек - решайте сами. Я подробно остановился на этом, потому что множество новичков бывают сбиты с толку наличием как официальных драйверов, так и разрабатываемых сообществом - первые нацелены на базовую коммуникацию с Mongo, в то время как вторые - больше на внедрение в конкретные языки и фреймворки.

По мере чтения старайтесь воспроизводить демонстрируемые примеры, а также изучать вопросы, которые могут при этом возникнуть. Поднять у себя MongoDB просто, нам понадобится несколько минут, чтобы все настроить.

1. Зайдите на официальную страницу скачивания и скачайте бинарные файлы из первой строки (рекомендованную стабильную версию) для операционной системы, которую вы используете. Для разработки можно использовать как 32-, так и 64-разрядную версию.
2. Распакуйте архив (куда угодно) и перейдите в папку `bin`. Пока ничего не запускайте, но запомните, что `mongod` - это сервер, а `mongo` - клиентская консоль - вот два исполняемых файла, с которыми нам чаще всего предстоит работать.
3. Создайте новый файл в папке `bin` и назовите его `mongodb.config`
4. Добавьте в mongodb.config одну строку: `dbpath=ПУТЬ_КУДА_ХОТИТЕ_СОХРАНИТЬ_ФАЙЛЫ_БАЗЫ_ДАННЫХ`. Например, в Windows можно написать `dbpath=c:\mongodb\data` а в Linux - `dbpath=/etc/mongodb/data`.
5. Убедитесь, что указанный вами путь `dbpath` существует.
6. Запустите mongod с параметром `--config /path/to/your/mongodb.config`.

Для пользователей Windows, например, если вы распаковали скачанный файл в `c:\mongodb\` и создали папку `c:\mongodb\data\`, то в `c:\mongodb\bin\mongodb.config` следует указать `dbpath=c:\mongodb\data\`. Теперь можно запускать `mongod` из командной строки с помощью команды `c:\mongodb\bin\mongod --config c:\mongodb\bin\mongodb.config`.

Папку `bin` можно для удобства добавить в переменную окружения `PATH`. Для пользователей MacOSX и Linux инструкции практически те же самые. Всё, что нужно сделать - это просто изменить пути.

Надеюсь, теперь MongoDB у вас установлена и запущена. Если есть ошибки - внимательно читайте сообщения в консоли - сервер подробно и ясно выводит диагностические сообщения.

Теперь, чтобы подключиться к запущенному серверу, можете запустить `mongo` (без *d* в конце). Попробуйте ввести `db.version()`, чтобы убедиться, что все в порядке. Если всё нормально - вы увидите номер версии вашего сервера.

Глава 1 - Основы

Начнем мы с изучения основных механизмов работы с MongoDB. Это самое основное, что понадобится для понимания MongoDB, но также мы коснемся высокоуровневых вопросов - о том, где применима MongoDB.

Для начала нужно понять шесть основных концепций.

1. MongoDB - концептуально то же самое, что обычная, привычная нам база данных (или в терминологии Oracle - схема). Внутри MongoDB может быть ноль или более баз данных, каждая из которых является контейнером для прочих сущностей.
2. База данных может иметь ноль или более `коллекций'. Коллекция настолько похожа на традиционную ``таблицу'', что можно смело считать их одним и тем же.
3. Коллекции состоят из нуля или более ``документов''. Опять же, документ можно рассматривать как ``строку''.
4. Документ состоит из одного или более ``полей'', которые - как можно догадаться - подобны ``колонкам''.
5. ``Индексы'' в MongoDB почти идентичны таковым в реляционных базах данных.
6. ``Курсоры'' отличаются от предыдущих пяти концепций, но они очень важны (хотя порой их обходят вниманием) и заслуживают отдельного обсуждения. Важно понимать, что когда мы запрашиваем у MongoDB какие-либо данные, то она возвращает курсор, с которыми мы можем делать все что угодно - подсчитывать, пропускать определенное число предшествующих записей - при этом не загружая сами данные.

Подводя итог, MongoDB состоит из ``баз данных'', которые состоят из ``коллекций''. ``Коллекции'' состоят из ``документов''. Каждый ``документ'' состоит из ``полей''. ``Коллекции'' могут быть проиндексированы, что улучшает производительность выборки и сортировки. И наконец, получение данных из MongoDB сводится к получению ``курсора'', который отдает эти данные по мере надобности.

Вы можете спросить - зачем придумывать новые термины (коллекция вместо таблицы, документ вместо записи и поле вместо колонки)? Не излишнее ли это усложнение? Ответ в том, что эти термины, хоть и близки своим ``реляционным'' аналогам, но не полностью идентичны им. Основное различие в том, что реляционные базы данных определяют ``колонки'' на уровне ``таблицы'', в то время как документ-ориентированные базы данных определяют ``поля'' на уровне ``документа''. Это значит, что любой документ внутри коллекции может иметь свой собственный уникальный набор полей. В этом смысле коллекция ``глупее'' чем таблица, тогда как документ имеет намного больше информации, чем строка.

Хоть это и важно понять, не волнуйтесь, если не сможете сразу. После нескольких вставок вы увидите, что имеется в виду. В конечном счете дело в том, что коллекция

не содержит информации о структуре содержащихся в ней данных. Информацию о полях содержит каждый отдельный документ. Преимущества и недостатки этого станут понятны из следующей главы.

Приступим. Запустите сервер `mongod` и консоль `mongo`, если еще не запустили. Консоль работает на JavaScript. Есть несколько глобальных команд, например `help` или `exit`. Команды, которые вы запускаете применительно к текущей базе данных исполняются у объекта `db`, например `db.help()` или `db.stats()` . Команды, которые вы запускаете применительно к конкретной коллекции, исполняются у объекта `db.ИМЯ_КОЛЛЕКЦИИ`, например `db.unicorns.help()` или `db.unicorns.count()` .

Введите `db.help()` и получите список команд, которые можно выполнить у объекта `db`.

Заметка на полях. Поскольку консоль интерпретирует JavaScript, если вы попытаетесь выполнить метод без скобок, то в ответ получите тело метода, но он не выполнится. Не удивляйтесь, увидев `function (...){`, если случайно сделаете так. Например, если введёте `db.help` (без скобок), вы увидите внутренне представление метода `help`.

Сперва для выбора базы данных воспользуемся глобальным методом `use` - введите `use learn`. Неважно, что база данных пока еще не существует. В момент создания первой коллекции создастся база данных `learn`. Теперь, когда вы внутри базы данных, можно вызывать у неё команды, например `db.getCollectionNames()`. В ответ увидите пустой массив (`[ ]`). Поскольку коллекции бесструктурны *(в оригинале ``schema-less''. Здесь и далее - прим. перев.)*, мы не обязаны создавать их явно. Мы просто можем вставить документ в новую коллекцию. Чтобы это сделать, используйте команду `insert`, передав ей вставляемый документ:

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

Данная строка выполняет метод `insert` (``вставить'') в коллекцию `unicorns`, передавая ему единственный аргумент. MongoDB у себя внутри использует бинарный сериализированный JSON формат. Снаружи это означает, что мы широко используем JSON, как, например, в случае с нашими параметрами. Если теперь выполнить `db.getCollectionNames()`, мы увидим две коллекции: `unicorns` и `system.indexes`. `system.indexes` создается в каждой базе данных и содержит в себе информацию об индексах этой базы.

Теперь у коллекции `unicorns` можно вызвать метод `find`, который вернет список документов:

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

Заметьте, что кроме данных, которые мы задавали, появилось дополнительное поле `_id` . Каждый документ должен иметь уникальное поле `_id`. Можете генерировать его сами

или позволить MongoDB самой сгенерировать для вас ObjectId. В большинстве случаев вы скорее всего возложите эту задачу на MongoDB. По умолчанию _id - индексируемое поле, вследствие чего и создается коллекция system.indexes. Давайте взглянем на system.indexes:

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

Вы увидите имя индекса, базы данных и коллекции, для которой индекс был создан, а также полей, которые включены в него.

Вернемся к обсуждению бесструктурных коллекций. Давайте вставим кардинально отличный от предыдущего документ в unicorns, вот такой:

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

И снова воспользуемся find для просмотра списка документов. Теперь, узнав чуть больше, мы можем обсудить это интересное поведение MongoDB, но, надеюсь, вы уже начинаете понимать, почему традиционная терминология здесь не совсем применима.

Осваиваем селекторы

В дополнение к изученным ранее шести концепциям, есть еще один немаловажный практический аспект MongoDB, который следует освоить, прежде чем переходить к более сложным темам: это - селекторы запросов. Селектор запросов MongoDB аналогичен предложению where SQL-запроса. Как таковой он используется для поиска, подсчёта, обновления и удаления документов из коллекций. Селектор - это JSON-объект, в простейшем случае это может быть даже {}, что означает выборку всех документов (аналогичным образом работает null). Если нам нужно выбрать всех единорогов (англ. ``unicorns'') женского рода, можно воспользоваться селектором {gender:'f'}.

Прежде, чем мы глубоко погрузимся в селекторы, давайте сначала создадим немного данных, с которыми будем экспериментировать. Сперва давайте удалим всё, что до этого вставляли в коллекцию unicorns с помощью команды: db.unicorns.remove() (поскольку мы не передали селектора, произойдет удаление всех документов). Теперь давайте произведем следующие вставки, чтобы получить данные для дальнейших экспериментов (можете скопировать и вставить это в консоль):

```
db.unicorns.insert({name: 'Horny', dob: new Date(1992,2,13,7,47), loves: ['carrot','papaya'], weight:
db.unicorns.insert({name: 'Aurora', dob: new Date(1991, 0, 24, 13, 0), loves: ['carrot', 'grape'], weig
db.unicorns.insert({name: 'Unicrom', dob: new Date(1973, 1, 9, 22, 10), loves: ['energon', 'redbull'],
db.unicorns.insert({name: 'Roooooodles', dob: new Date(1979, 7, 18, 18, 44), loves: ['apple'], weight:
db.unicorns.insert({name: 'Solnara', dob: new Date(1985, 6, 4, 2, 1), loves:['apple', 'carrot', 'choco
db.unicorns.insert({name:'Ayna', dob: new Date(1998, 2, 7, 8, 30), loves: ['strawberry', 'lemon'], wei
```

```
db.unicorns.insert({name:'Kenny', dob: new Date(1997, 6, 1, 10, 42), loves: ['grape', 'lemon'], weight
db.unicorns.insert({name: 'Raleigh', dob: new Date(2005, 4, 3, 0, 57), loves: ['apple', 'sugar'], weigh
db.unicorns.insert({name: 'Leia', dob: new Date(2001, 9, 8, 14, 53), loves: ['apple', 'watermelon'], we
db.unicorns.insert({name: 'Pilot', dob: new Date(1997, 2, 1, 5, 3), loves: ['apple', 'watermelon'], wei
db.unicorns.insert({name: 'Nimue', dob: new Date(1999, 11, 20, 16, 15), loves: ['grape', 'carrot'], wei
db.unicorns.insert({name: 'Dunx', dob: new Date(1976, 6, 18, 18, 18), loves: ['grape', 'watermelon'], w
```

Теперь, когда данные созданы, можно приступать к освоению селекторов. `{поле: значение}` используется для поиска всех документов, у которых `поле` равно `значение`. `{поле1: значение1, поле2: значение2}` работает как логическое И. Специальные операторы `$lt`, `$lte`, `$gt`, `$gte` и `$ne` используются для выражения операций ``меньше'', ``меньше или равно'', ``больше'', ``больше или равно'', и ``не равно''. Например, чтобы получить всех самцов единорога, весящих более 700 фунтов, мы можем написать:

```
db.unicorns.find({gender: 'm', weight: {$gt: 700}})
//или (что не полностью эквивалентно, но приведено здесь в демонстрационных целях)
db.unicorns.find({gender: {$ne: 'f'}, weight: {$gte: 701}})
```

Оператор `$exists` используется для проверки наличия или отсутствия поля, например:

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

Вернет единственный документ. Если нужно ИЛИ вместо И, мы можем использовать оператор `$or` и присвоить ему массив значений, например:

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

Вышеуказанный запрос вернет всех самок единорогов, которые или любят яблоки, или любят апельсины, или весят менее 500 фунтов.

В нашем последнем примере произошло кое-что интересное. Вы заметили - поле `loves` это массив. MongoDB поддерживает массивы как объекты первого класса. Это потрясающе удобная возможность. Начав это использовать, вы удивитесь, как вы раньше жили без этого. Самое интересное это та простота, с которой делается выборка по значению массива: `{loves: 'watermelon'}` вернет нам все документы, у которых `watermelon` является одним из значений поля `loves`.

Это еще не все операторы. Самый гибкий оператор - `$where`, позволяющий нам передавать JavaScript для его выполнения на сервере. Это описано в разделе Сложные запросы на сайте MongoDB. Мы изучили основы, которые нам нужны для начала работы. Это также то, что вы будете использовать большую часть времени.

Мы видели, как эти селекторы могут быть использованы с командой `find`. Они также могут быть использованы с командой `remove`, которую мы кратко рассмотрели, командой `count`, на которую мы пока не взглянули, но которую вы скорее всего изучите, и командой `update`, с которой в дальнейшем мы проведем большую часть времени.

`ObjectId`, сгенерированный MongoDB для поля `_id`, подставляется в селектор следующим образом:

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

В этой главе

Мы пока еще не рассматривали команду `update` или более интересные вещи, которые можно сделать с помощью `find`. Однако мы подняли MongoDB, кратко изучили команды `insert` и `remove` (изучив практически всё, что о них можно изучить) . Мы также начали исследовать `find` и узнали что такое селекторы MongoDB. Это неплохо для начала, и основы для дальнейшего изучения заложены. Верите или нет, но вы уже изучили практически всё, что нужно знать о MongoDB - настолько она проста и легка в изучении. Я настоятельно рекомендую вам поэкспериментировать с вашими данными, прежде, чем можно будет двигаться дальше. Вставьте несколько новых документов - возможно в новые коллекции - и поэкспериментируйте с селекторами. Используйте `find`, `count` и `remove`. После нескольких ваших собственных попыток вещи, казавшиеся непонятными, станут на свои места.

Глава 2 - Обновление

В первой главе мы изучили три из четырёх операций CRUD (create, read, update and delete). Эта глава посвящена четвёртой: `update`. У `update` имеются некоторые особенности, вот почему мы посвящаем этому целую главу.

Обновление данных: замена и $set

В простейшей форме, `update` принимает 2 аргумента: селектор (where) для выборки и то, чем обновить соответствующее поле. Чтобы Roooooodles прибавил в весе, используем следующий запрос:

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

(Если в ходе экспериментов вы удалили данные из ранее созданной коллекции `unicorns`, сделайте всем документам `remove`, и вставьте их заново с помощью кода из главы 1)

В реальной жизни, конечно, следует обновлять документы, выбирая их по `_id`, однако, поскольку я не знаю какой `_id` MongoDB сгенерировала для вас, будем выбирать по имени - `name`. Теперь, давайте взглянем на обновленную запись:

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

Вот и первый сюрприз, который нам преподнёс `update`. Документ не найден, поскольку второй параметр используется для **полной замены** оригинала. Иными словами, `update` нашел документ по имени и заменил его целиком на новый документ (свой второй параметр). Вот в чём отличие от SQL-команды `UPDATE`. Иногда это идеальный вариант, который может использоваться для некоторых действительно динамических обновлений. Однако, если вам нужно всего лишь изменить пару полей, лучше всего использовать модификатор `$set`:

```
db.unicorns.update({weight: 590}, {$set: {name: 'Roooooodles', dob: new Date(1979, 7, 18, 18, 44), love
```

Это восстановит утерянные ранее поля. Поле `weight` не перезапишется, поскольку мы его не передали в запрос. Теперь, если выполнить:

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

мы получим ожидаемый результат. Таким образом, в первом примере правильно было бы обновить `weight` следующим образом:

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

Модификаторы обновления

Кроме `$set` можно использовать и другие модификаторы для разных изящных вещей. Все эти модификаторы обновления действуют над полями - так что ваш документ не окажется перезаписан целиком. Например, модификатор `$inc` служит для того, чтобы изменить поле на положительную (увеличить) или отрицательную (уменьшить) величину . Например, если единорог Pilot был ошибочно награжден за убийство пары лишних вампиров, мы можем исправить эту ошибку следующим образом:

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

Если Aurora внезапно пристрастилась к сладостям, мы можем добавить соответствующее значение к ее полю `loves` с помощью модификатора `$push`:

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

Информацию об остальных модификаторах можно найти в разделе Обновление на сайте MongoDB.

Обновление/вставка

Один из приятных сюрпризов операции обновления - это возможность обновления/вставки *(`upsert` от `update` - обновить и `insert` - вставить)* Обновление/вставка обновляет документ, если он найден, или создаёт новый - если не найден. Обновление/вставка - полезная вещь в некоторых случаях; когда столкнётесь с подобным, сразу поймёте. Чтобы разрешить вставку при обновлении, установите третий параметр в `true`.

Пример из жизни - счетчик посещений для веб-сайта. Если мы хотим в реальном времени видеть количество посещений страницы, мы должны посмотреть, существует ли запись, и - в зависимости от результата - выполнить `update` либо `insert`. Если опустить (или установить в `false`) третий параметр, следующий пример не сработает:

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

Однако, если разрешить вставку при обновлении, результаты будут иными:

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

Поскольку документы с полем `page`, равным `unicorns`, не существуют, то будет создан новый документ. Если выполнить это вторично, существующий документ будет обновлён, и поле `hits` увеличится до 2.

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

Множественные обновления

Последний сюрприз метода `update` - это, то что он по умолчанию обновляет лишь один документ. До сих пор это было логично в случае с уже рассмотренными примерами. Однако, если выполнить что-нибудь вроде:

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

, то вы очевидно будете ожидать, что все единороги будут привиты *(vaccinated)*. Чтобы это сработало, нужно установить четвертый параметр в `true`:

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

В этой главе

Эта глава завершила введение в основные CRUD операции над коллекциями. Мы детально рассмотрели `update` и увидели три его интересных режима работы. Во-первых, в отличие от SQL-команды `UPDATE`, в MongoDB `update` заменяет документ целиком. Из-за этого модификатор `$set` очень полезен. Во-вторых, `update` поддерживает интуитивно простое обновление/вставку, которое особенно полезно с модификатором `$inc`. И, наконец, в-третьих, по умолчанию, `update` обновляет лишь первый найденный документ.

Помните, что мы рассматриваем MongoDB с точки зрения её консоли. Используемые вами драйверы и библиотеки могут иметь иное поведение и реализовывать иной API. Например, драйвер для Ruby сливает два параметра в один хэш: `{:upsert => false, :multi => false}`.

Глава 3 - Осваиваем Find

В главе 1 мы вкратце рассмотрели команду `find`. Однако, `find` - это не только селекторы. Как уже упоминалось, результатом `find` является курсор. Пришло время рассмотреть это детальнее.

Выбор полей

Прежде чем переходить к курсорам, следует знать, что `find` принимает второй необязательный параметр. Это - список полей, которые мы хотим получить. Например, мы можем получить все имена единорогов следующим запросом:

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

Поле `_id` по умолчанию возвращается всегда. Мы можем явным способом исключить его, указав `{name:1, _id: 0}`.

За исключением поля `_id`, нельзя смешивать включения и исключения полей. Задумавшись, можно понять, зачем так сделано. Можно или хотеть включить или хотеть наоборот - исключить определенные поля явным образом.

Сортировка

Я уже несколько раз упомянул, что `find` возвращает курсор, который исполняется отложенно - по мере необходимости. Однако, вы уже без сомнения могли видеть, что `find` исполняется мгновенно. Такое поведение характерно только для консоли. Можно пронаблюдать за истинным поведением курсоров, взглянув на любой из методов, который мы можем присоединить к `find`. Первым из них будет `sort`. Синтаксис `sort` примерно такой же, как у выбора полей, который мы видели в предыдущем разделе. Мы указываем поля, по которым надо сортировать, используя `1` для сортировки по возрастанию и `-1` для сортировки по убыванию. Например:

```
//сортируем по весу - от тяжёлых к лёгким единорогам
db.unicorns.find().sort({weight: -1})

//по имени вампира, затем по числу убитых вампиров:
db.unicorns.find().sort({name: 1, vampires: -1})
```

Подобно реляционной базе данных, MongoDB может использовать индексы для сортировки. Детальнее мы рассмотрим индексы несколько позже. Однако следует знать, что без индекса MongoDB ограничивает размер сортируемых данных. Если вы попытаетесь отсортировать большой объем данных, не используя индекс, вы получите ошибку. Некоторые считают это ограничением. Хотя я думаю, что и другим базам данных не мешало бы запрещать выполнение неоптимальных запросов. (Я не стану превращать каждый недостаток MongoDB в её достоинство, однако я сталкивался с большим числом неоптимальных баз данных, которым очень не хватало подобного режима строгой проверки.)

Разбиение на страницы

Разбиение на страницы может быть осуществлено с помощью методов `limit` и `skip`. Чтобы получить второго и третьего по весу единорога, можно выполнить:

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

Используя `limit` вместе с `sort` можно избежать проблем с сортировкой по неиндексированным полям.

Count

Консоль позволяет выполнить `count` прямо над коллекцией:

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

На практике же `count` - это метод курсора, консоль просто обеспечивает удобное сокращение. С драйверами, не поддерживающим подобного сокращения, нужно писать что-то вроде этого (конечно, и в консоли тоже так можно):

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

In This Chapter

Довольно просто пользоваться `find` и курсорами. Есть еще несколько дополнительных команд, которые мы либо рассмотрим позже, либо не рассмотрим вообще (так как они применяются лишь в граничных случаях), но теперь, я думаю, вы доолжны уже освоиться в работе с консолью `mongo` и пониманием основных принципов MongoDB.

Глава 4 - Моделирование данных

Давайте сменим тему и поговорим о более абстрактных концепциях MongoDB. Довольно просто объяснять новые термины и новый синтаксис. Гораздо сложнее говорить о моделировании в терминах новой парадигмы. Смысл в том, что большинство из нас привыкли пробовать любую новую технологию, моделируя реальные задачи. Мы поговорим об этом, но в конечном счете вы должны попрактиковаться и изучить реальный код.

Когда речь заходит о моделировании данных, то документ-ориентированные базы данных не настолько сильно отличаются от реляционных, как другие NoSQL-решения. Существующие различия не столь велики, однако это не уменьшает их важности.

Отсутствие JOIN-ов

Первое и самое фундаментальное различие, с которым вам надо свыкнуться, это отсутствие у MongoDB аналога конструкции JOIN. Неизвестно почему именно MongoDB не поддерживает JOIN-синтаксиса, однако точно можно сказать, что JOIN-ы не масштабируемы. Это значит, что когда вы начнёте разделять данные горизонтально, вам всё равно придется выполнять JOIN-ы на клиенте (которым является сервер приложений). Независимо от причин, факт остаётся фактом: данные реляционны по своей природе, но MongoDB не поддерживает JOIN-ов.

Мы должны делать JOIN-ы вручную, в коде своего приложения. По существу, мы должны делать второй запрос, чтобы найти связанные данные. Создание данных тут не сильно отличается от создания внешних ключей в реляционных базах. Теперь давайте от единорогов (`unicorns`) перейдём к сотрудникам (`employees`). Первым делом создадим сотрудника (я явным образом привожу здесь `_id`, чтобы наши примеры выполнялись как задумано)

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

Теперь добавим пару сотрудников и сделаем `Leto` их менеджером:

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

(стоит повторить, что `_id` может быть любым уникальным значением. Поскольку в жизни вы скорее всего станете использовать `ObjectId`, мы также здесь используем его.)

Чтобы найти всех сотрудников, принадлежащих Leto, выполним просто:

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

Никакой магии. В худших случаях отсутствие JOIN-ов чаще всего потребует дополнительного запроса (как правило индексированного).

Массивы и вложенные документы

Но тот факт, что у MongoDB нет JOIN-ов еще не означает, что у неё не припасено пару козырей в рукаве. Помните, как мы вкратце поведали ранее о поддержке в MongoDB массивов, как объектов первого класса? Оказывается, что она чертовски удобна, когда требуется смоделировать отношения ``один-ко-многим'' или ``многие-ко-многим''. Например, если у сотрудника есть несколько менеджеров, мы просто можем сохранить их в виде массива:

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

А самое интересное, что в одних документах `manager` можно сделать скалярным значением, а в других - массивом. А наш предыдущий запрос `find` сработает в обоих случаях:

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

Вскоре вы убедитесь, что массивы значений намного удобнее в использовании, нежели таблицы связи ``многие-ко-многим''.

Кроме массивов MongoDB также поддерживает вложенные документы. Попробуйте вставить документ со вложенным документом, например:

```
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d734"), name: 'Ghanima', family: {mother: 'Ch
```

Вложенные документы можно запрашивать с помощью точечной нотации:

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

Мы кратко обсудим, где могут использоваться вложенные документы, и как их следует применять.

DBRef

MongoDB поддерживает понятие под названием `DBRef`, которое является соглашением, принятым во многих драйверах. Когда драйвер видит `DBRef`, он может автоматически получить связанный документ. `DBRef` включает в себя коллекцию и `_id` документа, на который он ссылается. Это означает следующее - документы из одной и той же коллекции могут ссылаться на другие документы из различных коллекций. То есть документ 1 может ссылаться на документ из коллекции `managers`, в то же время документ 2 может ссылаться на документ из коллекции `employees`.

Денормализация

Еще одна альтернатива использованию JOIN-ов - денормализация. Исторически денормализация использовалась для оптимизации производительности, или когда с данных (например, журнала аудита) необходимо было иметь возможность делать снимок. Однако с быстрым ростом NoSQL решений, многие из которых лишены JOIN-ов, денормализация стала в порядке вещей. Это не означает, что нужно дублировать всё подряд в любых документах. Можно остерегаться дублирования данных, а можно соответствующим образом продумать архитектуру своей базы.

К примеру, мы разрабатываем форум. Традиционный путь ассоциировать пользователя с его постом - это колонка `userid` в таблице `posts`. с такой моделью нельзя отобразить список постов без дополнительного извлечения данных (JOIN) из таблицы пользователей. Возможное решение - хранить имя пользователя (`name`) вместе с `userid` для каждого поста. Можно также вставлять небольшой встроенный документ, например, `user: {id: ObjectId('Something'), name: 'Leto'}`. Да, если позволить пользователям изменять своё имя, нам придётся обновлять каждый документ (пост) - это один лишний запрос.

Не всем легко приспособиться к такому подходу. Во многих случаях даже не имеет смысла этого делать. Все же не бойтесь экспериментировать с таким подходом. Иногда это бывает полезным - чуть ли не единственным правильным - решением.

Что выбрать?

Также полезной стратегией в случаях отношения ``один-ко-многим'' или ``многие-ко-многим'' является массив идентификаторов. Бытует мнение, что `DBRef` используется не так часто, но конечно вы можете поэкспериментировать с ним. Обычно начинающие разработчики не уверены что подойдёт им лучше - вложенные документы или `DBRef`.

Во-первых, следует помнить, что одиночный документ ограничен в размере до 4 мегабайт. Факт ограничения (пусть и такого щедрого) размера документа дает понимание о том, как их следует использовать. Теперь понятно, что большинство разработчиков склоняются к использованию заданных вручную ссылок. Вложенные документы используются часто, но для небольших объёмов данных, если их желательно всегда извлекать вместе с родительским документом. Примером из жизни может быть документ `accounts`, сохраняемый с каждым пользователем, например:

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

Это не означает, что можно недооценивать мощь вложенных документов, либо отбрасывать их, как мелкую, второстепенную утилиту. Намного проще живётся, когда структура ваших данных напрямую отображает структуру ваших объектов. Особенно ценным является то, что MongoDB позволяет запрашивать и индексировать поля вложенных документов.

Мало или много коллекций

Учитывая то, что коллекции не привязывают нас к конкретной схеме, вполне возможно обойтись одной коллекцией, имеющей документы разной структуры. Построенные на MongoDB системы, с которыми мне приходилось сталкиваться, как правило, были похожи на реляционные базы данных. Другими словами, то, что являлось бы таблицей в реляционной базе данных, скорее всего реализуется, как коллекция в MongoDB (таблицы-связки ``многие-ко-многим'' являются важным исключением).

Дело принимает интересный оборот, если воспользоваться вложенными документами. Пример, который первым делом приходит на ум, это блог. Допустим, есть коллекция `posts` и коллекция `comments`, и каждый пост должен иметь вложенный массив комментариев. Если оставить в стороне ограничение 4Мб (``Гамлет'' на английском едва дотягивает до 200 килобайт, насколько же должен быть популярным ваш блог?), большинство разработчиков предпочитают разделять сущности. Так понятнее и яснее.

Нет какого бы то ни было строгого правила (ну, кроме 4MB). Поэкспериментируйте с различными подходами, и вам станет ясно, что будет правильнее, а что - нет.

В этой главе

Целью этой главы было представить некоторые полезные рекомендации для моделирования данных в MongoDB. Если угодно, стартовую точку. Моделирование в документ-ориентированных системах отличается от такового в реляционных, но не так уж сильно. Здесь намного больше гибкости, но есть одно ограничение, хотя для разработки новой системы это подходит, как правило, неплохо. Не выходит только у тех, кто не пробует.

Глава 5 - Когда использовать MongoDB

К этому моменту у вас должно сформироваться понимание MongoDB, достаточное для того, чтобы понять, где она может вписаться в вашу существующую систему. Есть так много новых, конкурирующих технологий хранения данных, что легко растеряться в выборе, какую же из них использовать.

Для меня самым большим уроком, не имеющим, впрочем, ничего общего с MongoDB, стало то, что не обязательно полагаться на единственное решение для работы с данными. Естественно, единственное решение имеет очевидные преимущества, и для многих - если не для большинства - проектов таковое является разумным подходом. Смысл не в том, что вы *должны*, а скорее в том, что вы *можете* использовать различные технологии. Только вы знаете, перевешивают ли преимущества от внедрения нового решения возможные издержки.

С учетом сказанного, я надеюсь, что виденное вами ранее позволило вам расценивать MongoDB в качестве общего решения. Пару раз упоминалось, что документ-ориентированные базы данных имеют много общего с реляционными. Таким образом, чтобы не ходить вокруг да около, позвольте просто заявить, что MongoDB может рассматриваться как прямая альтернатива реляционным базам данных. В то время, как Lucene можно рассматривать, как расширение реляционных баз полнотекстовым индексом, а Redis - как персистентное хранилище ключ-значение, MongoDB - это центральный репозиторий для ваших данных.

Заметьте, я не называю MongoDB *заменой* реляционных баз, это скорее *альтернатива*. Это инструмент, который может делать то же, что могут делать множество прочих. Кое-что - лучше, кое-что - нет. Проанализируем это чуть позже.

Бесструктурность

Часто рекламируемым преимуществом документ-ориентированных баз данных является то, что они бесструктурны. Это делает их гораздо более гибкими, нежели традиционные реляционные базы данных. Я согласен, что бесструктурность хороша, но только не в качестве упоминаемого многими главного преимущества.

Часто бесструктурность видится как хаотичная организация данных. Есть домены и наборы данных, которые и правда очень трудно смоделировать в терминах обычной реляционной базы данных, но я рассматриваю их скорее как граничные случаи. Бесструктурность заманчива, однако большая часть данных должна быть хорошо структурированной. Конечно, иногда это может быть удобно, особенно для добавления нового функционала, однако на деле это можно решить и добавлениями новых необязательных полей.

Для меня настоящее преимущество бесструктурной архитектуры - это отсутствие установки и сведённые к минимуму расхождения с ООП. Особенно это чувствуется при работе со статически типизированными языками. Я работал с MongoDB как в C#, так и в Ruby - разница бросается в глаза. Динамизм Ruby и популярная реализация ActiveRecord уже ощутимо сокращают расхождение объектной и реляционной моделей *(object-relational*

impedance mismatch). Это не означает, что MongoDB - плохое решение для Ruby, напротив. Скорее я думаю, что большинство Ruby-разработчиков видят MongoDB как небольшое улучшение, в то время как разработчики, пишущие на C# или Java, видят пропасть разделяющую MongoDB и их подход к манипулированию данными.

Подумайте об этом с точки зрения разработчика драйверов. Вам надо сохранить объект? Сериализируйте его в JSON (на самом деле в BSON, но это почти одно и то же) и отправьте в MongoDB. Нет никакого маппинга свойств или типов. Эта простота определённо должна подходить вам, как конечному разработчику.

Запись

Область, для которой MongoDB особенно подходит, - это логгирование. Есть два аспекта MongoDB, которые делают запись быстрой. Во-первых, можно отправить команду записи и продолжить работу, не ожидая её возврата и действительной свершившейся записи. Во-вторых, с появлением в версии 1.8 журналирования и некоторыми улучшениями, сделанными в версии 2.0, стало возможно контролировать поведение записи с учётом целостности данных. Эти параметры, в дополнение к тому, сколько серверов должны получить ваши данные, прежде чем запись будет считаться успешной, настраиваются на уровне отдельной записи, что дает вам большую степень контроля над выполнением записи данных и их долговечностью.

Кроме указанных факторов производительности, при логгировании как раз может оказаться полезной гибкая структура данных. Наконец, в MongoDB есть такое понятие, как ограниченная коллекция *(capped collection)*. До сих пор мы создавали обыкновенные коллекции. Мы можем создать ограниченную коллекцию с помощью команды `db.createCollection`, включив флаг `capped`:

```
//ограничиваем размер коллекции до 1 мегабайта
db.createCollection('logs', {capped: true, size: 1048576})
```

Когда наша ограниченная коллекция достигнет размера в 1 мегабайт, старые документы начнут автоматически удаляться. Можно также задать не размер коллекции, а максимальное количество документов, с помощью опции `max`. У ограниченных коллекций есть ряд интересных свойств. Например, можно изменить документ, но он не может вырасти в размере. Также сохраняется порядок вставки, так что не нужно добавлять дополнительное поле для хронологической сортировки.

Также стоит заметить, что если нужно выяснить, вызвала ли ваша запись какие-либо ошибки (как, например, в уже упомянутом случае, когда мы не дожидаемся её завершения), можно просто выполнить следующую команду: `db.getLastError()`. Большинство драйверов инкапсулируют эту функцию, как *безопасную запись*, например, можно указать `{:safe => true}` вторым параметром метода `insert`.

Устойчивость

MongoDB до версии 1.8 не обеспечивала устойчивости данных на одном сервере. Так, отказ сервера мог привести к потере данных. Решение всегда состояло в работе MongoDB на нескольких серверах (MongoDB поддерживает репликацию). Одной из самых важных функций, добавленных в MongoDB 1.8, стало журналирование. Чтобы включить его, добавьте `journal=true` в файл `mongodb.config`, созданный нами при первой настройке MongoDB (и перезапустите сервер, чтобы изменения вступили в силу). Скорее всего, журналирование вам понадобится (в следующих релизах по умолчанию оно будет включено). Несмотря на некоторое увеличение производительности, которое может быть достигнуто при отключении журналирования, возможен определенный риск. (С другой стороны, бывают приложения, которые допускают потерю некоторых данных).

Устойчивость данных упоминается здесь потому, что много сил было затрачено для того, чтобы добиться её в пределах одного сервера. Вы рано или поздно найдёте в Google упоминания о ненадёжности Mongo как хранилища. Однако эта информация уже устарела.

Полнотекстовый поиск

В будущих релизах, надеюсь, полнотекстовый поиск придёт в MongoDB. С поддержкой для массивов базовый полнотекстовый поиск будет довольно просто применять. Для мощных приложений скорее всего понадобится использовать нечто вроде Lucene или Solr. Конечно также это справедливо и для реляционных баз данных.

Транзакции

MongoDB не поддерживает транзакций. Есть две альтернативы: одна - замечательная, но ограниченная в использовании, а другая - громоздкая, но гибкая.

Первая альтернатива - это множество атомарных операций. Они прекрасны до тех пор, пока решают вашу проблему. Мы уже видели некоторые из них, например, `$inc` и `$set`. Также существуют команды вроде `findAndModify` которые могут обновлять или удалять документ и автоматически его возвращать.

Вторая альтернатива - когда атомарных операций не хватает - это двухфазный коммит. Двухфазный коммит по сравнению с транзакциями - это примерно то же самое, что ручное разруливание запросов по сравнению с JOIN-ами. Это независимое от хранилища решение, которое вы осуществляете в коде. Также двухфазный коммит достаточно распространён в реляционном мире, когда нужно обеспечить транзакции в пределах нескольких баз данных. На сайте MongoDB есть пример иллюстрирующий наиболее распространённый сценарий (перевод денежных средств). Общая идея состоит в том, что вы храните состояние транзакции внутри обновляющегося документа и проходите шаги init-pending-commit/rollback вручную.

Поддержка вложенных документов и бесструктурная архитектура MongoDB делают двухфазные коммиты не такими уж страшными, но всё равно это сложный процесс, особенно для тех, кто впервые с этим сталкивается.

Обработка данных

Для большинства задач обработки данных MongoDB использует MapReduce. Есть, конечно, некоторые базовые агрегирующие функции, но для чего-либо серьёзного вам понадобится MapReduce. В следующей главе мы рассмотрим MapReduce более детально. Сейчас можете считать его очень мощным и альтернативным вариантом `group by` (что, впрочем, будет преуменьшением его возможностей). Одно из преимуществ MapReduce в том, что для работы с большими объёмами данных он может выполняться параллельно. Однако реализация MongoDB основана на JavaScript, который сам по себе однопоточен. Что из этого следует? Для обработки больших данных вам, скорее всего, придётся полагаться на что-то другое, например, на Hadoop. К счастью, эти две системы настолько дополняют друг друга, что существует MongoDB адаптер для Hadoop.

Конечно, распараллеливание обработки данных не является однозначным предметом превосходства реляционных баз данных. В будущих релизах MongoDB планируется улучшить обработку огромных объёмов данных.

Геопространственные данные

Особенно мощной функцией MongoDB является её поддержка геопространственных индексов. Это позволяет сохранять x- и y-координаты у документов и затем находить документы вблизи (`$near`) определённых координат, или внутри (`$within`) прямоугольника либо окружности. Это легче понять визуально, поэтому я советую посмотреть пятиминутный практикум по геопространственным функциям MongoDB, если хотите углубить свои знания.

Инструментарий и зрелость

Вы уже, наверное, знаете - MongoDB значительно младше большинства реляционных баз данных. Это обязательно нужно учитывать. Насколько большую роль это играет - зависит от ваших задач и их реализации. Нельзя игнорировать тот факт, что MongoDB - молодая технология, и доступный инструментарий еще не очень разнообразен (впрочем, инструментарий зрелых реляционных баз данных бывает подчас просто ужасен). Например, отсутствие поддержки десятичных чисел с плавающей запятой, очевидно, будет проблемой (хотя и не обязательно непреодолимой) для систем, имеющих дело с деньгами.

Есть и положительные стороны: для большинства языков написаны хорошие драйверы, протокол - современный и простой, разработка движется довольно быстро. MongoDB используется на рабочих серверах у многих компаний, так что волнения о зрелости технологии скоро уйдут в историю.

В этой главе

Идея этой главы в том, что MongoDB в большинстве случаев способна стать заменой реляционной базе данных. Она намного проще и понятнее; быстрее работает и имеет меньше ограничений для разработчиков приложений. Отсутствие транзакций может вызывать серьёзную и правомочную озабоченность. Однако, когда спрашивают *какое место занимает MongoDB в экосистеме современных механизмов хранения?*, ответ прост: **строго посередине**.

Глава 6 - MapReduce

MapReduce - это подход к обработке данных, который имеет два серьёзных преимущества по сравнению с традиционными решениями. Первое и самое главное преимущество - это производительность. Теоретически MapReduce может быть распараллелен, что позволяет обрабатывать огромные массивы данных на множестве ядер/процессоров/машин. Как уже упоминалось, это пока не является преимуществом MongoDB. Вторым преимуществом MapReduce является возможность описывать обработку данных нормальным кодом. По сравнению с тем, что можно сделать с помощью SQL, возможности кода внутри MapReduce намного богаче и позволяют расширить рамки возможного даже без использования специализированных решений.

MapReduce - это стремительно приобретающий популярность шаблон, который уже можно использовать почти везде; реализации уже имеются в C#, Ruby, Java, Python. Должен предупредить, что на первый взгляд он может показаться очень непривычным и сложным. Не расстраивайтесь, не торопитесь и поэкспериментируйте с ним самостоятельно. Это стоит того - не важно, используете вы MongoDB или нет.

Теория и практика

MapReduce - процесс двухступенчатый. Сначала делается map *(отображение)*, затем - reduce *(свёртка)*. На этапе отображения входные документы трансформируются *(map)* и порождают *(emit)* пары ключ=>значение (как ключ, так и значение могут быть составными). При свёртке *(reduce)* на входе получается ключ и массив значений, порождённых для этого ключа, а на выходе получается финальный результат. Посмотрим на оба этапа и на их выходные данные.

В нашем примере мы будем генерировать отчёт по дневному количеству хитов для какого-либо ресурса (например, веб-страницы). Это *hello world* для MapReduce. Для наших задач мы воспользуемся коллекцией `hits` с двумя полями: `resource` и `date`. Желаемый результат - это отчёт в разрезе ресурса, года, месяца, дня и количества.

Пусть в `hits` лежат 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
```

На выходе мы хотим следующий 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
```

(Прелесть данного подхода заключается в хранении результатов; отчёты генерируются быстро и рост данных контролируется - для одного ресурса в день будет добавляться максимум один документ.)

Давайте теперь сосредоточимся на понимании концепции. В конце главы в качестве примера будут приведены данные и код.

Первым делом рассмотрим функцию отображения. Задача функции отображения - породить значения, которые в дальнейшем будут использоваться при свёртке. Порождать значения можно ноль или более раз. В нашем случае - как чаще всего бывает - это всегда будет делаться один раз. Представьте, что map в цикле перебирает каждый документ в коллекции `hits`. Для каждого документа мы должны породить *ключ*, состоящий из ресурса, года, месяца и дня, и примитивное *значение* - единицу:

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

`this` ссылается на текущий рассматриваемый документ. Надеюсь, результирующие данные прояснят для вас картину происходящего. При использовании наших тестовых данных, в результате получим:

```
{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 этапа даёт ключ к пониманию MapReduce. Порождённые данные собираются в массивы по одинаковому ключу. .NET и Java разработчики могут рассматривать это как тип `IDictionary<object, IList<object>>` (.NET) или `HashMap<Object, ArrayList>` (Java).

Давайте изменим нашу map-функцию несколько надуманным способом:

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

Первый промежуточный результат теперь изменится на:

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

Обратите внимание, как каждый emit порождает новое значение, которое группируется по ключу.

Reduce-функция берёт каждое из этих промежуточных значений и выдаёт конечный результат. Вот так будет выглядеть наша функция:

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

На выходе получим:

```
{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 в MongoDB результат выглядит так:

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

Это и есть наш конечный результат.

Если вы были внимательны, вы должны были спросить себя: *почему мы просто не написали `sum = values.length`?* Это было бы эффективным подходом, если бы мы суммировали массив единиц. На деле reduce не всегда вызывается с полным и совершенным набором промежуточных данных. Например вместо того, чтобы быть вызванным с:

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

Reduce может быть вызван с:

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

Конечный результат тот же самый (3), однако он получается немного разными путями. Таким образом, reduce должен всегда быть идемпотентным. То есть, вызывая reduce несколько раз, мы должны получать такой же результат, что и вызывая его один раз.

Мы не станем рассматривать этого здесь, однако распространена практика последовательных свёрток, когда требуется выполнить сложный анализ.

Чистая практика

С MongoDB мы вызываем у коллекции команду `mapReduce`. `mapReduce` принимает функцию map, функцию reduce и директивы для результата. В консоли мы можем создавать и передавать JavaScript функции. Из большинства библиотек вы будете передавать строковое представление функции (которое может выглядеть немного ужасно). Сперва давайте создадим 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 можно создать map и reduce функции (консоль MongoDB позволяет вводить многострочные конструкции):

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

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

Мы выполним команду `mapReduce` над коллекцией `hits` следующим образом:

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

Если вы выполните код, приведённый выше, вы увидите ожидаемый результат. Установив `out` в `inline` мы указываем, что `mapReduce` должен непосредственно вернуть результат в консоль. В данный момент размер результата ограничен 16 мегабайтами. Вместо этого мы могли бы написать `{out: 'hit_stats'}`, и результат был бы сохранён в коллекцию `hit_stats`:

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

В таком случае все существовавшие данные из коллекции `hit_stats` были бы вначале удалены. Если бы мы написали `{out: {merge: 'hit_stats'}}`, существующие значения по соответствующим ключам были бы заменены на новые, а другие были бы вставлены. И наконец, можно в `out` использовать `reduce` функцию - для более сложных случаев.

Третий параметр принимает дополнительные значения - например, можно сортировать, фильтровать или ограничивать анализируемые данные. Мы также можем передать метод `finalize`, который применится к результату возвращённому этапом `reduce`.

В этой главе

Это первая глава, в которой мы осветили совершенно новую для вас тему. Если вы испытываете неудобства, всегда можно обратиться к другим стредствам агрегирования

и более простым сценариям. Впрочем, MapReduce является одной из наиболее важных функций MongoDB. Чтобы научиться писать map и reduce функции, необходимо чётко представлять и понимать, как выглядят ваши данные и как они преобразовываются по пути через `map` и `reduce`.

Chapter 7 - Производительность и инструментарий

В этой главе мы коснёмся некоторых вопросов производительности, а также рассмотрим инструментарий, доступный разработчикам MongoDB. Мы не станем сильно погружаться в эти темы, но рассмотрим наиболее важные аспекты каждой.

Индексы

В самом начале мы видели коллекцию `system.indexes`, которая содержит информацию о всех индексах в нашей базе данных. Индексы в MongoDB работают схожим образом с индексами в реляционных базах данных: они ускоряют выборку и сортировку данных. Индексы создаются с помощью `ensureIndex`:

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

И уничтожаются с помощью `dropIndex`:

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

Уникальный индекс может быть создан, если во втором параметре установить `unique` в `true`:

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

Можно создавать индексы над вложенными полями (опять же, используя точечную нотацию), либо над массивами. Также можно создавать составные индексы:

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

Порядок вашего индекса (1 для восходящего и --1 для нисходящего) не играет роли в случае с простым индексом, однако он может быть существенен при сортировке или лимитировании с применением составных индексов.

На странице описания индексов можно найти дополнительную информацию.

Explain

Чтобы увидеть, используются ли индексы в ваших запросах, вызывайте у курсора метод `explain`:

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

В результате мы видим информацию, что использовался `BasicCursor` (то есть не индексированный), сканирование происходило по 12 объектам, как много это времени заняло, применялся ли индекс, и если да, то какой, а также прочие полезные сведения.

Если мы изменим запрос так, чтобы он использовал индекс, мы увидим, что использовался курсор `BtreeCursor`, а также увидим индекс, использованный при выборке:

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

Запись без подтверждения

Мы уже упоминали, что в запись данных в MongoDB происходит без подтверждения. Это может привести к приросту производительности, равно как и к риску потери данных в результате случайной ошибки. Возникает также побочный эффект, выражающийся в том, что когда обновление или вставка нарушают условие уникальности индекса, ошибки не происходит. Чтобы узнать о возникновении ошибки, после последней записи нужно вызывать `db.getLastError()`. Многие драйверы обходят это и позволяют писать *безопасно* - часто для этого имеется специальный параметр.

К сожалению, консоль не умеет этого делать, и пронаблюдать это в консоли будет непросто.

Шардинг

MongoDB поддерживает авто-шардинг. Шардинг - это подход к масштабируемости, когда отдельные части данных хранятся на разных серверах. Примитивный пример - хранить данные пользователей, чьё имя начинается на буквы А-М на одном сервере, а остальных - на другом. Возможности шардинга MongoDB значительно превосходят данный простой пример. Рассмотрение шардинга выходит за пределы данной книги, однако вы должны знать, что он существует, и вы должны воспользоваться им, когда ваши задачи выйдут за рамки одного сервера.

Репликация

Репликация в MongoDB работает сходным образом с репликацией в реляционных базах данных. Записи посылаются на один сервер - ведущий *(master)*, который потом синхронизирует своё состояние с другими серверами - ведомыми *(slave)*. Вы можете разрешить или запретить чтение с ведомых серверов, в зависимости от того, допускается ли в вашей системе чтение несогласованных данных. Если ведущий сервер падает, один из ведомых может взять на себя роль ведущего. Репликация MongoDB также выходит за пределы данной книги.

Хотя репликация увеличивает производительность чтения, делая его распределённым, основная её цель - увеличение надёжности. Типичным подходом является сочетание репликации и шардинга. Например, каждый шард может состоять из ведущего и ведомого серверов. (Технически, вам также понадобится арбитр, чтобы разрешить конфликт, когда два ведомых сервера пытаются объявить себя ведущими. Но арбитр потребляет очень мало ресурсов и может быть использован для нескольких шардов сразу.)

Статистика

Статистику базы данных можно получить с помощью вызова `db.stats()`. В основном информация касается размера вашей базы данных. Также можно получить статистику коллекции, например `unicorns`, с помощью вызова `db.unicorns.stats()`. Боьшая часть получаемой информации, опять же, касается размеров коллекции.

Веб-интерфейс

Когда `mongod` запускается, в консоли появляется, среди прочих, строчка со ссылкой на административный веб-интерфейс. Вы можете получить к нему доступ, зайдя в браузере на `http://localhost:28017/`. Чтобы получить от него максимальную отдачу, можете добавить `rest=true` в конфигурационный файл и перезапустить процесс `mongod`. Веб-интерфейс даёт много интересной информации о текущем состоянии сервера.

Профайлер

Профайлер MongoDB можно включить с помощью следующего вызова:

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

Со включённым профайлером можно запустить команду:

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

И обратиться к профайлеру:

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

В результате мы увидим, что и когда запускалось, как много документов сканировалось, как много данных было возвращено.

Можно выключить профайлер, повторно вызвав `setProfileLevel`, только передав `0` в качестве аргумента. Можно также передать `1`для профилирования запросов, выполняющихся дольше 100 миллисекунд. Также, можно вторым параметром передать время в миллисекундах:

```
//профилировать всё, что занимает более 1 секунды
db.setProfilingLevel(1, 1000);
```

Резервное копирование и восстановление

В папке `bin` MongoDB есть утилита `mongodump`. После выполнения `mongodump` произойдёт подключение к `localhost` и резервное копирование всех баз данных в подпапку `dump`.

Можно набрать `mongodump --help` и увидеть дополнительные опции. Распространённые опции: `--db DBNAME` для резервного копирования только указанной базы данных и `--collection COLLECTIONAME` для резервного копирования только указанной коллекции. После этого можно использовать `mongorestore`, расположенный в той же папке `bin`, чтобы восстановить базу данных из предварительно сделанной резервной копии. Здесь также можно указать `--db` и `--collection`, чтобы восстановить только указанные базу данных и коллекцию.

Например, чтобы сделать резервную копию базы данных `learn` в папку `backup`, мы должны выполнить (разумеется не в консоли самой MongoDB, а просто в консоли операционной системы):

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

Чтобы восстановить только коллекцию `unicorns` мы должны сделать следующее:

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

Также, стоит упомянуть, что есть две утилиты `mongoexport` и `mongoimport`, предназначенные для экспорта и импорта данных в виде JSON и CSV. Например, можно получить результат в виде JSON следующим образом:

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

И CSV:

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

Имейте в виду, что `mongoexport` и `mongoimport` не могут полностью отражать ваши данные. Только `mongodump` и `mongorestore` должны использоваться для настоящего резервного копирования.

В этой главе

В этой главе мы рассмотрели различные команды, инструменты и нюансы производительности MongoDB. Мы коснулись не всех тем, однако рассмотрели наиболее распространённые. Индексирование в MongoDB похоже на индексирование в реляционных базах данных, то же касается большинства инструментария. Однако в MongoDB пользоваться всем намного проще.

Заключение

Теперь у вас достаточно информации для того, чтобы начать пользоваться MongoDB в реальных проектах. MongoDB имеет в себе еще множество аспектов, о которых не говорилось в книге, однако вашей ближайшей задачей будет воспользоваться полученными знаниями и начать изучать драйвер, который вы будете использовать. На сайте MongoDB есть много полезной информации. В официальной группе MongoDB можно получить ответы на множество вопросов.

NoSQL создаётся не только из необходимости, но еще и из интереса к поиску новых подходов. Это значит, что мы находимся на передовом фронте, и успех может не прийти только к тем, кто опускает руки. Вот так, я думаю, и нужно жить в нашей с вами профессии.