O'REILLY

В этой книге детально рассматривается обновленная версия MongoDB 4.2 – мощной системы управления базами данных. Вы узнаете о том, как эта безопасная, высокопроизводительная система обеспечивает гибкие модели данных, высокую доступность и горизонтальную масштабируемость.
Авторы представляют руководство для разработчиков баз данных, расширенные настройки для системных администраторов и сценарии использования MongoDB в различных проектах.

Прочитав руководство, вы научитесь:

- работать с MongoDB, выполнять операции записи, находить документы и создавать сложные запросы;
- индексировать коллекции, агрегировать данные и использовать транзакции для своего приложения;
- настраивать набор локальных реплик и оценивать взаимодействие репликации с вашим приложением;
- настраивать компоненты кластера и выбирать ключ шардинга для различных приложений;
- исследовать аспекты администрирования приложений, настраивать аутентификацию и авторизацию;
- использовать статистику при мониторинге, резервном копировании и восстановлении развертываний, а также системные настройки при развертывании MongoDB.

Шеннон Брэдшоу – вице-президент по образованию в компании MongoDB. Управляет интерактивными и личными учебными продуктами, предоставляемыми через Университет MongoDB и программу профессиональной сертификации MongoDB.
Йон Брэзил – старший инженер по учебным программам в образовательной команде MongoDB. Работает над онлайн-продуктами и учебными продуктами под руководством инструктора.
Кристина Ходоров – инженер-программист. В течение пяти лет работала над ядром MongoDB. Отвечала за разработку набора реплик MongoDB, писала драйверы для PHP и Perl.

ISBN 978-5-97060-792-3

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

ДМК ИЗДАТЕЛЬСТВО
www.дмк.рф

Mongo DB
Полное руководство
O'REILLY

O'REILLY

Mongo DB

Полное руководство

Мощная и масштабируемая
система управления базами данных

Шеннон Брэдшоу
Йон Брэзил
Кристина Ходоров

ДМК ИЗДАТЕЛЬСТВО

Шеннон Брэдшоу, Йон Брэзил, Кристина Ходоров

MongoDB: полное руководство

THIRD EDITION

MongoDB: The Definitive Guide

Powerful and Scalable Data Storage

Shannon Bradshaw, Eoin Brazil,
and Kristina Chodorow

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

MongoDB: полное руководство

Мощная и масштабируемая система управления базами данных

Шеннон Брэдшоу, Йон Брэзил и Кристина Ходоров

Перевод с английского Беликова Д. А.

Москва, 2020 ДМК ПРЕСС издательство

УДК 004.65
ББК 32.972.134
Б87

Б87 Шеннон Брэдшоу, Йон Брэзил, Кристина Ходоров

MongoDB: полное руководство. Мощная и масштабируемая система управления базами данных / пер. с англ. Д. А. Беликова – М.: ДМК Пресс, 2020. – 540 с.: ил.

ISBN 978-5-97060-792-3

Эта книга представляет собой исчерпывающее руководство по работе с MongoDB 4.2 – мощной документоориентированной системой управления базами данных. Авторы, внесшие личный вклад в создание и развитие MongoDB, начинают описание системы с самых азов (история создания, базовая терминология) и постепенно переходят к более сложным темам (выполнение запросов, индексация, агрегирование, транзакции, наборы реплик, управление операциями, шардинг и администрирование данных, долговечность, мониторинг и безопасность).

Читатель получит конкретные советы по написанию приложения, которое хорошо работает с MongoDB, выяснит, какие системные настройки следует учитывать при ее развертывании и как устанавливать MongoDB в Windows, OS X и Linux.

Издание предназначено для разработчиков объемных баз данных, масштабирование которых является одной из приоритетных задач.

УДК 004.65
ББК 32.972.134

ISBN 978-1-49195-446-1 (англ.)
ISBN 978-5-97060-792-3 (рус.)

Эта книга посвящается нашим семьям
в благодарность за то время, свободу действий и поддержку,
которую они предоставили, чтобы сделать нашу работу
над этой книгой возможной, и за их любовь.

Анне, Сигурни, Грэму и Беккету – Шэннон
И Джемме, Клоде и Броне – Йон

Оглавление

Предисловие

Как устроена эта книга

Эта книга разделена на шесть частей, в которых приводятся сведения о разработке, администрировании и развертывании.

Начало работы с MongoDB

В главе 1 мы рассказываем о MongoDB: почему она была создана, какие цели пытается достичь и почему вы можете использовать ее для своего проекта. Более подробно мы рассмотрим главу 2, в которой представлены базовые понятия и словарь MongoDB. В главе 2 вы приступите к работе с базой данных и оболочкой. Следующие две главы посвящены основному материалу, который необходимо знать разработчикам для работы с MongoDB. В главе 3 мы опишем, как выполнять базовые операции записи, в том числе как делать это с различными уровнями безопасности и скорости. В главе 4 объясняется, как найти документы и создавать сложные запросы, а также рассказывается, как перебирать результаты, здесь приводятся варианты для ограничения, пропуска и сортировки результатов.

Разработка с MongoDB

В главе 5 описано, что такое индексирование и как индексировать свои коллекции MongoDB. В главе 6 объясняется, как использовать несколько специальных типов индексов и коллекций. В главе 7 рассматривается ряд методов агрегирования данных с MongoDB, включая подсчет, поиск различных значений, группировку документов, фреймворк агрегации и запись этих результатов в коллекцию. Глава 8 знакомит вас с транзакциями: что это такое, как лучше всего использовать их для своего приложения и как настроить. Наконец, эта часть заканчивается главой о разработке вашего приложения: в главе 9 содержатся советы по написанию приложения, которое хорошо работает с MongoDB.

Репликация

Часть, посвященная репликации, начинается с главы 10, в которой дается быстрый способ настроить набор реплик локально, и охватывает многие из доступных параметров конфигурирования. Затем в главе 11 рассматриваются различные концепции, связанные с репликацией. В главе 12 показано, как репликация взаимодействует с вашим приложением, а в главе 13 разбираются административные аспекты запуска набора реплик.

Шардинг

Часть, посвященная шардингу, начинается с главы 14, где дается описание быстрой локальной настройки. Затем в главе 15 приводится обзор компонентов кластера и рассказывается, как их настроить. Глава 16 содержит советы по выбору ключа шардинга для различных приложений. Наконец, глава 17 посвящена администрированию разделенного кластера.

Администрирование приложений

В следующих двух главах рассматривается множество аспектов администрирования MongoDB с точки зрения вашего приложения. В главе 18 обсуждается, как проанализировать то, что делает MongoDB. Глава 19 посвящена безопасности в MongoDb, настройке аутентификации и авторизации для вашего развертывания. В главе 20 объясняется, как MongoDB надежно хранит данные.

Администрирование сервера

Последняя часть посвящена администрированию сервера. В главе 21 описываются общие параметры при запуске и остановке MongoDB. В главе 22 обсуждается, что искать и как читать статистику во время мониторинга. В главе 23 описано, как сделать резервные копии и провести восстановление для каждого типа развертывания. Наконец, в главе 24 обсуждается ряд системных настроек, которые следует учитывать при развертывании MongoDB.

Приложения

В приложении A объясняется схема управления версиями в MongoDB и ее установка в Windows, OS X и Linux. Приложение B подробно описывает внутреннюю работу MongoDB: механизм хранения, формат данных и проводной протокол.

Обозначения, принятые в этой книге

В этой книге используются следующие типографские обозначения.

Курсив

Используется для обозначений новых терминов, URL-адресов, адресов электронной почты, имен коллекций, баз данных, файлов и расширений файлов.

`Моноширинный шрифт`

Используется в листингах программ, а также в абзацах для ссылки на элементы программы, такие как имена переменных или функ-

ций, утилиты командной строки, переменные среды, операторы и ключевые слова.

`Моноширинный полужирный шрифт`

Показывает команды или другой текст, который должен быть набран пользователем буквально.

`Моноширинный курсив`

Показывает текст, который должен быть заменен предоставленными пользователем значениями или значениями, определенными контекстом.

Этот элемент означает подсказку или предложение.

Этот элемент означает общее примечание.

Этот элемент указывает на предупреждение или предостережение.

Использование примеров кода

Дополнительный материал (примеры кода, упражнения и т. д.) можно загрузить по адресу https://github.com/mongodb-the-definitive-guide-3e/mongodb-the-definitive-guide-3e.

Если у вас есть вопрос технического характера или возникла проблема, связанная с примерами кода, отправьте письмо на адрес bookquestions@oreilly.com.

Данная книга призвана помочь вам выполнить свою работу. В общем, вы можете использовать код из этой книги в своих программах и документации. Вам не нужно обращаться к нам за разрешением, если вы не воспроизводите значительную часть кода. Например, для написания программы, в которой используется несколько фрагментов кода из этой книги, оно не требуется. Продажа или распространение CD-ROM с примерами из книг

O'Reilly требует разрешения. Чтобы ответить на вопрос, сославшись на эту книгу и приведя пример кода, разрешение не требуется. Включение значительного количества примеров кода из этой книги в документацию вашего продукта требует разрешения.

Атрибуция желательна, но не является обязательной. Обычно она включает в себя название книги, автора, издателя и ISBN. Например: «Книга рецептов R, 2-е изд., Дж. Д. Лонг и Пол Титор. Copyright 2019 Дж. Лонг и Пол Титор, 978-1-492-04068-2».

Если вы считаете, что использование примеров кода выходит за рамки добросовестного применения или только что описанного разрешения, свяжитесь с нами по адресу permissions@oreilly.com.

Обучение в режиме онлайн

На протяжении почти 40 лет O'Reilly Media (https://www.oreilly.com) предоставляет технологии и бизнес-тренинги, знания и анализ, чтобы помочь компаниям добиваться успеха.

Наша уникальная сеть экспертов и новаторов делится своими знаниями и опытом через книги, статьи, конференции и нашу онлайн-платформу обучения. Платформа онлайн-обучения O'Reilly предоставляет доступ по требованию к курсам обучения в режиме реального времени, углубленным способам обучения, интерактивным средам кодирования и обширной коллекции текстов и видео от O'Reilly и свыше 200 других издательств. Для получения дополнительной информации, пожалуйста, посетите сайт http://oreilly.com.

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте **www.dmkpress.com**, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и O'Reilly очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую предоставлять вам качественные материалы.

Часть I

Введение в MongoDB

Глава 1

Введение

MongoDB – это мощная, гибкая и масштабируемая система управления базами данных (СУБД) общего назначения. Она сочетает в себе возможность масштабирования с такими функциями, как вторичные индексы, запросы по диапазону, сортировка, агрегирование и геопространственные индексы. В этой главе рассматриваются основные проектные решения, которые сделали MongoDB тем, чем она является.

Простота использования

MongoDB – это не реляционная, а *документоориентированная* система управления базами данных. Основной причиной отказа от реляционной модели является упрощение горизонтального масштабирования, но есть и другие преимущества.

Документоориентированная СУБД заменяет концепцию «строки» более гибкой моделью, «документом». Позволяя использовать вложенные документы и массивы, документоориентированный подход дает возможность представлять сложные иерархические отношения с помощью одной записи. Это естественным образом вписывается в то, как разработчики, работающие с современными объектно-ориентированными языками, рассматривают свои данные.

Также нет предопределенных схем: ключи и значения документа не имеют фиксированных типов или размеров. Когда нет фиксированной схемы, добавлять или удалять поля по мере необходимости становится проще. Как правило, это ускоряет разработку, поскольку разработчики могут быстро выполнять итерации. Экспериментировать также проще. Разработчики могут опробовать десятки моделей для данных, а затем выбрать лучшую.

Разработана для масштабирования

Размеры наборов данных для приложений растут невероятными темпами. Увеличение доступной пропускной способности и дешевые хранилища создали среду, в которой даже небольшим приложениям необходимо

хранить больше данных, чем способны обработать многие базы данных. Терабайт данных, некогда неслыханный объем информации, теперь стал обычным явлением.

По мере роста объема данных, которые необходимо хранить разработчикам, последние сталкиваются с трудным решением: как масштабировать свои базы данных? Масштабирование базы данных сводится к выбору между вертикальным масштабированием (получение более крупной машины) и горизонтальным масштабированием (партиционирование данных на нескольких машинах). Вертикальное масштабирование часто является путем наименьшего сопротивления, но у него имеются свои недостатки: большие машины нередко очень дороги, и в конечном итоге достигается физический предел, когда более мощную машину нельзя купить любой ценой. Альтернативой является горизонтальное масштабирование: добавить место для хранения или увеличить пропускную способность для операций чтения и записи, приобрести дополнительные серверы и добавить их в свой кластер. Это и дешевле, и более масштабируемо; однако администрировать тысячу машин сложнее, чем заботиться об одной.

MongoDB была разработана для горизонтального масштабирования. Документоориентированная модель данных облегчает распределение данных между несколькими серверами. MongoDB автоматически заботится о балансировке данных и нагрузки в кластере, автоматически перераспределяя документы и направляя операции чтения и записи в нужные машины, как показано на рис. 1.1.

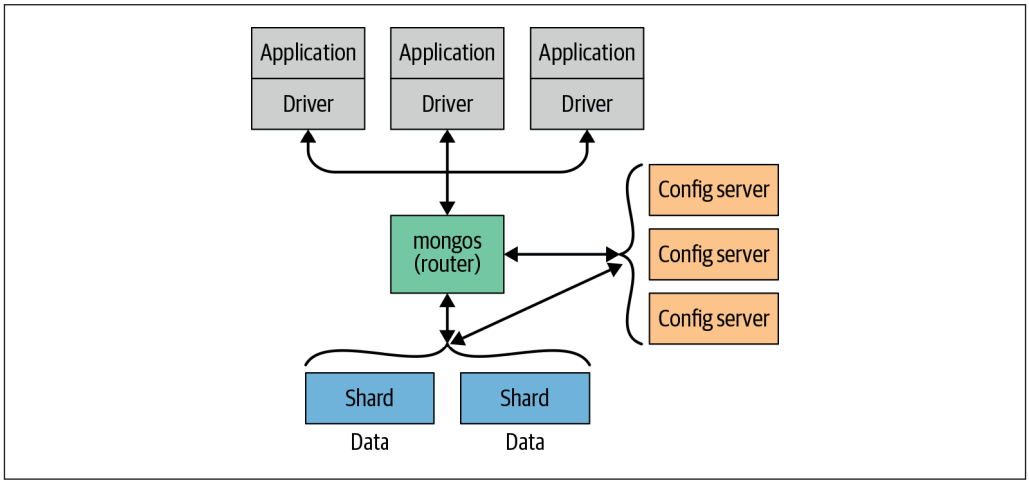

Рис. 1.1. Масштабирование MongoDB с использованием шардинга на нескольких серверах

Топология кластера MongoDB, или же это фактически кластер, а не один узел на другом конце соединения с базой данных, прозрачна для прило-

жения. Это позволяет разработчикам сосредоточиться на программировании приложения, а не на его масштабировании.

Аналогичным образом, если необходимо изменить топологию существующего развертывания, например для масштабирования, чтобы поддержать большую нагрузку, логика приложения может оставаться прежней.

Богатство функций...

MongoDB – СУБД общего назначения, поэтому помимо создания, чтения, обновления и удаления данных она предоставляет большинство тех функций, которые можно ожидать от системы управления базами данных, и многие другие, которые выделяют ее. Среди них:

Индексирование

MongoDB поддерживает общие вторичные индексы и предоставляет уникальное, составное, геопространственное и полнотекстовое индексирование. Также поддерживаются вторичные индексы в иерархических структурах, таких как вложенные документы и массивы, которые позволяют разработчикам в полной мере использовать возможность моделирования таким способом, который наиболее подходит для их приложений.

Агрегация

MongoDB предоставляет фреймворк для агрегации на базе концепции конвейеров обработки данных. Конвейеры агрегации позволяют создавать сложные аналитические механизмы, обрабатывая данные через ряд относительно простых этапов на стороне сервера, используя все преимущества оптимизации базы данных.

Специальные типы коллекций и индексов

MongoDB поддерживает коллекции данных TTL (time-to-live), срок действия которых должен истечь в определенное время, такие как сеансы и коллекции фиксированного размера, для хранения недавно полученных данных, например журналов. MongoDB также поддерживает частичные индексы, ограниченные только теми документами, которые соответствуют фильтру критериев, чтобы повысить эффективность и уменьшить необходимый объем дискового пространства.

Файловое хранилище

MongoDB поддерживает простой в использовании протокол для хранения больших файлов и метаданных файлов.

Некоторые функции, распространенные в реляционных СУБД, в MongoDB отсутствуют, особенно сложные соединения. MongoDB поддержива-

ет соединения очень ограниченным образом посредством использования оператора агрегации `$lookup`, который появился в выпуске 3.2. В версии 3.6 более сложные соединения возможны с использованием нескольких условий объединения, а также несвязанных подзапросов. Обработка соединений в MongoDB была архитектурным решением, обеспечивающим бóльшую масштабируемость, поскольку обе эти функции сложно эффективно реализовать в распределенной системе.

...Без ущерба для скорости

Производительность является основной целью MongoDB и во многом сформировала ее дизайн. Она использует уступающую блокировку в своей подсистеме хранения WiredTiger, чтобы максимизировать параллелизм и пропускную способность. Она применяет столько оперативной памяти, сколько может, как и свой кеш, и пытается автоматически выбирать правильные индексы для запросов. Говоря кратко, почти каждый аспект MongoDB был разработан для поддержания высокой производительности.

Хотя MongoDB является мощным средством, включающим в себя множество функций реляционных систем, она не предназначена для выполнения всего того, что делает реляционная СУБД. В случае с некоторыми функциями сервер базы данных переносит обработку и логику на клиентскую сторону (обработка осуществляется либо драйверами, либо кодом приложения пользователя). Поддержание этого обтекаемого дизайна является одной из причин, по которой MongoDB может достигать такой высокой производительности.

Философия

В этой книге мы уделим время тому, чтобы отметить причины или мотивы, стоящие за конкретными решениями, которые принимались при разработке MongoDB. С помощью этого мы надеемся поделиться философией MongoDB. Однако лучший способ подвести итоги проекта MongoDB – обозначить его главную цель – создание полноценного хранилища данных, которое является масштабируемым, гибким и быстрым.

Глава 2

Начало работы

MongoDB – мощный инструмент, с которым легко начать работу. В этой главе мы познакомимся с некоторыми основными понятиями MongoDB:

- *документ* представляет собой основную единицу данных в MongoDB и приблизительно эквивалентен строке в реляционной системе управления базами данных (но гораздо более выразителен);
- аналогично *коллекцию* можно рассматривать как таблицу с динамической схемой;
- один экземпляр MongoDB может содержать несколько независимых *баз данных*, каждая из которых содержит свои собственные коллекции;
- у каждого документа есть специальный ключ `"_id"`, который является уникальным в рамках коллекции;
- MongoDB распространяется с помощью простого, но мощного инструмента под названием оболочка *mongo*. Оболочка *mongo shell* предоставляет встроенную поддержку для администрирования экземпляров MongoDB и манипулирования данными с использованием языка запросов MongoDB. Это также полнофункциональный интерпретатор JavaScript, который позволяет пользователям создавать и загружать собственные сценарии для различных целей.

Документы

В основе MongoDB лежит *документ*: упорядоченный набор ключей со связанными значениями. Представление документа зависит от языка программирования, но большинство языков имеют естественную структуру данных, например ассоциативный массив, хеш или словарь. Например, в JavaScript документы представлены в виде объектов:

```
{"greeting" : "Hello, world!"}
```

Этот простой документ содержит единственный ключ `"greeting"` со значением `"Hello, world!"`. Большинство документов будут более сложными

по сравнению с этим и часто будут содержать несколько пар типа «ключ/значение»:

```
{"greeting" : "Hello, world!", "views" : 3}
```

Как видно, значения в документах – это не просто «BLOB-объекты». Они могут относиться к одному из нескольких типов данных (или даже ко всему вложенному документу – *см.* раздел «Вложенные документы»). В этом примере значение `"greeting"` является строкой, тогда как значение `"views"` – это целое число.

Ключи в документе – это строки. В ключе допустимо использование любого символа в кодировке UTF-8, за несколькими заметными исключениями:

- ключи не должны содержать символ \ *0* (нулевой символ). Он используется для обозначения конца ключа;
- символы . и $ обладают некоторыми специальными свойствами и должны использоваться только при определенных обстоятельствах, о чем будет рассказано в последующих главах. В целом они должны считаться зарезервированными, и драйверы будут жаловаться, если эти символы будут использоваться не по назначению.

MongoDB чувствительна к типу и регистру. Например, эти документы отличаются:

```
{"count" : 5}
{"count" : "5"}
```

равно как и эти:

```
{"count" : 5}
{"Count" : 5}
```

И еще одна важная вещь: документы в MongoDB не могут содержать дубликаты ключей. Например, приведенный ниже документ не является допустимым:

```
{"greeting" : "Hello, world!", "greeting" : "Hello, MongoDB!"}
```

Коллекции

Коллекция представляет собой группу документов. Если в MongoDB документ является аналогом строки в реляционной СУБД, то коллекцию можно рассматривать как аналог таблицы.

Динамические схемы

Коллекции имеют *динамические схемы*. Это означает, что документы в одной коллекции могут иметь любое число различных «фигур». Например, оба приведенных ниже документа могут храниться в одной коллекции:

```
{"greeting" : "Hello, world!", "views": 3}
{"signoff": "Good night, and good luck"}
```

Обратите внимание на то, что предыдущие документы имеют разные ключи, разное количество ключей и значения разных типов. Поскольку любой документ можно поместить в любую коллекцию, часто возникает вопрос: «Зачем вообще нужны отдельные коллекции?» Если нет необходимости в отдельных схемах для разных видов документов, зачем использовать дополнительные коллекции? На то есть ряд веских причин:

- хранение разных видов документов в одной коллекции может стать кошмаром для разработчиков и администраторов. Разработчики должны убедиться, что каждый запрос возвращает только документы, привязанные к определенной схеме, или что код приложения, выполняющий запрос, может обрабатывать документы различной формы. Если мы запрашиваем посты в блоге, очень сложно отсеять документы, содержащие данные об авторах;
- получить список коллекций намного быстрее, чем извлечь список типов документов в коллекции. Например, если бы в каждом документе было поле `"type"`, в котором указывалось, был ли это документ «skim», «whole» или «chunky monkey», было бы намного медленнее искать эти три значения в одной коллекции, чем иметь три отдельные коллекции и запросить правильную;
- группировка документов одного и того же вида в одной коллекции допускает локальность данных. Получение нескольких постов в блоге из коллекции, содержащей только посты, вероятно, потребует меньше операций поиска на диске, нежели получение тех же постов из коллекции, где содержатся посты и данные об авторах;
- мы начинаем навязывать своим документам некую структуру при создании индексов. (Это особенно верно в случае с уникальными индексами.) Эти индексы определяются для каждой коллекции. Помещая в одну коллекцию только документы одного типа, можно более эффективно индексировать свои коллекции.

Существуют веские причины для создания схемы и группировки связанных типов документов. Хотя этого и не требуется по умолчанию, определение схем для вашего приложения является хорошей практикой и может быть реализовано с помощью функций проверки документации MongoDB

и библиотек объектно-документного отображения, доступных для множества языков программирования.

Именование

Коллекция идентифицируется по имени. Имена коллекций могут быть любой строкой в кодировке UTF-8 с некоторыми ограничениями:

- пустая строка (“”) не является допустимым именем коллекции;
- имена коллекций не могут содержать символ \ *0* (нулевой символ), поскольку он обозначает конец имени коллекции;
- не следует создавать коллекции с именами, начинающимися со слова *system*. Этот префикс зарезервирован для внутренних коллекций. Например, коллекция *system.users* содержит пользователей базы данных, а коллекция *system.namespaces* содержит информацию обо всех коллекциях базы данных;
- созданные пользователем коллекции не должны содержать зарезервированный символ *$* в своих именах. Различные драйверы, доступные для базы данных, все же поддерживают применение этого символа в именах коллекций, потому что он содержится в некоторых сгенерированных системой коллекциях, но вы не должны использовать *$* в имени, только если вы не выполняете доступ к одной из этих коллекций.

Вложенные коллекции

Одно из соглашений для организации коллекций состоит в том, чтобы использовать вложенные коллекции пространства имен, разделенные символом «.». Например, приложение, содержащее блог, может иметь коллекцию с именем *blog.posts* и отдельную коллекцию с именем *blog.authors*. Это служит только организационным целям – нет никакой связи между коллекцией *blog* (она даже не должна существовать) и ее «потомками».

Хотя вложенные коллекции не имеют каких-либо специальных свойств, они полезны и включены во многие инструменты MongoDB. Например:

- GridFS, протокол для хранения больших файлов, использует вложенные коллекции для хранения метаданных файлов отдельно от блоков содержимого (дополнительную информацию о GridFS *см.* в главе 6);
- большинство драйверов предоставляют некий «синтаксический сахар» для доступа к вложенной коллекции. Например, в оболочке базы данных `db.blog` предоставит вам коллекцию *blog*, а `db.blog.posts` – коллекцию *blog.posts*.

Вложенные коллекции – хороший способ организовать данные в MongoDB для множества случаев использования.

Базы данных

В дополнение к группированию документов по коллекции MongoDB группирует коллекции в *базы данных*. Один экземпляр MongoDB может содержать несколько баз данных, каждая из которых объединяет ноль или более коллекций. Есть хорошее практическое правило – хранить все данные одного приложения в одной и той же базе данных. Отдельные базы данных полезны при хранении данных для нескольких приложений или пользователей на одном сервере MongoDB.

Как и коллекции, базы данных идентифицируются по имени. Имена баз данных могут быть любой строкой в формате UTF-8 со следующими ограничениями:

- пустая строка (“”) не является допустимым именем базы данных;
- имя базы данных не может содержать следующие символы: /, \, ., ”, *, <, >, :, |, ?, $, (один пробел) или \ *0* (нулевой символ). В основном придерживаться буквенно-цифровой таблицы ASCII;
- имена баз данных нечувствительны к регистру;
- имена баз данных ограничены максимум 64 байтами.

Традиционно, до использования подсистемы хранения WiredTiger, имена баз данных становились файлами в вашей файловой системе. Теперь этого больше нет. Это объясняет, почему многие из предыдущих ограничений вообще существуют.

Есть также некоторые зарезервированные имена баз данных, к которым можно получить доступ, но которые имеют особую семантику. Вот они:

admin

: База данных *admin* играет роль в аутентификации и авторизации. Кроме того, доступ к этой базе данных необходим для ряда административных операций. *См.* главу 19 для получения дополнительной информации о базе данных *admin*.

local

: В этой базе данных хранятся данные, относящиеся к одному серверу. В наборах реплик в базе *local* хранятся данные, используемые в процессе репликации. Сама база данных *local* никогда не реплицируется. (*См.* главу 10 для получения дополнительной информации о репликации и локальной базе данных.)

config

Разделенные (сегментированные) кластеры MongoDB (*см.* главу 14) используют базу данных *config* для хранения информации о каждом шарде.

Объединяя имя базы данных с коллекцией в этой базе данных, вы можете получить полное имя коллекции, которое называется *пространством имен*. Например, если вы используете коллекцию *blog.posts* в базе данных *cms*, пространство имен этой коллекции будет таким: *cms.blog.posts*. Длина пространств имен ограничена 120 байтами, а на практике должна быть менее 100 байт. Подробнее о пространствах имен и внутреннем представлении коллекций в MongoDB *см.* приложение B.

Начало работы с MongoDB

Чтобы запустить сервер, выполните исполняемый файл *mongod* в выбранной вами среде с интерфейсом командной строки Unix:

```
$ mongod
2016-04-27T22:15:55.871-0400 I CONTROL [initandlisten] MongoDB starting :
pid=8680 port=27017 dbpath=/data/db 64-bit host=morty
2016-04-27T22:15:55.872-0400 I CONTROL [initandlisten] db version v4.2.0
2016-04-27T22:15:55.872-0400 I CONTROL [initandlisten] git version:
34e65e5383f7ea1726332cb175b73077ec4a1b02
2016-04-27T22:15:55.872-0400 I CONTROL [initandlisten] allocator: system
2016-04-27T22:15:55.872-0400 I CONTROL [initandlisten] modules: none
2016-04-27T22:15:55.872-0400 I CONTROL [initandlisten] build environment:
2016-04-27T22:15:55.872-0400 I CONTROL [initandlisten] distarch: x86_64
2016-04-27T22:15:55.872-0400 I CONTROL [initandlisten] target_arch: x86_64
2016-04-27T22:15:55.872-0400 I CONTROL [initandlisten] options: {}
2016-04-27T22:15:55.889-0400 I JOURNAL [initandlisten]
journal dir=/data/db/journal
2016-04-27T22:15:55.889-0400 I JOURNAL [initandlisten] recover :
no journal files
present, no recovery needed
2016-04-27T22:15:55.909-0400 I JOURNAL [durability] Durability thread started
2016-04-27T22:15:55.909-0400 I JOURNAL [journal writer] Journal writer thread
started
2016-04-27T22:15:55.909-0400 I CONTROL [initandlisten]
2016-04-27T22:15:56.777-0400 I NETWORK [HostnameCanonicalizationWorker]
Starting hostname canonicalization worker
2016-04-27T22:15:56.778-0400 I FTDC [initandlisten] Initializing full-time
diagnostic data capture with directory '/data/db/diagnostic.data'
2016-04-27T22:15:56.779-0400 I NETWORK [initandlisten] waiting for connections
on port 27017
```

Если вы работаете в Windows, запустите это:

```
> mongod.exe
```

Для получения подробной информации об установке MongoDB в вашей системе см. приложение A или соответствующее руководство по установке (https://oreil.ly/5WP5e) в документации MongoDB.

При запуске без аргументов файл *mongod* будет использовать каталог данных по умолчанию, */data/db/* (или *\data\db* на текущем томе в Windows). Если каталог данных отсутствует или недоступен для записи, сервер не запустится. Важно создать каталог данных (например, `mkdir -p /data/db/`) и убедиться, что у вашего пользователя есть права на запись в каталог перед запуском MongoDB.

При запуске сервер выведет информацию о версии и системе, а затем начнет ждать подключения. По умолчанию MongoDB прослушивает подключения к сокету на порту 27017. Сервер не сможет запуститься, если этот порт недоступен, – наиболее частой причиной этого является еще один экземпляр MongoDB, который уже запущен.

Всегда следует обезопасить свои экземпляры `mongod`. См. главу 19 для получения дополнительной информации.

Можно безопасно остановить *mongod*, набрав сочетание клавиш **Ctrl-C** в окружении с интерфейсом командной строки, из которой вы запускали сервер.

Для получения дополнительной информации о запуске или остановке MongoDB см. главу 21.

Знакомство с оболочкой MongoDB

MongoDB поставляется с оболочкой JavaScript, которая позволяет взаимодействовать с экземпляром MongoDB из командной строки. Эта оболочка

полезна для выполнения административных функций, проверки работающего экземпляра или просто изучения MongoDB. Оболочка *mongo* является важным инструментом для использования MongoDB. Мы будем широко использовать ее на протяжении всей книги.

Запуск оболочки

Чтобы запустить оболочку, запустите исполняемый файл *mongo*:

```
$ mongo
MongoDB shell version: 4.2.0
connecting to: test
>
```

Оболочка автоматически пытается подключиться к серверу MongoDB, работающему на локальной машине, при запуске, поэтому убедитесь, что вы запускаете *mongod* перед запуском оболочки.

Оболочка представляет собой полнофункциональный интерпретатор JavaScript, способный запускать произвольные программы на языке JavaScript. В качестве иллюстрации давайте выполним базовые математические вычисления:

```
> x = 200;
200
> x / 5;
40
```

Мы также можем использовать все стандартные библиотеки JavaScript:

```
> Math.sin(Math.PI / 2);
1
> new Date("20109/1/1");
ISODate("2019-01-01T05:00:00Z")
> "Hello, World!".replace("World", "MongoDB");
Hello, MongoDB!
```

И даже можем определять и вызывать функции JavaScript:

```
> function factorial (n) {
... if (n <= 1) return 1;
... return n * factorial(n - 1);
... }
> factorial(5);
120
```

Обратите внимание, что вы можете создавать многострочные команды. Оболочка будет определять, завершен ли оператор JavaScript, когда

вы нажимаете **Enter**. Если оператор не завершен, оболочка позволит вам продолжить запись в следующей строке. Нажав **Enter** три раза подряд, вы отмените наполовину сформированную команду и снова увидите приглашение к вводу >.

Клиент MongoDB

Хотя возможность выполнения произвольного кода JavaScript полезна, реальная сила оболочки заключается в том, что она также является автономным клиентом MongoDB. При запуске оболочка подключается к базе данных *test* на сервере MongoDB и присваивает это подключение глобальной переменной `db`. Эта переменная является основной точкой доступа к вашему серверу MongoDB через оболочку.

Чтобы увидеть базу данных, которой присвоена переменная `db`, наберите `db` и нажмите **Enter**:

```
> db
test
```

Оболочка содержит некоторые дополнения, которые не являются допустимым синтаксисом в JavaScript, но они были реализованы по причине того, что они знакомы пользователям оболочек SQL. Эти дополнения не предоставляют никакой дополнительной функциональности, но являются хорошим синтаксическим сахаром. Например, одна из наиболее важных операций – выбор базы данных, которая будет использоваться:

```
> use video
switched to db video
```

Теперь если вы посмотрите на переменную `db`, то увидите, что она относится к базе данных *video*:

```
> db
Video
```

Поскольку это оболочка JavaScript, при вводе имени переменной имя будет оцениваться как выражение. После будет выведено значение (в данном случае имя базы данных).

Вы можете получить доступ к коллекциям из переменной `db`. Например:

```
> db.movies
```

возвращает коллекцию *movies* из текущей базы данных. Теперь, когда мы можем получить доступ к коллекции в оболочке, мы можем выполнить практически любую операцию с базой данных.

Основные операции с оболочкой

Мы можем использовать четыре основные операции: создание, чтение, обновление и удаление (CRUD), чтобы манипулировать данными и просматривать их в оболочке.

Создание

Функция `insertOne` добавляет документ в коллекцию. Например, предположим, что мы хотим сохранить фильм. Сперва мы создадим локальную переменную с именем `movie`, которая представляет собой объект JavaScript, обозначающий наш документ. У нее будут ключи `"title"`, `"director»` и `"year"` (год выпуска фильма):

```
> movie = {"title" : "Star Wars: Episode IV - A New Hope",
... "director" : "George Lucas",
... "year" : 1977}
{
"title" : "Star Wars: Episode IV - A New Hope",
"director" : "George Lucas",
"year" : 1977
}
```

Этот объект является валидным документом MongoDB, поэтому мы можем сохранить его в коллекции *movies,* используя метод `insertOne`:

```
> db.movies.insertOne(movie)
{
"acknowledged" : true,
"insertedId" : ObjectId("5721794b349c32b32a012b11")
}
```

Фильм был сохранен в базе данных. Его можно увидеть, вызвав функцию `find`:

```
> db.movies.find().pretty()
{
"_id" : ObjectId("5721794b349c32b32a012b11"),
"title" : "Star Wars: Episode IV - A New Hope",
"director" : "George Lucas",
"year" : 1977
}
```

Видно, что был добавлен ключ `"_id"` и что другие пары типа «ключ/значение» были сохранены, когда мы их вводили. Причина внезапного появления поля `"_id"` объяснена в конце этой главы.

Чтение

Функции `find` и `findOne` можно использовать для того, чтобы выполнять запросы к коллекции. Если мы просто хотим увидеть один документ из коллекции, можно использовать функцию `findOne`:

```
> db.movies.findOne()
{
"_id" : ObjectId("5721794b349c32b32a012b11"),
"title" : "Star Wars: Episode IV - A New Hope",
"director" : "George Lucas",
"year" : 1977
}
```

Этим функциям также можно передавать критерии в виде документа запроса. Это ограничит документы, соответствующие запросу. Оболочка автоматически отобразит до 20 документов, соответствующих находке, но можно получить и другие (*см.* главу 4 для получения дополнительной информации о запросах).

Обновление

Если мы хотим изменить свой пост, можно использовать функцию `updateOne`. Эта функция принимает (как минимум) два параметра: первый – это критерий для поиска документа, который нужно обновить, а второй – документ, описывающий обновления, которые необходимо сделать. Предположим, мы решили активировать отзывы о фильме, которые создали ранее. Нам нужно добавить массив отзывов в качестве значения нового ключа в наш документ.

Для выполнения обновления нам понадобится оператор обновления, `set`:

```
> db.movies.updateOne({title : "Star Wars: Episode IV - A New Hope"},
... {$set : {reviews: []}})
WriteResult({"nMatched": 1, "nUpserted": 0, "nModified": 1})
```

Теперь у документа есть ключ `"reviews"`. Если мы снова вызовем функцию `find`, то увидим новый ключ:

```
> db.movies.find().pretty()
{
"_id" : ObjectId("5721794b349c32b32a012b11"),
"title" : "Star Wars: Episode IV - A New Hope",
"director" : "George Lucas",
"year" : 1977,
"reviews" : [ ]
}
```

См. раздел «Обновление документов» для получения подробной информации.

Удаление

Функции `deleteOne` и `deleteMany` навсегда удаляют документы из базы данных. Оба метода берут документ фильтра с указанием критериев удаления. Например, с помощью приведенного ниже кода мы удалим фильм, который только что создали:

```
> db.movies.deleteOne({title : "Star Wars: Episode IV - A New Hope"})
```

Используйте метод `deleteMany`, чтобы удалить все документы, соответствующие фильтру.

Типы данных

В начале этой главы были рассмотрены основы того, что такое документ. Теперь, когда вы работаете с MongoDB и можете попробовать сделать что-то в командной оболочке, в этом разделе мы пойдем немного дальше. MongoDB поддерживает широкий спектр типов данных в качестве значений в документах. В этом разделе мы опишем все поддерживаемые типы.

Основные типы данных

Документы в MongoDB можно рассматривать как «JSON-подобные» в том смысле, что они концептуально похожи на объекты в JavaScript. JSON (http://www.json.org) является простым представлением данных: спецификация может быть описана в одном параграфе (веб-сайт, ссылка на который приведена выше, доказывает это) и содержит только шесть типов данных. Это хорошо во многих отношениях: его легко понять, конвертировать и запоминать. С другой стороны, выразительные возможности формата JSON ограничены, потому что единственными типами являются *null*, *логический тип данных*, *число*, *строка*, *массив* и *объект*.

Хотя эти типы обеспечивают впечатляющую степень выразительности, есть пара дополнительных типов, которые имеют решающее значение для большинства приложений, особенно при работе с базой данных. Например, в JSON нет типа даты, что делает работу с датами еще более раздражающей, чем обычно. Существует тип числа, но он только один – нет никакого способа различить числа с плавающей точкой и целые числа, не говоря уже о различии между 32-битными и 64-битными числами. Невозможно обозначить и другие часто используемые типы, такие как регулярные выражения или функции.

MongoDB добавляет поддержку ряда дополнительных типов данных, сохраняя при этом существенную природу пары типа «ключ/значение» в

JSON. Как именно представлены значения каждого типа, зависит от языка, но это список часто поддерживаемых типов и то, как они представлены как часть документа в оболочке. Наиболее распространенные типы:

Null

Нулевой тип можно использовать для обозначения как нулевого значения, так и несуществующего поля:

```
{"x" : null}
```

Логический тип

Существует логический тип данных, который можно использовать для значений `true` и `false`:

```
{"x" : true}
```

Число

По умолчанию оболочка использует 64-битные числа с плавающей точкой. Таким образом, в оболочке эти числа выглядят «нормально»:

```
{"x" : 3.14}
{"x" : 3}
```

В случае с целыми числами используйте классы `NumberInt` или `NumberLong`, которые обозначают 4-байтовые или 8-байтовые целые числа со знаком соответственно.

```
{"x" : NumberInt ("3")}
{"x" : NumberLong ("3")}
```

Строка

Любая строка символов в кодировке UTF-8 может быть представлена с использованием типа строки:

```
{"x" : "foobar"}
```

Дата

MongoDB хранит даты в виде 64-битных целых чисел, обозначающих миллисекунды с момента эпохи Unix (1 января 1970 г.). Часовой пояс не сохраняется:

```
{"x" : new Date()}
```

Регулярное выражение

Запросы могут использовать регулярные выражения, используя синтаксис регулярных выражений JavaScript:

```
{"x" : / foobar / i}
```

Массив

Наборы или списки значений могут быть представлены в виде массивов:

```
{"x" : ["a", "b", "c"]}
```

Встраиваемый документ

Документы могут содержать целые документы, встроенные в качестве значений в родительский документ:

```
{"x" : {"foo": "bar"}}
```

Идентификатор объекта

Идентификатор объекта – это 12-байтовый идентификатор для документов:

```
{"x" : ObjectId ()}
```

Подробности *см.* в разделе «_id и ObjectIds».

Есть также несколько менее распространенных типов, которые могут вам понадобиться, в том числе:

Двоичные данные

Двоичные данные – это строка из произвольных байтов. Ими нельзя манипулировать из оболочки. Двоичные данные – единственный способ сохранять строки не в формате UTF-8 в базе данных.

Код

MongoDB также позволяет хранить произвольный код JavaScript в запросах и документах:

```
{"x" : function() { /* ... */ }}
```

Наконец, существует несколько типов, которые в основном используются внутри (или заменяются другими типами). Их описание будет даваться в тексте по мере необходимости.

Для получения дополнительной информации о формате данных MongoDB *см.* приложение B.

Даты

В JavaScript класс `Date` используется для типа даты MongoDB. При создании нового объекта `Date` всегда вызывайте метод `new Date()`, а не просто `Date()`. Вызов конструктора в качестве функции (т. е. не используя слово `new`) возвращает строковое представление даты, а не фактический объект `Date`. Это не выбор MongoDB; так работает JavaScript. Если вы не будете

осторожны при использовании конструктора `Date`, это может привести к мешанине из строк и дат. Строки не совпадают с датами, и наоборот, поэтому это может вызвать проблемы с удалением, обновлением, запросом… практически со всем.

Для получения полного объяснения класса JavaScript `Date` и приемлемых форматов конструктора *см.* раздел 15.9 спецификации ECMAScript (http://www.ecmainternational.org).

Даты в оболочке отображаются с использованием настроек местного часового пояса. Однако даты в базе данных просто хранятся в миллисекундах с момента начала эпохи, поэтому у них нет никакой информации о часовом поясе, связанной с ними. (Информация о часовом поясе, конечно, может храниться в качестве значения другого ключа.)

Массивы

Массивы – это значения, которые могут использоваться взаимозаменяемо как для упорядоченных (как если бы они были списками, стеками или очередями), так и для неупорядоченных операций (как если бы они были наборами).

В приведенном ниже документе ключ `"things"` имеет значение массива:

```
{"things" : ["pie", 3.14]}
```

Как видно из этого примера, массивы могут содержать различные типы данных в качестве значений (в данном случае это строка и число с плавающей запятой). Фактически значения массива могут быть любыми поддерживаемыми типами значений для обычных пар типа «ключ/значение», даже для вложенных массивов.

Одна из замечательных особенностей массивов в документах заключается в том, что MongoDB «понимает» их структуру и знает, как добраться до внутренностей массивов для выполнения операций с их содержимым. Это позволяет нам делать запросы к массивам и создавать индексы, используя их содержимое. Например, в предыдущем примере MongoDB может запрашивать все документы, где `3.14` является элементом массива `"things"`. Если это обычный запрос, можно даже создать индекс для ключа `"things"`, чтобы повысить скорость запроса.

MongoDB также допускает атомарные обновления, которые изменяют содержимое массивов, например доступ к массиву и изменение значения `"pie"` на `pi`. Мы будем встречать и другие примеры этих типов операций на протяжении всей книги.

Вложенные документы

Документ может использоваться как значение ключа. Такой документ называется *вложенным*. Вложенные документы могут использоваться для

организации данных более естественным образом, чем просто плоская структура пар типа «ключ/значение».

Например, если у нас есть документ, обозначающий человека, и мы хотим сохранить адрес этого человека, можно вложить эту информацию во вложенный документ `"address"`:

```
{
  "name" : "John Doe",
  "address" : {
     "street" : "123 Park Street",
     "city" : "Anytown",
     "state" : "NY"
     }
}
```

Значением ключа `"address"` в этом примере является вложенный документ со своими собственными парами типа «ключ/значение» для `"street"`, `"city"` и `"state"`.

Как и в случае с массивами, MongoDB «понимает» структуру вложенных документов и может использовать их для создания индексов, выполнения запросов или обновлений.

Мы обсудим дизайн схемы более подробно позже, но, даже основываясь на этом базовом примере, можно увидеть, что вложенные документы могут изменить способ работы с данными. В реляционной СУБД предыдущий документ, вероятно, будет смоделирован как две отдельные строки в двух разных таблицах (*people* и *addresses*). В MongoDB мы можем встраивать документ `"address"` непосредственно в документ `"person"`. Таким образом, при правильном использовании вложенные документы могут обеспечить более естественное представление информации.

Обратная сторона состоит в том, что в случае с MongoDB повторений данных может быть больше. Предположим, что *addresses* были отдельной таблицей в реляционной базе данных и нам нужно было исправить опечатку в адресе. Когда мы выполняли соединение с таблицами *people* и *addresses*, то получали обновленный адрес для всех, кто использует его. При работе с MongoDB нам нужно исправлять опечатку в документе каждого человека.

_id и ObjectId

Каждый документ, хранящийся в MongoDB, должен иметь ключ `"_id"`. Значение ключа `"_id"` может быть любого типа, но по умолчанию используется `ObjectId`. В одной коллекции каждый документ должен иметь уникальное значение `"_id"`, что гарантирует уникальную идентификацию каждого документа в коллекции. То есть если бы у вас было две коллекции,

каждая из них могла бы иметь документ, в котором значение `"_id"` было бы равно 123. Однако ни одна коллекция не может содержать более одного документа с `"_id"`, равным 123.

ObjectId

`ObjectId` является типом по умолчанию для `"_id"`. Класс `ObjectId` разработан так, чтобы быть легковесным, но при этом его легко можно было генерировать глобально уникальным способом на разных машинах. Распределенная природа MongoDB является основной причиной, по которой она использует `ObjectId`, а не что-то более традиционное, например автоинкрементный первичный ключ: синхронизировать автоинкрементные первичные ключи на нескольких серверах сложно и отнимает много времени. Поскольку MongoDB была спроектирована как распределенная СУБД, было важно иметь возможность генерировать уникальные идентификаторы в разделенной среде.

`ObjectId` использует 12 байт памяти, что дает ему строковое представление, состоящее из 24 шестнадцатеричных цифр: по две цифры на каждый байт. Из-за этого он кажется больше, чем есть на самом деле, что заставляет некоторых нервничать. Важно отметить, что хотя `ObjectId` часто представляется в виде гигантской шестнадцатеричной строки, эта строка на самом деле в два раза длиннее сохраняемых данных.

Если вы создадите несколько новых `ObjectId` в быстрой последовательности, то увидите, что только последние несколько цифр меняются каждый раз. Кроме того, пара цифр в середине `ObjectId` изменится, если вы разместите творения на пару секунд. Это объясняется манерой создания `ObjectId`. 12 байт `ObjectId` генерируются следующим образом:

```
0                1                   2                   3 4 5 6 7 8 9 10 11
Временная метка  Случайное значение  Счетчик (случайное
                                     начальное значение)
```

Первые четыре байта `ObjectId` – это временная отметка в секундах с момента начала эпохи. Это обеспечивает пару полезных свойств:

- временная метка в сочетании с последующими пятью байтами (которые будут описаны ниже) обеспечивает уникальность с точностью до секунды;
- поскольку временная метка идет первой, `ObjectId` будут сортироваться в порядке *грубой* вставки. Это не является надежной гарантией, но имеет некоторые приятные свойства, например делает `ObjectId` эффективными для индексации;
- в этих четырех байтах существует неявная метка времени, когда был создан каждый документ. Большинство драйверов предоставляют метод для извлечения этой информации из `ObjectId`.

Поскольку в `ObjectId` используется текущее время, некоторые пользователи беспокоятся о том, что на их серверах нужно будет синхронизировать часы. Хотя синхронизированные часы являются неплохой идеей по другим причинам (*см.* раздел «Синхронизация часов»), фактическая временная метка не имеет значения для `ObjectId`, разве только то, что она часто является новой (раз в секунду) и увеличивается.

Следующие пять байт `ObjectId` – это случайное значение. Последние три байта представляют собой счетчик, который начинается со случайного значения, чтобы избежать создания конфликтующих `ObjectId` на разных машинах.

Таким образом, эти первые девять байт `ObjectId` гарантируют его уникальность на разных машинах и процессов в течение одной секунды. Последние три байта – просто инкрементный счетчик, который отвечает за уникальность в течение секунды в одном процессе.

Это позволяет генерировать до 256^3 (16 777 216) уникальных `ObjectId` *на процесс* в одну секунду.

Автогенерация _id

Как было сказано ранее, если при вставке документа ключ `"_id"` отсутствует, он будет добавлен автоматически во вставленный документ. Это может быть обработано сервером MongoDB, но обычно делается драйвером на стороне клиента.

Использование оболочки MongoDB

В этом разделе описывается, как использовать оболочку как часть вашего инструментария командной строки, настроить ее и использовать некоторые из ее более продвинутых функций.

Хотя мы и подключились к локальному экземпляру *mongod*, вы можете подключить свою оболочку к любому экземпляру MongoDB, к которому может подключиться ваша машина. Чтобы подключиться к *mongod* на другом компьютере или порту, укажите имя хоста, порт и базу данных при запуске оболочки:

```
$ mongo some-host:30000/myDB
MongoDB shell version: 4.2.0
connecting to: some-host:30000/myDB
>
```

`db` теперь будет ссылаться на базу данных `myDB` на *some-host*:*30000*.

Иногда удобно вообще не подключаться к *mongod* при запуске оболочки *mongo*. Если вы запустите оболочку с параметром `--nodb`, она запустится без попытки подключения к чему-либо:

```
$ mongo --nodb
MongoDB shell version: 4.2.0
>
```

После запуска вы можете подключиться к *mongod*, выполнив команду `new Mongo("имяхоста")`:

```
> conn = new Mongo("some-host:30000")
connection to some-host:30000
> db = conn.getDB("myDB")
myDB
```

После этих двух команд можно использовать `db` как обычно. Вы можете применять эти команды для подключения к другой базе данных или серверу в любое время.

Советы по использованию оболочки

Поскольку *mongo* – это просто оболочка JavaScript, вы можете получить по ней большое количество справочной информации, просто просмотрев документацию по JavaScript в интернете. Если вы хотите узнать об особой функциональности MongoDB, на этот случай оболочка включает в себя встроенную справку, к которой можно получить доступ, введя слово **`help`**:

```
> help
    db.help()               help on db methods
    db.mycoll.help()        help on collection methods
    sh.help()               sharding helpers
    ...

    show dbs                show database names
    show collections        show collections in current database
    show users              show users in current database
    ...
```

Справка на уровне базы данных предоставляется с помощью метода `db.help()`, а справка на уровне коллекции – с помощью метода `db.foo.help()`.

Хороший способ выяснить, что делает функция, – это ввести ее без скобок. В результате чего будет выведен исходный код JavaScript функции. Например, если вам интересно, как работает функция `update`, или вы не помните порядок параметров, можно сделать следующее:

```
> db.movies.updateOne
function (filter, update, options) {
    var opts = Object.extend({}, options || {});

    // Проверяем, содержит ли первый ключ символ $;
```

```
    var keys = Object.keys(update);
    if (keys.length == 0) {
        throw new Error("the update operation document must contain at
        least one atomic operator");
    }
    ...
```

Запуск скриптов с помощью оболочки

В дополнение к интерактивному использованию оболочки вы также можете передавать оболочке файлы JavaScript для выполнения. Просто передайте свои скрипты в командной строке:

```
$ mongo script1.js script2.js script3.js
MongoDB shell version: 4.2.1
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 4.2.1

loading file: script1.js
I am script1.js
loading file: script2.js
I am script2.js
loading file: script3.js
I am script3.js
...
```

Оболочка *mongo* выполнит каждый из перечисленных скриптов и завершит работу. Если вы хотите запустить скрипт, используя соединение с хостом/портом *mongod* не по умолчанию, для начала укажите адрес, а затем скрипт (скрипты):

```
$ mongo server-1:30000/foo --quiet script1.js script2.js script3.js
```

Будут выполнены три скрипта, при этом для `db` будет установлена база данных *foo* на сервере *1:30000*.

В сценариях можно вывести информацию на экран (как это делали предыдущие скрипты), используя функцию `print`. Это позволяет использовать оболочку как часть конвейера команд. Если вы планируете перенаправлять вывод скрипта оболочки в другую команду, используйте параметр `--quiet`, чтобы запретить вывод баннера «MongoDB shell version v4.2.0».

Вы также можете запускать скрипты из интерактивной оболочки, используя функцию `load`:

```
> load("script1.js")
I am script1.js
true
>
```

У скриптов есть доступ к переменной db (как и к любой другой глобальной переменной). Однако такие команды оболочки, как use db или show collection, не работают с файлами. Существуют допустимые эквиваленты JavaScript для каждого из них, как показано в табл. 2.1.

Таблица 2.1. JavaScript-эквиваленты команд-помощников оболочки

Команда	Эквивалент
use video	db.getSisterDB("video")
show dbs	db.getMongo().getDBs()
show collections	db.getCollectionNames()

Вы также можете использовать скрипты для ввода переменных в оболочку. Например, у вас может быть скрипт, просто инициализирующий вспомогательные функции, которые вы обычно используете. Приведенный ниже скрипт, например, может быть полезен для части III и части IV. Он определяет функцию connectTo, которая подключается к локально работающей базе данных на заданном порту и устанавливает db для этого подключения:

```
// defineConnectTo.js

/**
 *Подключение к базе данных и установка переменной db.
 */
var connectTo = function(port, dbname) {
    if (!port) {
        port = 27017;
    }

    if (!dbname) {
        dbname = "test";
    }
    db = connect("localhost:"+port+"/"+dbname);
    return db;
};
```

Если вы загрузите этот скрипт в оболочку, функция connectTo теперь определена:

```
> typeof connectTo
undefined
> load('defineConnectTo.js')
> typeof connectTo
function
```

Помимо добавления вспомогательных функций, вы можете использовать скрипты для автоматизации распространенных задач и действий по администрированию.

По умолчанию оболочка будет выполнять поиск в каталоге, в котором вы ее запустили (используйте функцию `pwd()`, чтобы увидеть, что это за каталог). Если в вашем текущем каталоге скрипта нет, можно указать оболочке относительный или абсолютный путь к нему. Например, если вы хотите поместить свои скрипты оболочки в *~/my-scripts*, можно скачать файл *defineConnectTo.js* с помощью следующей строки кода: `load("/home/myUser/my-scripts/defineConnectTo.js")`. Обратите внимание, что метод `load` не может разрешить ~.

Можно использовать метод `run` для запуска программ командной строки из оболочки. Вы можете передать аргументы функции в качестве параметров:

```
> run("ls", "-l", "/home/myUser/my-scripts/")
sh70352| -rw-r--r-- 1 myUser myUser 2012-12-13 13:15 defineConnectTo.js
sh70532| -rw-r--r-- 1 myUser myUser 2013-02-22 15:10 script1.js
sh70532| -rw-r--r-- 1 myUser myUser 2013-02-22 15:12 script2.js
sh70532| -rw-r--r-- 1 myUser myUser 2013-02-22 15:13 script3.js
```

Обычно такого рода использование ограничено, поскольку вывод форматируется странным образом и не поддерживает конвейеры.

Создание файла .mongorc.js

Если вы часто загружаете скрипты, можно поместить их в файл *.mongorc.js*. Этот файл запускается всякий раз, когда вы запускаете оболочку.

Например, предположим, что вы хотите, чтобы оболочка приветствовала вас при входе в систему. Создайте файл с именем *.mongorc.js* в своем домашнем каталоге, а затем добавьте в него следующие строки:

```
// .mongorc.js

var compliment = ["attractive", "intelligent", "like Batman"];
var index = Math.floor(Math.random()*3);

print("Hello, you're looking particularly "+compliment[index]+" today!");
```

Затем, когда вы запустите оболочку, вы увидите нечто вроде этого:

```
$ mongo
MongoDB shell version: 4.2.1
connecting to: test
Hello, you're looking particularly like Batman today!
>
```

С практической точки зрения можно использовать этот сценарий для установки любых глобальных переменных, которые вы хотели бы использовать, заменять длинные имена на более короткие и переопределять встроенные функции. Одним из наиболее распространенных применений файла *.mongorc.js* является удаление некоторых из наиболее «опасных» вспомогательных функций оболочки. Можно переопределить такие функции как `dropDatabase` или `deleteIndexes` с помощью фиктивных команд или вообще отменить их определение:

```
var no = function() {
    print("Not on my watch.");
};

// Предотвращаем удаление баз данных;
db.dropDatabase = DB.prototype.dropDatabase = no;

// Предотвращаем удаление коллекций;
DBCollection.prototype.drop = no;

// Предотвращаем удаление индекса;
DBCollection.prototype.dropIndex = no;

// Предотвращаем удаление индексов;
DBCollection.prototype.dropIndexes = no;
```

Теперь, если вы попытаетесь вызвать любую из этих функций, она просто выведет сообщение об ошибке. Обратите внимание, что этот метод не защищает вас от злонамеренных пользователей. Он может помочь только в случае, если вы по ошибке нажали лишние клавиши.

Вы можете отключить загрузку своего файла *.mongorc.js*, используя опцию `--norc` при запуске оболочки.

Настройка приглашения

Приглашение оболочки по умолчанию можно переопределить путем установки переменной `prompt` в строку либо в функцию. Например, если вы делаете запрос, выполнение которого занимает несколько минут, вы, вероятно, захотите, чтобы у вас было приглашение к вводу, отображающее текущее время, и вы могли видеть, когда была завершена последняя операция:

```
prompt = function() {
    return (new Date())+"> ";
};
```

Еще один удобный вариант, где показана текущая база данных, которую вы используете:

```
prompt = function() {
    if (typeof db == 'undefined') {
        return '(nodb)> ';
    }

    // Проверка последней операции;
    try {
        db.runCommand({getLastError:1});
    }
    catch (e) {
        print(e);
    }

    return db+"> ";
};
```

Обратите внимание, что такие функции должны возвращать строки и быть очень осторожными при перехвате исключений: это может быть очень запутанным, если ваша подсказка превращается в исключение!

В целом такая функция должна включать в себя вызов `getLastError`, что позволяет ловить ошибки при записи и повторно соединять вас автоматически, если оболочка отключается (например, если вы перезапустите *mongod*).

Файл *.mongorc.js* является неплохим местом для установки приглашения, если вы хотите всегда использовать пользовательский вариант (или настроить пару пользовательских вариантов, между которыми можно переключаться в оболочке).

Редактирование сложных переменных

Многострочная поддержка в оболочке несколько ограничена: нельзя редактировать предыдущие строки, что может раздражать, когда вы понимаете, что первая строка содержит опечатку, и в настоящее время вы работаете со строкой 15. Таким образом, в случае с более крупными блоками кода или объектами у вас, возможно, возникнет желание редактировать их в редакторе. Для этого установите в оболочке переменную `EDITOR` (или в вашем окружении, но поскольку вы уже находитесь в оболочке…):

```
> EDITOR="/usr/bin/emacs"
```

Теперь, если вы хотите отредактировать переменную, можно использовать это: *`edit имяпеременной`* – например:

```
> var wap = db.books.findOne({title: "War and Peace"});
> edit wap
```

Когда вы закончите вносить изменения, сохраните результаты и выйдите из редактора. Переменная будет проанализирована и загружена обратно в оболочку.

Добавьте `EDITOR="/path/to/editor";` в свой файл *.mongorc.js*, и вам не придется беспокоиться о повторной настройке.

Неудобные имена коллекций

Извлечение коллекции с использованием синтаксиса `db.`*`Имяколлекции`* почти всегда работает, если только имя коллекции не является зарезервированным словом или недопустимым именем свойства JavaScript.

Например, предположим, что мы пытаемся получить доступ к коллекции *version*. Мы не можем использовать `db.version`, потому что `db.version` – это метод, используемый при работе с базой данных (он возвращает версию работающего сервера MongoDB):

```
> db.version
function () {
return this.serverBuildInfo().version;
}
```

Чтобы получить доступ конкретно к коллекции *version*, нужно использовать функцию `getCollection`:

```
> db.getCollection("version");
test.version
```

Такой вариант также можно использовать для имен коллекций с символами, которые не являются допустимыми именами свойств JavaScript, таких как *foo-bar-baz* и *123abc* (имена свойств JavaScript могут содержать только буквы, цифры, символы *$* и _ и не могут начинаться с цифры).

Еще один способ обойти недействительные свойства – использовать синтаксис доступа к массиву. В JavaScript `x.y` идентично `x['y']`. Это означает, что к вложенным коллекциям можно получить доступ с использованием переменных, а не только литеральных имен. Таким образом, если вам нужно выполнить какую-либо операцию для каждой вложенной коллекции *blog*, вы можете перебирать их примерно так:

```
var collections = ["posts", "comments", "authors"];
for (var i in collections) {
print(db.blog[collections[i]]);
}
```

вместо:

```
print(db.blog.posts);
print(db.blog.comments);
print(db.blog.authors);
```

Обратите внимание, что нельзя использовать `db.blog.i`, что будет интерпретироваться как `test.blog.i`, а не `test.blog.posts`. Вы должны использовать синтаксис `db.blog[i]`, чтобы `i` интерпретировалась как переменная.

Вы можете применять эту технику для доступа к коллекциям с нескладными именами:

```
> var name = "@#&!"
> db[name].find()
```

Попытка запроса `db.@#&!` будет недопустимой, но `db[name]` будет работать.

Глава 3

Создание, обновление и удаление документов

В этой главе рассматриваются основы перемещения данных в базу данных и из нее, включая следующие действия:

- добавление новых документов в коллекцию;
- удаление документов из коллекции;
- обновление существующих документов;
- выбор правильного уровня безопасности в зависимости от скорости всех этих операций.

Вставка документов

Вставка – основной метод добавления данных в MongoDB. Чтобы вставить один документ, используйте метод коллекции `insertOne`:

```
> db.movies.insertOne({"title" : "Stand by Me"})
```

`insertOne` добавит в документ ключ `"_id"` (если вы его не предоставили) и сохранит документ в MongoDB.

insertMany

Если вам нужно вставить несколько документов в коллекцию, можно использовать метод `insertMany`. Этот метод позволяет передавать массив документов в базу данных, что гораздо более эффективно, потому что ваш код не будет бегать в базу данных и обратно ради каждого документа, а вставит их все сразу.

В оболочке можно опробовать это следующим образом:

```
> db.movies.drop()
true
> db.movies.insertMany([{"title" : "Ghostbusters"},
...                       {"title" : "E.T."},
```

```
...                          {"title" : "Blade Runner"}]);
{
      "acknowledged" : true,
       "insertedIds" : [
           ObjectId("572630ba11722fac4b6b4996"),
           ObjectId("572630ba11722fac4b6b4997"),
           ObjectId("572630ba11722fac4b6b4998")
       ]
}
> db.movies.find()
{ "_id" : ObjectId("572630ba11722fac4b6b4996"), "title" : "Ghostbusters" }
{ "_id" : ObjectId("572630ba11722fac4b6b4997"), "title" : "E.T." }
{ "_id" : ObjectId("572630ba11722fac4b6b4998"), "title" : "Blade Runner" }
```

Отправка десятков, сотен или даже тысяч документов за один раз может значительно ускорить процесс вставки.

Метод `insertMany` полезен, если вы вставляете несколько документов в одну коллекцию. Если вы просто импортируете необработанные данные (например, из потока данных или MySQL), для этого существуют инструменты командной строки, такие как *mongoimport*, которые можно использовать вместо массовой вставки. С другой стороны, часто удобно разбирать данные перед тем, как сохранить их в MongoDB (преобразовывая даты в тип даты или добавляя пользовательский `"_id"`, например). В таких случаях метод `insertMany` можно использовать и для импорта данных.

Текущие версии MongoDB не принимают сообщения длиннее 48 МБ, поэтому существует ограничение на количество, которое можно поместить в одну массовую вставку. Если вы попытаетесь вставить более 48 МБ, многие драйверы разделят вставку на несколько вставок по 48 МБ. Ознакомьтесь с документацией к вашему драйверу для получения подробной информации.

При выполнении массовой вставки с использованием метода `insertMany`, если документ посреди массива выдает какую-либо ошибку, что произойдет, зависит от того, выбрали ли вы упорядоченные или неупорядоченные операции. В качестве второго параметра `insertMany` можно указать документ опций. Укажите значение `true` для ключа `"ordered"` в документе параметров, чтобы обеспечить вставку документов в том порядке, в котором они были предоставлены. Укажите значение `false`, и MongoDB может изменить порядок вставок для повышения производительности. Упорядоченные вставки используются по умолчанию, если порядок не указан. В случае с упорядоченными вставками массив, переданный в `insertMany`, определяет порядок вставки. Если документ вызывает ошибку вставки, ни один документ за пределами этой точки в массиве не будет вставлен. В случае с неупорядоченными вставками MongoDB попытается вставить все документы независимо от того, приводят ли некоторые вставки к ошибкам.

В этом примере, поскольку упорядоченные вставки используются по умолчанию, будут вставлены только первые два документа. Третий документ выдаст ошибку, потому что нельзя вставить два документа с одинаковым `"_id"`:

```
> db.movies.insertMany([
    ... {"_id" : 0, "title" : "Top Gun"},
    ... {"_id" : 1, "title" : "Back to the Future"},
    ... {"_id" : 1, "title" : "Gremlins"},
    ... {"_id" : 2, "title" : "Aliens"}])
2019-04-22T12:27:57.278-0400 E QUERY [js] BulkWriteError: write
error at item 2 in bulk operation :
BulkWriteError({
    "writeErrors" : [
        {
          "index" : 2,
          "code" : 11000,
          "errmsg" : "E11000 duplicate key error collection:
          test.movies index: _id_ dup key: { _id: 1.0 }",
          "op" : {
                  "_id" : 1,
                  "title" : "Gremlins"
          }
       }
    ],
    "writeConcernErrors" : [ ],
    "nInserted" : 2,
    "nUpserted" : 0,
    "nMatched" : 0,
    "nModified" : 0,
    "nRemoved" : 0,
    "upserted" : [ ]
})
BulkWriteError@src/mongo/shell/bulk_api.js:367:48
BulkWriteResult/this.toError@src/mongo/shell/bulk_api.js:332:24
Bulk/this.execute@src/mongo/shell/bulk_api.js:1186:23
DBCollection.prototype.insertMany@src/mongo/shell/crud_api.js:314:5
@(shell):1:1
```

Если вместо этого мы указываем неупорядоченные вставки, первый, второй и четвертый документы в массиве вставляются. Единственная неудачная вставка – третий документ, опять же по причине дублирующейся ошибки `"_id"`:

```
> db.movies.insertMany([
... {"_id" : 3, "title" : "Sixteen Candles"},
... {"_id" : 4, "title" : "The Terminator"},
... {"_id" : 4, "title" : "The Princess Bride"},
... {"_id" : 5, "title" : "Scarface"}],
... {"ordered" : false})
2019-05-01T17:02:25.511-0400 E QUERY [thread1] BulkWriteError: write
error at item 2 in bulk operation :
BulkWriteError({
    "writeErrors" : [
        {
          "index" : 2,
          "code" : 11000,
          "errmsg" : "E11000 duplicate key error index: test.movies.$_id_
          dup key: { : 4.0 }",
          "op" : {
                  "_id" : 4,
                  "title" : "The Princess Bride"
          }
        }
    ],
    "writeConcernErrors" : [ ],
    "nInserted" : 3,
    "nUpserted" : 0,
    "nMatched" : 0,
    "nModified" : 0,
    "nRemoved" : 0,
    "upserted" : [ ]
})
BulkWriteError@src/mongo/shell/bulk_api.js:367:48
BulkWriteResult/this.toError@src/mongo/shell/bulk_api.js:332:24
Bulk/this.execute@src/mongo/shell/bulk_api.js:1186.23
DBCollection.prototype.insertMany@src/mongo/shell/crud_api.js:314:5
@(shell):1:1
```

Если вы внимательно изучите эти примеры, то можете заметить, что выходные данные двух этих вызовов `insertMany` намекают на то, что в случае с массовыми записями могут поддерживаться и другие операции, помимо простых вставок. Хотя метод `insertMany` не поддерживает операции, отличные от вставки, MongoDB поддерживает специальный API, который позволяет группировать некое число операций разных типов в одном вызове. Хотя это и выходит за рамки данной главы, информацию об этом API можно найти (https://docs.mongodb.com/manual/core/bulk-write-operations/) в документации по MongoDB.

Проверка вставки

MongoDB выполняет минимальные проверки вставляемых данных: она проверяет базовую структуру документа и добавляет поле `"_id"`, если оно не существует. Одной из проверок базовой структуры является размер: все документы должны быть меньше 16 МБ. Это несколько произвольный предел (и может быть повышен в будущем); в основном это предназначено для предотвращения неправильного проектирования схемы и обеспечения стабильной производительности. Чтобы увидеть двоичный бинарный размер JSON(BSON)-документа *doc* в байтах, выполните функцию `Object.bsonsize(`*doc*`)` из оболочки.

Чтобы у вас было представление о том, сколько данных составляет 16 МБ, весь текст романа «*Война и мир*» составляет всего 3,14 МБ.

Эти минимальные проверки также означают, что довольно просто вставить неверные данные (если вы пытаетесь это сделать). Таким образом, нужно разрешать только надежные источники, такие как серверы приложений, для подключения к базе данных. Все драйверы MongoDB для основных языков (и большинство второстепенных тоже) проверяют различные недействительные данные (документы, которые слишком большие, содержат строки в кодировке, отличной от UTF-8, или используют нераспознанные типы), перед тем как отправить что-либо в базу данных.

insert

До появления версии MongoD 3.0 `insert` был основным методом вставки документов в MongoDB. Драйверы MongoDB представили новый CRUD API одновременно с выпуском сервера MongoDB 3.0. Начиная с версии 3.2 оболочка *mongo* тоже поддерживает этот API, который включает в себя методы `insertOne` и `insertMany`, а также несколько других методов. Цель текущего CRUD API – сделать семантику всех операций CRUD согласованными и понятными для драйверов и оболочки. Хотя такие методы, как `insert`, по-прежнему поддерживаются ради обратной совместимости, их не следует использовать в будущих приложениях. Вместо этого стоит отдать предпочтение методам `insertOne` и `insertMany` для создания документов.

Удаление документов

Теперь, когда в нашей базе данных есть данные, давайте удалим их. Для этой цели CRUD API предоставляет методы `deleteOne` и `deleteMany`. Оба этих метода принимают документ фильтра в качестве первого параметра. Фильтр задает набор критериев для сопоставления при удалении документов. Чтобы удалить документ со значением `"_id"`, равным 4, мы используем метод `deleteOne` в оболочке *mongo*, как показано здесь:

```
> db.movies.find()
{ "_id" : 0, "title" : "Top Gun"}
{ "_id" : 1, "title" : "Back to the Future"}
{ "_id" : 3, "title" : "Sixteen Candles"}
{ "_id" : 4, "title" : "The Terminator"}
{ "_id" : 5, "title" : "Scarface"}
> db.movies.deleteOne({"_id" : 4})
{ "acknowledged" : true, "deletedCount" : 1 }
> db.movies.find()
{ "_id" : 0, "title" : "Top Gun"}
{ "_id" : 1, "title" : "Back to the Future"}
{ "_id" : 3, "title" : "Sixteen Candles"}
{ "_id" : 5, "title" : "Scarface"}
```

В этом примере мы использовали фильтр, который может соответствовать только одному документу, поскольку значения `"_id"` уникальны в коллекции. Однако мы также можем указать фильтр, который соответствует нескольким документам в коллекции. В этом случае метод `deleteOne` удалит первый найденный документ, который соответствует фильтру. Какой документ будет найден первым, зависит от нескольких факторов, в том числе от порядка, в котором были вставлены документы, того, какие обновления были внесены в них (для некоторых подсистем хранения), и того, какие индексы указаны. Как и при любой операции с базой данных, убедитесь, что вы знаете, как использование метода `deleteOne` повлияет на ваши данные.

Чтобы удалить все документы, которые соответствуют фильтру, используйте метод `deleteMany`:

```
> db.movies.find()
{ "_id" : 0, "title" : "Top Gun", "year" : 1986 }
{ "_id" : 1, "title" : "Back to the Future", "year" : 1985 }
{ "_id" : 3, "title" : "Sixteen Candles", "year" : 1984 }
{ "_id" : 4, "title" : "The Terminator", "year" : 1984 }
{ "_id" : 5, "title" : "Scarface", "year" : 1983 }
> db.movies.deleteMany({"year" : 1984})
{ "acknowledged" : true, "deletedCount" : 2 }
> db.movies.find()
{ "_id" : 0, "title" : "Top Gun", "year" : 1986 }
{ "_id" : 1, "title" : "Back to the Future", "year" : 1985 }
{ "_id" : 5, "title" : "Scarface", "year" : 1983 }
```

В качестве более реалистичного варианта использования предположим, что вы хотите удалить каждого пользователя из коллекции *mailing.list*, где значение `"opt-out"` равно `true`:

```
> db.mailing.list.deleteMany({"opt-out" : true})
```

В версиях MongoDB до 3.0 `remove` был основным методом удаления документов. Драйверы MongoDB представили методы `deleteOne` и `deleteMany` одновременно с выпуском сервера MongoDB 3.0, и оболочка начала поддерживать эти методы в версии 3.2. Хотя метод `remove` все еще поддерживается по причине обратной совместимости, в своих приложениях следует использовать методы `deleteOne` и `deleteMany`. Текущий CRUD API предоставляет более чистый набор семантики и, особенно в случае с операциями с несколькими документами, помогает разработчикам приложений избежать распространенных ошибок с предыдущим API.

drop

Можно использовать метод `deleteMany`, чтобы удалить все документы в коллекции:

```
> db.movies.find()
{ "_id" : 0, "title" : "Top Gun", "year" : 1986 }
{ "_id" : 1, "title" : "Back to the Future", "year" : 1985 }
{ "_id" : 3, "title" : "Sixteen Candles", "year" : 1984 }
{ "_id" : 4, "title" : "The Terminator", "year" : 1984 }
{ "_id" : 5, "title" : "Scarface", "year" : 1983 }
> db.movies.deleteMany({})
{ "acknowledged" : true, "deletedCount" : 5 }
> db.movies.find()
```

Удаление документов обычно является довольно быстрой операцией. Однако если вы хотите очистить всю коллекцию, ее быстрее удалить с помощью метода `drop`:

```
> db.movies.drop()
true
```

а затем воссоздать любые индексы в пустой коллекции.

Как только данные удаляются, они исчезают навсегда. Операции `drop` и `delete` невозможно отменить, равно как нельзя восстановить удаленные документы, разумеется, если только речь не идет о восстановлении предварительно сохраненной версии данных. *См.* главу 23, где подробно обсуждается резервное копирование и восстановление данных в MongoDB.

Обновление документов

После сохранения документа в базе данных его можно изменить с помощью одного из нескольких методов обновления: `updateOne`, `updateMany` и `replaceOne`. Методы `updateOne` и `updateMany` принимают документ фильтра в качестве первого параметра и документ модификатора, описывающий

изменения, которые необходимо внести, в качестве второго параметра. Метод `replaceOne` также принимает фильтр в качестве первого параметра, а в качестве второго параметра ожидает документ, которым он заменит документ, соответствующий фильтру.

Обновление документа является атомарным: если два обновления происходят одновременно, будет применено то из них, которое достигнет сервера первым, а затем будет применено следующее. Таким образом, конфликтующие обновления можно безопасно отправлять в быстрой последовательности, не опасаясь повреждения каких-либо документов: последнее обновление «победит». Стоит рассмотреть шаблон версионности документов (*см.*раздел «Шаблоны проектирования схем»), если вам не нужно поведение по умолчанию.

Замена документа

Метод `replaceOne` полностью заменяет соответствующий документ новым. Это может быть полезно для драматической схемы миграции (*см.* главу 9, где рассказывается о стратегиях миграции схем). Например, предположим, что мы вносим серьезные изменения в пользовательский документ, который выглядит следующим образом:

```
{
  "_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
  "name" : "joe",
  "friends" : 32,
  "enemies" : 2
}
```

Мы хотим переместить поля `"friends"` и `"enemies"` в поддокумент `"relationships"`. Мы можем изменить структуру документа в оболочке, а затем заменить версию базы данных с помощью `replaceOne`:

```
> var joe = db.users.findOne({"name" : "joe"});
> joe.relationships = {"friends" : joe.friends, "enemies" : joe.enemies};
{
  "friends" : 32,
  "enemies" : 2
}
> joe.username = joe.name;
"joe"
> delete joe.friends;
true
> delete joe.enemies;
true
> delete joe.name;
```

```
true
> db.users.replaceOne({"name" : "joe"}, joe);
```

Теперь при использовании метода `findOne` видно, что структура документа была обновлена:

```
{
   "_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
   "username" : "joe",
   "relationships" : {
       "friends" : 32,
       "enemies" : 2
   }
}
```

Распространенной ошибкой является сопоставление более чем одного документа с критериями, а затем создание дублирующего значения `"_id"` со вторым параметром. База данных выдаст ошибку, и ни один из документов не будет обновлен.

Например, предположим, что мы создали несколько документов с одинаковым значением `"name"`, но мы этого не понимаем:

```
> db.people.find()
{"_id" : ObjectId("4b2b9f67a1f631733d917a7b"), "name" : "joe", "age" : 65}
{"_id" : ObjectId("4b2b9f67a1f631733d917a7c"), "name" : "joe", "age" : 20}
{"_id" : ObjectId("4b2b9f67a1f631733d917a7d"), "name" : "joe", "age" : 49}
```

Теперь, если у Джо № 2 день рождения, мы хотим увеличить значение его ключа `"age"`, поэтому можно написать следующее:

```
> joe = db.people.findOne({"name" : "joe", "age" : 20});
{
   "_id" : ObjectId("4b2b9f67a1f631733d917a7c"),
   "name" : "joe",
   "age" : 20
}
> joe.age++;
> db.people.replaceOne({"name" : "joe"}, joe);
E11001 duplicate key on update
```

Что произошло? Когда вы выполните обновление, база данных будет искать документ, соответствующий `{"name" : "joe"}`. Первым она найдет 65-летнего Джо. Она попытается заменить этот документ на тот, что содержится в переменной `joe`, но в этой коллекции уже есть документ с таким же `"_id"`. Таким образом, обновление завершится неудачно, поскольку значения `"_id"` должны быть уникальными. Лучший способ избежать подобной

ситуации – убедиться, что в вашем обновлении всегда указан уникальный документ, возможно, путем сопоставления с ключом, подобным `"_id"`. В случае с предыдущим примером это обновление будет правильным:

```
> db.people.replaceOne({"_id" : ObjectId("4b2b9f67a1f631733d917a7c")}, joe)
```

Использование `"_id"` для фильтра также будет эффективным, поскольку значения `"_id"` формируют основу для первичного индекса коллекции. Мы рассмотрим первичные и вторичные индексы и то, как индексация влияет на обновления и другие операции, подробнее в главе 5.

Использование операторов обновления

Обычно нужно обновить только определенные части документа. Вы можете обновить определенные поля в документе, используя *операторы атомарного обновления*. Операторы обновления – это специальные ключи, которые можно применять для указания сложных операций обновления, таких как изменение, добавление или удаление ключей, и даже манипулирование массивами и встраиваемыми документами.

Предположим, мы храним аналитику веб-сайта в коллекции и хотим, чтобы счетчик увеличивался каждый раз, когда кто-то посещает страницу. Мы можем использовать операторы обновления, чтобы делать это увеличение атомарно. Каждый URL-адрес и количество просмотров страниц хранятся в документе, который выглядит следующим образом:

```
{
   "_id" : ObjectId("4b253b067525f35f94b60a31"),
   "url" : "www.example.com",
   "pageviews" : 52
}
```

Каждый раз, когда кто-то заходит на страницу, мы можем найти эту страницу по URL-адресу и использовать модификатор `"$inc"`, чтобы увеличить значение ключа `"pageviews"`:

```
> db.analytics.updateOne({"url" : "www.example.com"},
... {"$inc" : {"pageviews" : 1}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

Теперь если мы воспользуемся методом `findOne`, то увидим, что количество просмотров страниц увеличилось на единицу:

```
> db.analytics.findOne()
{
   "_id" : ObjectId("4b253b067525f35f94b60a31"),
   "url" : "www.example.com",
```

```
    "pageviews" : 53
}
```

При использовании операторов значение `"_id"` нельзя изменить. (Обратите внимание, что `"_id"` *можно* изменить путем замены всего документа.) Можно изменить значения для любого другого ключа, включая иные ключи с уникальной индексацией.

Начало работы с модификатором "$set"

Модификатор `"$set"` устанавливает значение поля. Если поле еще не существует, оно будет создано. Это может быть удобно для обновления схем или добавления пользовательских ключей. Например, предположим, что у вас есть простой профиль пользователя, сохраненный в виде документа, который выглядит примерно так:

```
> db.users.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "name" : "joe",
    "age" : 30,
    "sex" : "male",
    "location" : "Wisconsin"
}
```

Это довольно простой профиль. Если пользователь хочет сохранить свою любимую книгу в своем профиле, он может добавить ее, используя модфикатор `"$set"`:

```
> db.users.updateOne({"_id" : ObjectId("4b253b067525f35f94b60a31")},
... {"$set" : {"favorite book" : "War and Peace"}})
```

Теперь у документа будет ключ `"favorite book"`:

```
> db.users.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "name" : "joe",
    "age" : 30,
    "sex" : "male",
    "location" : "Wisconsin",
    "favorite book" : "War and Peace"
}
```

Если пользователь решает, что ему по-настоящему нравится другая книга, можно снова использовать модификатор `"$set"` для изменения значения:

```
> db.users.updateOne({"name" : "joe"},
... {"$set" : {"favorite book" : "Green Eggs and Ham"}})
```

`"$set"` может даже изменить тип ключа, который он модифицирует. Например, если наш непостоянный пользователь решит, что ему на самом деле нравится всего несколько книг, он может изменить значение ключа `"favorite book"` в массив:

```
> db.users.updateOne({"name" : "joe"},
... {"$set" : {"favorite book" :
... ["Cat's Cradle", "Foundation Trilogy", "Ender's Game"]}})
```

Если пользователь понимает, что ему на самом деле не нравится чтение, он может полностью удалить ключ с помощью `"$unset"`:

```
> db.users.updateOne({"name" : "joe"},
... {"$unset" : {"favorite book" : 1}})
```

Теперь документ будет таким же, каким он был в начале этого примера.

Модификатор `"$set"` также можно использовать для доступа к вложенным документам и изменять их:

```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "title" : "A Blog Post",
    "content" : "...",
    "author" : {
        "name" : "joe",
        "email" : "joe@example.com"
    }
}
> db.blog.posts.updateOne({"author.name" : "joe"},
... {"$set" : {"author.name" : "joe schmoe"}})
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "title" : "A Blog Post",
    "content" : "...",
    "author" : {
        "name" : "joe schmoe",
        "email" : "joe@example.com"
    }
}
```

Всегда нужно использовать $-модификатор для добавления, изменения или удаления ключей. Распространенная ошибка, которую некоторые со-

вершают поначалу, состоит в том, что они пытаются установить значение ключа на какое-то другое значение, выполняя обновление, подобное этому:

```
> db.blog.posts.updateOne({"author.name" : "joe"},
... {"author.name" : "joe schmoe"})
```

Это приведет к ошибке. Документ обновления должен содержать операторы обновления. Предыдущие версии CRUD API не перехватывали этот тип ошибки. Более ранние методы обновления просто выполняли замену всего документа в подобных ситуациях. Именно такой тип ловушек и привел к созданию нового CRUD API.

Инкрементирование и декрементирование

Оператор `"$inc"` можно использовать для изменения значения существующего ключа или для создания нового ключа, если он еще не существует. Это полезно для обновления аналитики, кармы, голосов или чего-либо еще, имеющего изменяемое числовое значение.

Предположим, мы создаем коллекцию игр, в которой хотим сохранять игры и обновлять оценки по мере их изменения. Когда пользователь начинает играть, скажем, в пейнтбол, мы можем вставить документ, который идентифицирует эту игру по имени и пользователю, который в нее играет:

```
> db.games.insertOne({"game" : "pinball", "user" : "joe"})
```

Когда мяч попадает в бампер, счет должен расти. Поскольку очки в пейнтболе начисляются довольно свободно, допустим, что базовая единица очков, которую игрок может заработать, равна 50. Мы можем использовать модификатор `"$inc"`, чтобы добавить 50 к счету игрока:

```
> db.games.updateOne({"game" : "pinball", "user" : "joe"},
... {"$inc" : {"score" : 50}})
```

Если мы посмотрим на документ после этого обновления, то увидим следующее:

```
> db.games.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "game" : "pinball",
    "user" : "joe",
    "score" : 50
}
```

Ключа `"score"` еще не было, поэтому он был создан с помощью модификатора `"$inc"`, и для него было установлено значение, равное сумме приращения: 50.

Если мяч попадает в «бонусный» слот, нужно добавить к счету 10 000 очков. Это можно сделать, передав "$inc" другое значение:

```
> db.games.updateOne({"game" : "pinball", "user" : "joe"},
... {"$inc" : {"score" : 10000}})
```

Теперь если мы посмотрим на игру, то увидим следующее:

```
> db.games.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "game" : "pinball",
    "user" : "joe",
    "score" : 10050
}
```

Ключ "score" существовал и имел числовое значение, поэтому сервер добавил к нему 10 000. Модификатор "$inc" похож на "$set", но он предназначен для увеличения (и уменьшения) чисел. Его можно использовать только для значений типа integer, long, double или decimal. Если он используется для любого другого типа значения, это окончится неудачей. Сюда входят типы, которые многие языки будут автоматически преобразовывать в числа, такие как нули, логические значения или строки из числовых символов:

```
> db.strcounts.insert({"count" : "1"})
WriteResult({ "nInserted" : 1 })
> db.strcounts.update({}, {"$inc" : {"count" : 1}})
WriteResult({
    "nMatched" : 0,
    "nUpserted" : 0,
    "nModified" : 0,
    "writeError" : {
        "code" : 16837,
        "errmsg" : "Cannot apply $inc to a value of non-numeric type.
        {_id: ObjectId('5726c0d36855a935cb57a659')} has the field 'count' of
        non-numeric type String"
    }
})
```

Кроме того, значение ключа "$inc" должно быть числом. Нельзя увеличивать на строку, массив или другое нечисловое значение. В результате появится сообщение об ошибке «Допускается использование модификатора "$inc" только с числами». Чтобы модифицировать другие типы, используйте модификатор "$set" или один из следующих операторов массива.

Операторы массива

Для манипулирования массивами существует обширный класс операторов обновления. Массивы представляют собой распространенные и мощные структуры данных: это не только списки, на которые можно ссылаться по индексу, они также могут удваиваться как наборы.

Добавление элементов. Оператор `"$push"` добавляет элементы в конец массива, если массив существует, и создает новый массив, если его нет. Например, предположим, что мы сохраняем посты из блога и хотим добавить ключ `"comments"`, содержащий массив. Мы можем вставить комментарий в несуществующий массив `"comments"`, который создаст массив и добавит комментарий:

```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "..."
}
> db.blog.posts.updateOne({"title" : "A blog post"},
... {"$push" : {"comments" :
...    {"name" : "joe", "email" : "joe@example.com",
...    "content" : "nice post."}}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "joe",
            "email" : "joe@example.com",
            "content" : "nice post."
        }
    ]
}
```

Теперь, если мы хотим добавить еще один комментарий, мы можем просто снова использовать `"$push"`:

```
> db.blog.posts.updateOne({"title" : "A blog post"},
... {"$push" : {"comments" :
... {"name" : "bob", "email" : "bob@example.com",
... "content" : "good post."}}})
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "joe",
            "email" : "joe@example.com",
            "content" : "nice post."
        },
        {
            "name" : "bob",
            "email" : "bob@example.com",
            "content" : "good post."
        }
    ]
}
```

Это «простая» форма оператора `"push"`, но вы можете использовать ее и для более сложных операций с массивами. Язык запросов MongoDB предоставляет модификаторы для некоторых операторов, включая `"$push"`. Вы можете сдвинуть несколько значений за одну операцию, используя модификатор `"$each"` для оператора `"$push"`:

```
> db.stock.ticker.updateOne({"_id" : "GOOG"},
... {"$push" : {"hourly" : {"$each" : [562.776, 562.790, 559.123]}}})
```

В результате этого в массив будет добавлено три новых элемента.

Если вы хотите, чтобы массив увеличивался до определенной длины, вы можете использовать модификатор `"$slice"` с `"$push"`, чтобы предотвратить рост массива выше определенного размера, успешно создавая список элементов `"top N"`:

```
> db.movies.updateOne({"genre" : "horror"},
... {"$push" : {"top10" : {"$each" : ["Nightmare on Elm Street", "Saw"],
...                        "$slice" : -10}}})
```

В этом примере мы ограничиваем массив последними 10 добавленными элементами.

Если массив меньше 10 элементов (после добавления), все элементы будут сохранены. Если массив больше 10 элементов, будут сохранены только последние 10 элементов. Таким образом, `"$slice"` можно использовать для создания очереди в документе.

Наконец, можно применять модификатор `"$sort"` к операциям с `"$push"` перед усечением:

```
> db.movies.updateOne({"genre" : "horror"},
... {"$push" : {"top10" : {"$each" : [{"name" : "Nightmare on Elm Street",
...                                   "rating" : 6.6},
...                                  {"name" : "Saw", "rating" : 4.3}],
...                         "$slice" : -10,
...                         "$sort" : {"rating" : -1}}}})
```

Все объекты в массиве будут отсортированы по полю `"rating"`, и первые 10 останутся. Обратите внимание, что вы должны использовать модификатор `"$each"`; нельзя просто использовать модификаторы `"$slice"` или `"$sort"` с модификатором `"$push"` при работе с массивом.

Использование массивов в качестве наборов. Возможно, вы захотите рассматривать массив как набор, добавляя только значения, если они отсутствуют. Это можно сделать с помощью модификатора `"$ne"` в документе запроса. Например, чтобы вставить автора в список цитат, но только если его там еще нет, используйте следующий код:

```
> db.papers.updateOne({"authors cited" : {"$ne" : "Richie"}},
... {$push : {"authors cited" : "Richie"}})
```

Это также можно сделать с помощью `"$addToSet"`, что полезно в тех случаях, когда модификатор `"$ne"` не работает или же `"$addToSet"` лучше описывает то, что происходит.

Например, предположим, у вас есть документ, который представляет пользователя. У вас может быть набор адресов электронной почты, которые добавили пользователи:

```
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
        "joe@example.com",
        "joe@gmail.com",
        "joe@yahoo.com"
    ]
}
```

При добавлении еще одного адреса вы можете использовать модификатор `"$addToSet"` для предотвращения дублирования:

```
> db.users.updateOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")},
... {"$addToSet" : {"emails" : "joe@gmail.com"}})
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
        "joe@example.com",
        "joe@gmail.com",
        "joe@yahoo.com",
    ]
}
> db.users.updateOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")},
... {"$addToSet" : {"emails" : "joe@hotmail.com"}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
        "joe@example.com",
        "joe@gmail.com",
        "joe@yahoo.com",
        "joe@hotmail.com"
    ]
}
```

Вы также можете использовать его в сочетании с модификатором `"$each"` для добавления нескольких уникальных значений, что невозможно сделать с помощью комбинации `"$ne"/"$push"`. Например, можно использовать эти операторы, если пользователь хочет добавить несколько адресов электронной почты:

```
> db.users.updateOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")},
... {"$addToSet" : {"emails" : {"$each" :
... ["joe@php.net", "joe@example.com", "joe@python.org"]}}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
        "joe@example.com",
        "joe@gmail.com",
        "joe@yahoo.com",
        "joe@hotmail.com"
        "joe@php.net"
```

```
            "joe@python.org"
        ]
    }
```

Удаление элементов. Существует несколько способов удалить элементы из массива. Если вы хотите рассматривать массив как очередь или стек, можно использовать оператор `"$pop"`, который может удалять элементы с любого конца. `{"$pop" : {"key" : 1}}` удаляет элемент из конца массива. `{"$pop" : {"key" : -1}}` удаляет его с начала.

Иногда элемент должен быть удален на основе определенных критериев, а не своего положения в массиве. Оператор `"$pull"` используется для удаления элементов массива, соответствующих заданным критериям. Например, предположим, у нас есть список вещей, которые нужно сделать, но не в каком-то определенном порядке:

```
> db.lists.insertOne({"todo" : ["dishes", "laundry", "dry cleaning"]})
```

Если мы сначала занимается стиркой (`"laundry"`), можно удалить ее из списка следующим образом:

```
> db.lists.updateOne({}, {"$pull" : {"todo" : "laundry"}})
```

Теперь, если мы применим метод `findeOne()`, то увидим, что в массиве осталось только два элемента:

```
> db.lists.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "todo" : [
        "dishes",
        "dry cleaning"
    ]
}
```

При использовании оператора `"$pull"` удаляются все совпадающие документы, а не только одно совпадение. Если у вас есть массив, который выглядит как [1, 1, 2, 1], и вы удалите 1, то получите массив из одного элемента, [2].

Операторы массива могут использоваться только для ключей со значениями массива. Например, нельзя применять оператор `$push`, когда речь идет о целом числе, или использовать оператор `$pop`, когда перед вами строка. Используйте операторы `"$set"` или `"$inc"` для изменения скалярных значений.

Модификации позиционного массива. Манипулирование массивом становится немного сложнее, когда у вас есть несколько значений в мас-

сиве и вы хотите изменить некоторые из них. Есть два способа манипулирования значениями в массивах: по позиции или с помощью оператора позиции (символ $).

Массивы используют индексацию с отсчетом от 0, и элементы можно выбирать так, как если бы их индекс был ключом документа. Например, предположим, что у нас есть документ, содержащий массив с несколькими встраиваемыми документами, например пост в блоге с комментариями:

```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b329a216cc613d5ee930192"),
    "content" : "...",
    "comments" : [
        {
            "comment" : "good post",
            "author" : "John",
            "votes" : 0
        },
        {
            "comment" : "i thought it was too short",
            "author" : "Claire",
            "votes" : 3
        },
        {
            "comment" : "free watches",
            "author" : "Alice",
            "votes" : -5
        },
        {
            "comment" : "vacation getaways",
            "author" : "Lynn",
            "votes" : -7
        }
    ]
}
```

Если мы хотим увеличить количество голосов за первый комментарий, то можем написать следующее:

```
> db.blog.updateOne({"post" : post_id},
... {"$inc" : {"comments.0.votes" : 1}})
```

Однако во многих случаях мы не знаем, какой индекс массива нужно изменить, предварительно не запросив документ и не изучив его. Чтобы решить этот вопрос, в MongoDB существует позиционный оператор $, который определяет, какому элементу массива соответствует документ за-

проса, и обновляет этот элемент. Например, если у нас есть пользователь по имени Джон, который меняет свое имя на Джим, мы можем заменить его в комментариях с помощью позиционного оператора:

```
> db.blog.updateOne({"comments.author" : "John"},
... {"$set" : {"comments.$.author" : "Jim"}})
```

Позиционный оператор обновляет только первое совпадение. Таким образом, если бы Джон оставил несколько комментариев, его имя было бы изменено только для первого оставленного им комментария.

Обновления с использованием фильтров массива. В MongoDB версии 3.6 появился еще один параметр для обновления отдельных элементов массива: `arrayFilters`. Он позволяет изменять элементы массива, соответствующие конкретным критериям. Например, если мы хотим скрыть все комментарии с пятью или более отрицательными голосами, то можем сделать что-то вроде этого:

```
db.blog.updateOne(
  {"post" : post_id },
  { $set: { "comments.$[elem].hidden" : true } },
  {
    arrayFilters: [ { "elem.votes": { $lte: -5 } } ]
  }
)
```

Данная команда определяет `elem` как идентификатор для каждого совпадающего элемента в массиве `"comments"`. Если значение `votes` для комментария, обозначенного `elem`, меньше или равно –5, мы добавим поле с именем `"hidden"` в документ `"comments"` и установим для него значение `true`.

Upsert

Upsert (от англ. *update* (обновлять) + *insert* (вставить)) – особый тип обновления. Если не найден ни один документ, который соответствует фильтру, будет создан новый документ путем объединения критериев и обновленных документов. Если совпадающий документ найден, он будет обновлен в обычном режиме. Upsert'ы могут быть удобны, потому что могут избавить вас от необходимости «засевать» коллекцию: часто у вас может быть один и тот же код для создания и обновления документов.

Давайте вернемся к нашему примеру, где ведется запись количества просмотров для каждой страницы сайта. Без использования `upsert` можно было бы попытаться найти URL-адрес и увеличить количество просмотров или создать новый документ, если URL-адреса не существует. Если бы мы написали это как программу на языке JavaScript, это могло бы выглядеть примерно так:

```
// Проверяем, есть ли у нас запись для этой страницы;
blog = db.analytics.findOne({url : "/blog"})

// Если таковая имеется, добавляем ее к числу просмотров и сохраняем;
if (blog) {
  blog.pageviews++;
  db.analytics.save(blog);
}
// В противном случае мы создаем новый документ для этой страницы;
else {
  db.analytics.insertOne({url : "/blog", pageviews : 1})
}
```

Это означает, что мы отправляемся в базу данных и потом обратно, а также отправляем обновление или вставку каждый раз, когда кто-то посещает страницу. Если мы выполняем этот код в нескольких процессах, мы также сталкиваемся с состоянием гонки, при котором для данного URL-адреса может быть вставлено более одного документа.

Мы можем устранить состояние гонки и сократить объем кода, просто отправив `upsert` в базу данных (третий параметр для методов `updateOne` и `updateMany` – это документ параметров, который позволяет нам указать это):

```
> db.analytics.updateOne({"url" : "/blog"}, {"$inc" : {"pageviews" : 1}},
... {"upsert" : true})
```

Эта строка делает именно то, что делает предыдущий блок кода, за исключением того, что это быстрее! Новый документ создается с использованием документа критериев в качестве основы и применения к нему любых документов-модификаторов.

Например, если вы используете `upsert`, который совпадает с ключом и увеличивается до значения этого ключа, приращение будет применено к совпадению:

```
> db.users.updateOne({"rep" : 25}, {"$inc" : {"rep" : 3}}, {"upsert" : true})
WriteResult({
    "acknowledged" : true,
    "matchedCount" : 0,
    "modifiedCount" : 0,
    "upsertedId" : ObjectId("5a93b07aaea1cb8780a4cf72")
})
> db.users.findOne({"_id" : ObjectId("5727b2a7223502483c7f3acd")} )
{ "_id" : ObjectId("5727b2a7223502483c7f3acd"), "rep" : 28 }
```

`Upsert` создает новый документ, где `"rep"` равен 25, а затем увеличивает его на 3, давая нам документ, где `"rep"` равен 28. Если опция `upsert` не была

указана, `{"rep" : 25}` не будет совпадать ни с одним документом, поэтому ничего не произойдет.

Если мы снова запустим `upsert` (с критерием `{"rep" : 25}`), он создаст еще один новый документ. Это связано с тем, что критерий не соответствует единственному документу в коллекции. (Его `"rep"` равен 28.)

Иногда при создании документа необходимо задать поле, но не изменять его при последующих обновлениях. Для этого и используется оператор `"$setOnInsert"`. `"$setOnInsert"` – это оператор, который устанавливает значение поля только при вставке документа. Таким образом, мы могли бы сделать что-то вроде этого:

```
> db.users.updateOne({}, {"$setOnInsert" : {"createdAt" : new Date()}},
... {"upsert" : true})
{
    "acknowledged" : true,
    "matchedCount" : 0,
    "modifiedCount" : 0,
    "upsertedId" : ObjectId("5727b4ac223502483c7f3ace")
}
> db.users.findOne()
{
    "_id" : ObjectId("5727b4ac223502483c7f3ace"),
    "createdAt" : ISODate("2016-05-02T20:12:28.640Z")
}
```

Если мы запустим это обновление снова, оно будет соответствовать существующему документу, ничего не будет вставлено, поэтому поле `"createAt"` не будет изменено:

```
> db.users.updateOne({}, {"$setOnInsert" : {"createdAt" : new Date()}},
... {"upsert" : true})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }
> db.users.findOne()
{
    "_id" : ObjectId("5727b4ac223502483c7f3ace"),
    "createdAt" : ISODate("2016-05-02T20:12:28.640Z")
}
```

Обратите внимание, что, как правило, вам не нужно сохранять поле `"createAt"`, поскольку `ObjectId` содержат метку времени создания документа. Однако оператор `"$setOnInsert"` может быть полезен для создания отступов, инициализации счетчиков и для коллекций, которые не используют `ObjectId`.

Сохранение

`save` – это функция оболочки, которая позволяет вставлять документ, если он не существует, и обновлять его, если он уже есть. Она принимает один аргумент: документ. Если документ содержит ключ `"_id"`, функция использует upsert. В противном случае будет выполнена вставка. `save` – действительно удобная функция, поэтому программисты могут быстро изменять документы в оболочке:

```
> var x = db.testcol.findOne()
> x.num = 42
42
> db.testcol.save(x)
```

Без нее последняя строка была бы более громоздкой:

```
db.testcol.replaceOne({"_id" : x._id}, x)
```

Обновление нескольких документов

До сих пор в этой главе мы использовали метод `updateOne` для иллюстрации операций обновления. `updateOne` обновляет только первый найденный документ, который соответствует критериям фильтра. Если совпадающих документов больше, они останутся без изменений. Чтобы изменить все документы, соответствующие фильтру, используйте метод `updateMany`. `updateMany` следует той же семантике, что и `updateOne`, и принимает те же параметры. Основное различие заключается в количестве документов, которые можно изменить.

`updateMany` предоставляет мощный инструмент для выполнения миграций схемы или развертывания новых функций для определенных пользователей. Предположим, например, что мы хотим сделать подарок каждому пользователю, у которого день рождения в определенный день. Можно использовать метод `updateMany`, чтобы добавить «подарок» (`"gift"`) в их аккаунты. Например:

```
> db.users.insertMany([
... {birthday: "10/13/1978"},
... {birthday: "10/13/1978"},
... {birthday: "10/13/1978"}])
{
    "acknowledged" : true,
    "insertedIds" : [
        ObjectId("5727d6fc6855a935cb57a65b"),
        ObjectId("5727d6fc6855a935cb57a65c"),
        ObjectId("5727d6fc6855a935cb57a65d")
    ]
```

```
}
> db.users.updateMany({"birthday" : "10/13/1978"},
... {"$set" : {"gift" : "Happy Birthday!"}})
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
```

Вызов `updateMany` добавляет поле `"gift"` в каждый из трех документов, которые мы вставили в коллекцию *users* непосредственно перед этим.

Возврат обновленных документов

В некоторых случаях использования важно вернуть измененный документ. В более ранних версиях MongoDB в таких ситуациях предпочтение отдавалось методу `findAndModify`. Он удобен для манипулирования очередями и выполнения других операций, которые требуют атомарности в стиле get-and-set. Однако метод `findAndModify` подвержен пользовательским ошибкам, потому что это комплексный метод, сочетающий в себе функциональность трех различных типов операций: удаления, замены и обновления (включая upsert'ы).

В MongoDB версии 3.2 появились три новых метода коллекции для обеспечения функциональности `findAndModify`, но с семантикой, которую легче изучить и запомнить: `findOneAndDelete`, `findOneAndReplace` и `findOneAndUpdate`. Основное различие между этими методами и, например, методом `updateOne` заключается в том, что они позволяют атомарно получить значение модифицированного документа. В MongoDB версии 4.2 метод `findOneAndUpdate` был расширен, чтобы принимать конвейер агрегации для обновления. Конвейер может состоять из следующих этапов: `$addFields` и его псевдоним `$set`, `$project` и его псевдоним `$unset` и `$replaceRoot` и его псевдоним `$replaceWith`.

Предположим, у нас есть коллекция процессов, запущенных в определенном порядке. Каждый из них представлен документом следующей формы:

```
{
    "_id" : ObjectId(),
    "status" : "state",
    "priority" : N
}
```

`"status"` – это строка, которая может находиться в состоянии `"READY"`, `"RUNNING"` или `"DONE"`. Нам нужно найти задание с наивысшим приоритетом в состоянии `"READY"`, запустить функцию процесса, а затем обновить состояние до `"DONE"`. Мы могли бы попытаться запросить готовые процессы, рассортировать их по приоритету и обновить статус процесса с наивысшим приоритетом, чтобы пометить его как `"RUNNING"`. После того как мы обработали его, мы обновляем статус на `"DONE"`. Выглядит это примерно так:

```
var cursor = db.processes.find({"status" : "READY"});
ps = cursor.sort({"priority" : -1}).limit(1).next();
db.processes.updateOne({"_id" : ps._id}, {"$set" : {"status" : "RUNNING"}});
do_something(ps);
db.processes.updateOne({"_id" : ps._id}, {"$set" : {"status" : "DONE"}});
```

Данный алгоритм не очень хорош, потому что он зависит от состояния гонки. Предположим, у нас работает два потока. Если один поток (назовем его *A*) получил документ, а другой поток (назовем его *B*) получил тот же документ до того, как *A* обновил свой статус до `"RUNNING"`, оба потока будут работать в одном и том же процессе. Этого можно избежать, проверяя результат как часть запроса на обновление, но это становится сложным:

```
var cursor = db.processes.find({"status" : "READY"});
cursor.sort({"priority" : -1}).limit(1);
while ((ps = cursor.next()) != null) {
    var result = db.processes.updateOne({"_id" : ps._id, "status" : "READY"},
                                {"$set" : {"status" : "RUNNING"}});

    if (result.modifiedCount === 1) {
        do_something(ps);
        db.processes.updateOne({"_id" : ps._id}, {"$set" : {"status" : "DONE"}});
        break;
    }
    cursor = db.processes.find({"status" : "READY"});
    cursor.sort({"priority" : -1}).limit(1);
}
```

Кроме того, в зависимости от синхронизации один поток может выполнить всю работу, в то время как другой поток впустую отслеживает это. Поток *A* всегда может захватить процесс, а затем, когда *B* попытается заполучить тот же процесс, это окончится неудачей, и *A* выполнит всю работу.

Подобные ситуации идеально подходят для метода `findOneAndUpdate`. Он может вернуть элемент и обновить его за одну операцию. В этом случае это выглядит следующим образом:

```
> db.processes.findOneAndUpdate({"status" : "READY"},
... {"$set" : {"status" : "RUNNING"}},
... {"sort" : {"priority" : -1}})
{
    "_id" : ObjectId("4b3e7a18005cab32be6291f7"),
    "priority" : 1,
    "status" : "READY"
}
```

Обратите внимание, что в возвращаемом документе мы по-прежнему видим слово "READY", поскольку метод findOneAndUpdate по умолчанию возвращает состояние документа до его изменения. Он вернет обновленный документ, если мы установим для поля "returnNewDocument" в документе параметров значение true. Документ параметров передается в качестве третьего параметра методу findOneAndUpdate:

```
> db.processes.findOneAndUpdate({"status" : "READY"},
... {"$set" : {"status" : "RUNNING"}},
... {"sort" : {"priority" : -1},
...  "returnNewDocument": true})
{
    "_id" : ObjectId("4b3e7a18005cab32be6291f7"),
    "priority" : 1,
    "status" : "RUNNING"
}
```

Таким образом, программа будет выглядеть так:

```
ps = db.processes.findOneAndUpdate({"status" : "READY"},
                                   {"$set" : {"status" : "RUNNING"}},
                                   {"sort" : {"priority" : -1},
                                    "returnNewDocument": true})
do_something(ps)
db.process.updateOne({"_id" : ps._id}, {"$set" : {"status" : "DONE"}})
```

В дополнение к этому есть два других метода, о которых вы должны знать.

findOneAndReplace принимает те же параметры и возвращает документ, соответствующий фильтру, до или после замены, в зависимости от значения returnNewDocument. Метод findOneAndDelete работает аналогичным образом, за исключением того, что он не принимает документ обновления в качестве параметра и имеет подмножество параметров двух других методов. Метод findOneAndDelete возвращает удаленный документ.

Глава 4

Выполнение запросов

В этой главе подробно рассматривается, как выполнять запросы. Основные области:

- вы можете запрашивать диапазоны, установить включение, неравенства и многое другое, используя условные операторы, начинающиеся с символа `$`;
- запросы возвращают курсор базы данных, который лениво возвращает пакеты документов по мере необходимости;
- существует множество метаопераций, которые можно выполнять с курсором, включая пропуск определенного количества результатов, ограничение количества возвращаемых результатов и сортировку результатов.

Знакомство с методом find

Метод `find` используется в MongoDB для выполнения запросов. При выполнении запроса возвращается подмножество документов в коллекции, при этом документов может не быть вообще либо это может быть вся коллекция. Какие документы возвращаются, определяется первым аргументом метода, который представляет собой документ, определяющий критерии запроса.

Пустой документ запроса (т. е. {}) соответствует всему, что есть в коллекции. Если методу `find` не предоставлен документ запроса, по умолчанию используется {}. Например, строка

```
> db.c.find()
```

соответствует каждому документу в коллекции *c* (и возвращает эти документы в пакетном режиме).

Когда мы начинаем добавлять пары типа «ключ/значение» в документ запроса, мы начинаем ограничивать наш поиск. Для большинства типов это работает просто: числа соответствуют числам, логические значения соответствуют логическим значениям, а строки соответствуют строкам.

Запросить простой тип так же просто, как указать значение, которое вы ищете. Например, чтобы найти все документы, где значение `"age"` равно 27, можно добавить эту пару типа «ключ/значение» в документ запроса:

```
> db.users.find({"age" : 27})
```

Если у нас есть строка, которую мы хотим сопоставить, например ключ `«username»` и значение `"joe"`, мы используем эту пару:

```
> db.users.find({"username" : "joe"})
```

Можно объединить несколько условий, добавив больше пар типа «ключ/значение» в документ запроса, который интерпретируется как «*условие*1 И *условие*2 И... И *условие*N». Например, чтобы получить всех пользователей, которым 27 лет, с именем «joe», можно выполнить следующий запрос:

```
> db.users.find({"username" : "joe", "age" : 27})
```

Указываем, какие ключи нужно вернуть

Иногда вам не нужны все пары типа «ключ/значение» в возвращаемом документе. В этом случае можно передать второй аргумент методу `find` (или `findOne`), указывая нужные вам ключи. Это уменьшает как объем передаваемых данных, так и время и память, используемые для декодирования документов на стороне клиента.

Например, если у вас есть коллекция пользователей и вас интересуют только ключи `"username"` и `"email"`, можно вернуть лишь эти ключи с помощью следующего запроса:

```
> db.users.find({}, {"username" : 1, "email" : 1})
{
    "_id" : ObjectId("4ba0f0dfd22aa494fd523620"),
    "username" : "joe",
    "email" : "joe@example.com"
}
```

Как видно из предыдущего вывода, ключ `"_id"` возвращается по умолчанию, даже если он не запрашивается специально.

Вы также можете использовать этот второй параметр, чтобы исключить определенные пары типа «ключ/значение» из результатов запроса. Например, у вас могут быть документы с различными ключами, и единственное, что вы знаете, – это то, что вы ни за что не хотите возвращать ключ `"fatal_weakness"`:

```
> db.users.find({}, {"fatal_weakness" : 0})
```

Приведенный ниже код также может предотвратить возвращение `"_id"`:

```
> db.users.find({}, {"username" : 1, "_id" : 0})
{
    "username" : "joe",
}
```

Ограничения

Есть некоторые ограничения на запросы. Значение документа запроса должно быть константой для базы данных. (Это может быть обычная переменная в вашем собственном коде.) То есть она не может ссылаться на значение другого ключа в документе. Например, если бы мы вели учет и у нас были бы ключи `"in_stock"` и `"num_sold"`, мы не могли бы сравнивать их значения с помощью следующего запроса:

```
> db.stock.find({"in_stock" : "this.num_sold"}) //Это не сработает;
```

Есть способы сделать это (*см.* раздел «Операторы $where»), но, как правило, вы будете получать лучшую производительность, слегка реструктурировав документ, поэтому «обычного» запроса будет достаточно. В этом примере мы могли бы использовать ключи `"initial_stock"` и `"in_stock"`. Затем каждый раз, когда кто-то покупает товар, мы уменьшаем значение ключа `"in_stock"` на единицу. Наконец, мы можем сделать простой запрос, чтобы проверить, каких товаров нет в наличии:

```
> db.stock.find({"in_stock" : 0})
```

Критерии запроса

Запросы могут выходить за рамки точного соответствия, описанного в предыдущем разделе; они могут соответствовать более сложным критериям, таким как диапазоны, операторы OR и отрицание.

Условные операторы

`"$lt"`, `"$lte"`, `"$gt"` и `"$gte"` – все это операторы сравнения, соответствующие <, <=, > и >= соответственно. Их можно комбинировать для поиска диапазона значений. Например, чтобы найти пользователей в возрасте от 18 до 30 лет, можно выполнить следующий запрос:

```
> db.users.find({"age": {"$gte": 18, "$lte": 30}})
```

При этом будут найдены все документы, где поле `"age"` было больше или равно 18 и меньше или равно 30.

Эти типы запросов диапазона часто полезны, когда речь идет о датах. Например, чтобы найти людей, которые зарегистрировались до 1 января 2007 года, можно выполнить следующий запрос:

```
> start = new Date("01/01/2007")
> db.users.find({"registered" : {"$lt" : start}})
```

В зависимости от того, как вы создаете и храните даты, точное совпадение может быть менее полезным, поскольку даты хранятся с точностью до миллисекунды. Часто вам требуется целый день, неделя или месяц, что делает необходимым запрос по диапазону.

Чтобы запросить документы, в которых значение ключа не равно определенному значению, нужно использовать другой условный оператор `"$ne"`, который означает «не равно». Если вы хотите найти всех пользователей, у которых нет имени «joe», можно запросить их таким образом:

```
> db.users.find({"username" : {"$ne" : "joe"}})
```

Оператор «`$ne`» можно использовать с любым типом.

Запросы с оператором OR

В MongoDB есть два способа выполнить запрос оператором OR. Оператор `"$in"` может использоваться для запроса различных значений для одного ключа. Оператор `"$or"` является более общим; его можно использовать для запроса любого из заданных значений по нескольким ключам.

Если у вас имеется более одного возможного значения для одного ключа, используйте массив критериев с `"$in"`. Например, предположим, что мы проводим розыгрыш и номера выигрышных билетов – 75, 542 и 390. Чтобы найти все три этих документа, можно создать следующий запрос:

```
> db.raffle.find({"ticket_no": {"$in": [725, 542, 390]}})
```

Оператор `"$in"` очень гибкий и позволяет указывать критерии различных типов, а также значения. Например, если мы постепенно переносим нашу схему, чтобы использовать имена пользователей вместо идентификаторов пользователей, можно запросить любой из них:

```
> db.users.find({"user_id": {"$in": [12345, "joe"]}})
```

Этот запрос сопоставляет документы, где `"user_id"` равен 12 345, и документы, где `"user_id"` имеет значение `"joe"`.

Если оператору `"$in"` задан массив с одним значением, он ведет себя так же, как и прямое сопоставление со значением. Например, `{ticket_no: {$in: [725]}}` совпадает с теми же документами, что и `{ticket_no: 725}`.

Противоположностью оператору `"$in"` является оператор `"$nin"`, который возвращает документы, которые не соответствуют ни одному из критериев в массиве. Если мы хотим вернуть всех тех, кто ничего не выиграл во время лотереи, это можно сделать с помощью следующего запроса:

```
> db.raffle.find({"ticket_no": {"$ nin": [725, 542, 390]}})
```

Этот запрос возвращает всех, у кого не было билетов с этими номерами.

Оператор `"$in"` дает вам запрос OR для одного ключа, но что, если нам нужно найти документы, где `"ticket_no"` равен 725 или значение `"winner"` равно `true`? Для этого типа запроса нам понадобится использовать условный оператор `"$or"`. `"$or"` принимает массив возможных критериев. В случае с лотереей использование оператора `"$or"` будет выглядеть так:

```
> db.raffle.find({"$or": [{"ticket_no": 725}, {"winner": true}]})
```

`"$or"` может содержать другие условия. Если, например, мы хотим сопоставить любое из трех значений `"ticket_no"` или ключ `"winner"`, можно использовать это:

```
> db.raffle.find({"$or" : [{"ticket_no" : {"$in" : [725, 542, 390]}},
... {"winner" : true}]})
```

С помощью обычного запроса типа AND вы хотите максимально сузить результаты, используя как можно меньше аргументов. Запросы типа OR действуют наоборот: они наиболее эффективны, если первые аргументы соответствуют как можно большему количеству документов.

Хотя оператор `"$or"` будет работать, всегда, когда это возможно, используйте `"$in"`, поскольку оптимизатор запросов обрабатывает его более эффективно.

$not

`"$not"` является метаусловным оператором: его можно применять поверх любых других критериев. В качестве примера давайте рассмотрим оператор модуля `"$mod"`. `"$mod"` запрашивает ключи, значения которых при делении на первое заданное значение имеют остаток от второго значения:

```
> db.users.find({"id_num": {"$mod": [5, 1]}})
```

Предыдущий запрос возвращает пользователей с номерами 1, 6, 11, 16 и т. д. Если вместо этого мы хотим вернуть пользователей с номерами 2, 3, 4, 5, 7, 8, 9, 10, 12 и т. д., можно использовать оператор `"$not"`:

```
> db.users.find({"id_num": {"$not": {"$mod": [5, 1]}}})
```

`"$not"` может быть особенно полезен в сочетании с регулярными выражениями для поиска всех документов, которые не соответствуют заданному шаблону (использование регулярных выражений описано в разделе «Регулярные выражения»).

Запросы для определенных типов

Как описано в главе 2, MongoDB имеет широкий спектр типов, которые можно использовать в документе. Некоторые из этих типов ведут себя особым образом при выполнении запросов.

null

Тип `null` ведет себя немного странно. Он соответствует самому себе, поэтому если у нас есть коллекция со следующими документами:

```
> db.c.find()
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{ "_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
```

мы можем запросить документы, у которых ключ `"y"` равен нулю, ожидаемым образом:

```
> db.c.find({"y" : null})
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
```

Однако `null` также соответствует значению «не существует». Таким образом, запрос ключа со значением `null` вернет все документы, в которых этот ключ отсутствует:

```
> db.c.find({«z» : null})
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{ "_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
```

Если мы хотим найти только ключи со значением `null`, можно проверить, что ключ имеет значение `null` и что он существует, используя условный оператор `"$exists"`:

```
> db.c.find({"z" : {"$eq" : null, "$exists" : true}})
```

Регулярные выражения

Оператор `"$regex"` предоставляет возможности регулярных выражений для сопоставления с образцом в запросах. Регулярные выражения полезны для гибкого сопоставления строк. Например, если нам нужно найти всех пользователей с именем «Joe» или «joe», можно использовать регулярное выражение для нечувствительного к регистру сопоставления:

```
> db.users.find( {"name" : {"$regex" : /joe/i}})
```

Флаги регулярных выражений (например, `i`) допустимы, но не обязательны. Если мы хотим сопоставить не только различные варианты написания имени «joe», но и «joey», можно продолжить улучшать наше регулярное выражение:

```
> db.users.find({"name": /joey?/i})
```

MongoDB использует библиотеку PCRE (Perl Compatible Regular Expressions), которая реализует работу регулярных выражений в стиле Perl; любой синтаксис регулярных выражений, разрешенный PCRE, разрешен в MongoDB. Рекомендуется проверить синтаксис в оболочке JavaScript, прежде чем использовать его в запросе, чтобы убедиться, что он совпадает с тем, с чем, по вашему мнению, он должен совпадать.

MongoDB может использовать индекс для запросов, когда речь идет о префиксных регулярных выражениях (например, `/^joey/`). Индексы *нельзя* использовать для нечувствительного к регистру поиска (`/^joey/i`). Регулярное выражение является «префиксным выражением», когда оно начинается со знака каретки (^) или наклонной черты (`\A`). Если регулярное выражение использует регистрозависимый запрос, тогда, если для поля существует индекс, сопоставления могут проводиться со значениями в индексе. Если оно также является префиксным выражением, поиск может быть ограничен значениями в диапазоне, созданном этим префиксом из индекса.

Регулярные выражения также могут совпадать друг с другом. Мало кто вставляет регулярные выражения в базу данных, но если вы вставите одно из них, можно сопоставить его с самим собой:

```
> db.foo.insertOne({"bar" : /baz/})
> db.foo.find({"bar" : /baz/})
{
    "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
    "bar" : /baz/
}
```

Запросы элементов массива

Запросы элементов массива разработаны таким образом, чтобы вести себя так же, как запросы скаляров. Например, если массив представляет собой список фруктов:

```
> db.food.insertOne({"fruit": ["apple", "banana", "peach"]})
```

приведенный ниже запрос будет успешно совпадать с документом:

```
> db.food.find({"fruit" : "banana"})
```

Мы можем запросить его практически так же, как если бы у нас был документ, который выглядел бы как (недопустимый) документ `{"fruit" : "apple", "fruit": "banana", "fruit": "peach"}`.

"$all"

Если вам нужно сопоставить массивы по нескольким элементам, можно использовать оператор `"$all"`. Он позволяет сопоставить список элементов. Например, предположим, что мы создаем коллекцию из трех элементов:

```
> db.food.insertOne({"_ id" : 1, "fruit": ["apple", "banana", "peach"]})
> db.food.insertOne({"_ id" : 2, "fruit": ["apple", "kumquat", "orange"]})
> db.food.insertOne({"_ id" : 3, "fruit": ["cherry", "banana", "apple"]})
```

Затем мы можем найти все документы с элементами `"apple"` и `"banana"`, выполнив запрос с помощью оператора `"$all"`:

```
> db.food.find({fruit : {$ all : ["apple", "banana"]}})
{"_id" : 1, "fruit" : ["apple", "banana", "peach"]}
{"_id" : 3, "fruit" : ["cherry", "banana", "apple"]}
```

Порядок не имеет значения. Обратите внимание, что `"banana"` стоит перед `"apple"` во втором результате. Использование одноэлементного массива с `"$all"` эквивалентно неиспользованию `"$all"`. Например, `{fruit : {$ all: ['apple']}` будет соответствовать тем же документам, что и `{fruit : 'apple'}`.

Вы также можете выполнить запрос по точному совпадению, используя весь массив. Однако точное совпадение не будет соответствовать документу, если какие-либо элементы отсутствуют или являются лишними. Например, этот запрос будет соответствовать первому из трех наших документов:

```
> db.food.find({"fruit" : ["apple", "banana", "peach"]})
```

А этот не будет:

```
> db.food.find({"fruit" : ["apple", "banana"]})
```

Равно как и этот:

```
> db.food.find({"fruit" : ["banana", "apple", "peach"]})
```

Если вы хотите запросить определенный элемент массива, можно указать индекс, используя синтаксис *ключ. индекс*:

```
> db.food.find({"fruit.2" : "peach"})
```

В массивах индексация всегда начинается с 0, поэтому третий элемент массива сопоставляется со строкой `"peach"`.

"$size"

Полезным условным оператором для запроса массивов является оператор `"$size"`, который позволяет запрашивать массивы заданного размера. Пример:

```
> db.food.find({"fruit" : {"$ size": 3}})
```

Один из распространенных запросов – получение диапазона размеров. Оператор `"$size"` нельзя объединить с другим условным оператором (в данном примере с `"$gt"`), но этот запрос можно выполнить, добавив в документ ключ `"size"`. Затем каждый раз, когда вы добавляете элемент в массив, увеличивайте значение `"size"`. Если исходное обновление выглядело так:

```
> db.food.update(criteria, {"$push" : {"fruit" : "strawberry"}})
```

можно просто поменять его на это:

```
> db.food.update(criteria,
... {"$push" : {"fruit" : "strawberry"}, "$inc" : {"size" : 1}})
```

Прирост идет очень быстро, поэтому любое снижение производительности незначительно. Хранение документов, подобных этому, позволяет вам выполнять такие запросы:

```
> db.food.find({"size": {"$ gt": 3}})
```

К сожалению, данный метод не работает с оператором `"$addToSet"`.

"$slice"

Как упоминалось ранее в этой главе, второй необязательный аргумент для метода `find` определяет ключи, которые должны быть возвращены. Можно использовать специальный оператор `"$slice"`, чтобы вернуть подмножество элементов для ключа массива.

Например, предположим, что у нас был документ поста в блоге, и нам нужно вернуть первые 10 комментариев:

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : 10}})
```

Также, если бы нам были нужны последние 10 комментариев, мы могли бы использовать –10:

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : -10}})
```

Оператор `"$slice"` еще может возвращать страницы посреди результатов, принимая смещение и число возвращаемых элементов:

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : [23, 10]}})
```

Первые 23 элемента будут пропущены, и вернутся элементы с 24-го по 33-й. Если бы в массиве было меньше 33 элементов, он вернул бы как можно больше элементов.

Если не указано иное, при использовании оператора `"$slice"` возвращаются все ключи в документе. Это отличается от других спецификаторов ключей, которые подавляют возврат неупомянутых ключей. Например, если бы у нас был документ поста блога следующего вида:

```
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "joe",
            "email" : "joe@example.com",
            "content" : "nice post."
        },
        {
            "name" : "bob",
            "email" : "bob@example.com",
            "content" : "good post."
        }
    ]
}
```

и мы бы использовали оператор `"$slice"`, чтобы получить последний комментарий, вот что бы вышло:

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : -1}})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "bob",
            "email" : "bob@example.com",
            "content" : "good post."
```

```
        }
    ]
}
```

И `"title"`, и `"content"` по-прежнему возвращаются, хотя они не были явно включены в спецификатор ключа.

Возврат совпадающего элемента массива

Оператор `"$slice"` полезен, когда вы знаете индекс элемента, но иногда вам нужно, чтобы любой элемент массива совпадал с вашими критериями. Совпадающий элемент можно вернуть с помощью оператора `$`. Учитывая предыдущий пример с блогом, можно получить комментарий Боба с помощью этого запроса:

```
> db.blog.posts.find({"comments.name" : "bob"}, {"comments.$" : 1})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "comments" : [
        {
            "name" : "bob",
            "email" : "bob@example.com",
            "content" : "good post."
        }
    ]
}
```

Обратите внимание на то, что для каждого документа возвращается только первое совпадение: если бы Боб оставил несколько комментариев к этому посту, был бы возвращен только первый комментарий в массиве `"comments"`.

Взаимодействия запросов по диапазону и запросов к массиву

Скаляры (элементы, не являющиеся элементами массива) в документах должны совпадать с каждым предложением критериев запроса.

Например, если вы запросили `{"x": {"$gt": 10, "$lt": 20}}`, `"x"` должно быть больше 10 и меньше 20. Однако если поле документа `"x"` является массивом, документ совпадает, если существует элемент `"x"`, который совпадает с каждой частью критерия, но *каждое предложение запроса может совпадать с другим элементом массива*.

Лучший способ понять это поведение – посмотреть пример. Предположим, у нас есть следующие документы:

```
{"x" : 5}
{"x" : 15}
```

```
{"x" : 25}
{"x" : [5, 25]}
```

Если бы мы хотели найти все документы, где `"x"` находится между 10 и 20, то могли бы наивно структурировать запрос в виде `db.test.find({"x" : {"$gt" : 10, "$lt" : 20}})`, ожидая получить один документ: `{"x" : 15}`. Однако, выполнив этот запрос, мы получим два документа:

```
> db.test.find({"x" : {"$gt" : 10, "$lt" : 20}})
{"x" : 15}
{"x" : [5, 25]}
```

Ни 5, ни 25 не находятся между 10 и 20, но документ возвращается, потому что 25 совпадает с первым предложением (это больше 10), а 5 совпадает со вторым (это меньше 20).

Таким образом запросы по диапазону к массивам становятся практически бесполезными: диапазон будет совпадать с любым многоэлементным массивом. Есть несколько способов получить ожидаемое поведение.

Во-первых, можно использовать оператор `"$elemMatch"`, чтобы заставить MongoDB сравнивать оба предложения с одним элементом массива. Однако подвох заключается в том, что этот оператор не будет соответствовать элементам без массивов:

```
> db.test.find({"x" : {"$elemMatch" : {"$gt" : 10, "$lt" : 20}}})
> // Никаких результатов;
```

Документ `{"x" : 15}` больше не совпадает с запросом, поскольку поле `"x"` не является массивом. Тем не менее у вас должна быть веская причина для смешивания массивов и скалярных значений в поле. Во многих случаях использования смешивания не требуется. Для них оператор `"$elemMatch"` предлагает хорошее решение для запросов по диапазону к элементам массива.

Если у вас есть индекс по полю, по которому вы выполняете запрос (*см.* главу 5), можно использовать `min` и `max`, чтобы ограничить диапазон индекса, пройденного запросом, значениями `"$gt"` и `"$lt"`:

```
> db.test.find({"x" : {"$gt": 10, "$lt" : 20}}).min ({"x" : 10}).max ({"x" : 20} )
{"x" : 15}
```

Теперь мы проходим индекс только с 10 до 20, пропуская записи 5 и 25. `min` и `max` можно использовать только тогда, когда у вас есть индекс по полю, по которому вы выполняете запрос, и вы должны передать `min` и `max` все поля индекса.

Использование `min` и `max` при запросе по диапазону для документов, которые могут включать в себя массивы, – как правило, неплохая идея. Индексные границы для запроса `"$gt"/"$lt"` по массиву неэффективны. В ос-

новном он принимает любое значение, поэтому будет искать все записи индекса, а не только те, которые находятся в диапазоне.

Запросы по вложенным документам

Существует два способа запроса по вложенному документу: запросить весь документ или запросить его отдельные пары типа «ключ/значение».

Запрос встраиваемого документа целиком работает так же, как и в случае с обычным запросом. Например, если у нас есть документ, который выглядит следующим образом:

```
{
    "name" : {
        "first" : "Joe",
        "last" : "Schmoe"
    },
    "age" : 45
}
```

мы можем выполнить запрос, чтобы найти человека по имени Джо Шмо:

```
> db.people.find ({"name" : {"first" : "Joe", "last" : "Schmoe"}})
```

Однако запрос всего вложенного документа должен точно совпадать с ним. Если Джо решит добавить поле для указания среднего имени, этот запрос больше не будет работать; он не совпадает с документом! Данный тип запроса также чувствителен к порядку: `{"last" : "Schmoe", "first" : "Joe"}` не будет совпадением.

Если это возможно, обычно рекомендуется запросить только определенный ключ или ключи вложенного документа. Затем, если ваша схема изменится, все ваши запросы не будут внезапно прерываться из-за того, что они больше не являются точными совпадениями. Можно запросить вложенные ключи, используя точечную нотацию:

```
> db.people.find({"name.first" : "Joe", "name.last" : "Schmoe"})
```

Теперь если Джо добавит дополнительные ключи, этот запрос все равно будет соответствовать его имени и фамилии.

Эта точечная нотация является основным отличием между документами запроса и документами других типов. Документы запроса могут содержать точки, которые означают «добраться до вложенного документа». Точечная нотация также является причиной того, что документы, которые нужно вставить, не могут содержать символ «.». Часто люди сталкиваются с этим ограничением при попытке сохранить URL-адреса в качестве ключей. Один из способов обойти эту проблему – всегда выполнять глобаль-

ную замену перед вставкой или после извлечения, заменяя символ, который не разрешен в URL-адресах, символом точки.

Сопоставление встроенных документов может стать немного сложнее, поскольку структура документа усложняется. Например, предположим, что мы сохраняем посты в блоге и хотим найти комментарии Джо, которые были оценены как минимум в 5 баллов. Можно смоделировать этот пост следующим образом:

```
> db.blog.find()
{
    "content" : "...",
    "comments" : [
        {
            "author" : "joe",
            "score" : 3,
            "comment" : "nice post"
        },
        {
            "author" : "mary",
            "score" : 6,
            "comment" : "terrible post"
        }
    ]
}
```

Теперь мы не можем делать запросы, используя `db.blog.find({"comments" : {"author" : "joe", "Score" : {"$ gte" : 5}}})`. Соответствия встроенного документа должны совпадать со всем документом, а это не совпадает с ключом `"comment".db.blog.find({"comments.author" : "joe", "comments.score": {"$gte" : 5}})` также не сработает, поскольку критерий автора может совпадать с другим комментарием, а не с критерием оценки. То есть в результате запроса мы бы получили документ, показанный выше: он соответствовал бы `"author" : "joe"` в первом комментарии и `"score" : 6` во втором комментарии.

Чтобы правильно сгруппировать критерии без указания каждого ключа, используйте оператор `"$elemMatch"`. Этот условный оператор со смутным названием позволяет вам частично указать критерии для совпадения с одним вложенным документом в массиве. Правильный запрос выглядит так:

```
> db.blog.find({"comments" : {"$elemMatch" :
... {"author" : "joe", "score" : {"$gte" : 5}}}})
```

`"$elemMatch"` позволяет вам «группировать» свои критерии. Таким образом, он необходим только тогда, когда у вас есть несколько ключей, которым вам нужно совпадение во вложенном документе.

Операторы $where

Пары типа «ключ/значение» являются довольно выразительным способом запроса, но есть запросы, которые они не могут представлять. Для запросов, которые не могут быть выполнены каким-либо другим способом, существуют операторы `$where`, позволяющие выполнять произвольный код на языке JavaScript как часть вашего запроса. Это позволяет вам делать (почти) все, что угодно, в рамках запроса. В целях безопасности использование операторов `$where` должно быть строго ограничено или исключено. Конечным пользователям никогда не следует разрешать использовать произвольные операторы `$where`.

Наиболее распространенным случаем использования оператора `$where` является сравнение значений двух ключей в документе. Например, предположим, у нас есть документы, которые выглядят так:

```
> db.foo.insertOne({"apple" : 1, "banana" : 6, "peach" : 3})
> db.foo.insertOne({"apple" : 8, "spinach" : 4, "watermelon" : 4})
```

Мы хотели бы вернуть документы, где любые два поля одинаковы. Например, во втором документе `"spinach"` и `"watermelon"` имеют одно и то же значение, поэтому мы бы хотели, чтобы этот документ был возвращен. Маловероятно, что в MongoDB когда-либо появится условный оператор для этого, поэтому можно использовать оператор `$where`:

```
> db.foo.find({"$where" : function () {
... for (var current in this) {
...     for (var other in this) {
...         if (current != other && this[current] == this[other]) {
...             return true;
...         }
...     }
... }
... return false;
... }});
```

Если функция возвращает значение `true`, документ будет частью набора результатов; если она вернет значение `false`, этого не произойдет.

Запросы с помощью оператора `$where` не следует использовать без крайней необходимости: они намного медленнее, по сравнению с обычными запросами. Каждый документ необходимо преобразовать из формата BSON в объект JavaScript, а затем прибегнуть к оператору `$where`. Индексы здесь также нельзя применять. Следовательно, вы должны использовать оператор `$where` только тогда, когда нет другого способа выполнить запрос. Можно сократить снижение производительности, используя другие фильтры запросов в сочетании с `$where`. Если это возможно, индекс будет

использоваться для фильтрации на основе операторов, отличных от `$where`; оператор `$where` будет применяться только для точной настройки результатов. В MongoDB версии 3.6 был добавлен оператор `$expr`, позволяющий использовать выражения агрегации на языке запросов MongoDB. Он быстрее, чем `$where`, поскольку не выполняет код на JavaScript и рекомендуется в качестве замены этого оператора, где это возможно.

Еще одним способом выполнения сложных запросов является использование одного из инструментов агрегации, которые описаны в главе 7.

Курсоры

База данных возвращает результаты из метода `find`, используя *курсор*. Реализация курсоров на стороне клиента, как правило, позволяет в значительной степени контролировать конечный результат запроса. Вы можете ограничить количество результатов, пропустить некоторое количество результатов, отсортировать результаты по любой комбинации клавиш в любом направлении и выполнить ряд других мощных операций.

Чтобы создать курсор с помощью оболочки, поместите документы в коллекцию, выполните запрос к ним и присвойте результаты локальной переменной (переменные, определенные с помощью слова `"var"`, являются локальными). Здесь мы создаем очень простую коллекцию и выполняем запрос, сохраняя результаты в переменной `cursor`:

```
> for(i=0; i<100; i++) {
...     db.collection.insertOne({x : i});
... }
> var cursor = db.collection.find();
```

Преимущество этих действий состоит в том, что вы можете смотреть на один результат за раз. Если вы сохраните результаты в глобальной переменной или вообще не будете ее использовать, оболочка MongoDB будет автоматически перебирать и отображать первую пару документов. Это то, что мы видели до этого момента, и часто именно такое поведение вам и нужно, чтобы увидеть, что находится в коллекции, но не заниматься программированием с помощью оболочки.

Чтобы перебрать результаты, можно использовать метод `next`. Можно воспользоваться методом `hasNext`, чтобы проверить, есть ли еще результат. Типичная итерация результата выглядит следующим образом:

```
> while (cursor.hasNext()) {
...     obj = cursor.next();
...     // do stuff
... }
```

Метод `cursor.hasNext()` проверяет, существует ли следующий результат, а метод `cursor.next()` выбирает его.

Класс `cursor` также реализует интерфейс итератора JavaScript, поэтому вы можете использовать его в цикле `forEach`:

```
> var cursor = db.people.find();
> cursor.forEach(function(x) {
...     print(x.name);
... });
adam
matt
zak
```

Когда вы вызываете метод `find`, оболочка не запрашивает базу данных тотчас же. Она ждет, пока вы не начнете запрашивать результаты для отправки запроса, что позволит вам связать дополнительные параметры в запросе до его выполнения. Почти каждый метод в объекте `cursor` возвращает сам курсор, поэтому можно связывать опции в любом порядке. Например, все приведенное ниже равнозначно:

```
> var cursor = db.foo.find().sort({"x" : 1}).limit(1).skip(10);
> var cursor = db.foo.find().limit(1).sort({"x" : 1}).skip(10);
> var cursor = db.foo.find().skip(10).limit(1).sort({"x" : 1});
```

На данный момент запрос еще не выполнен. Все эти функции просто создают его. Теперь предположим, что мы вызываем следующий метод:

```
> cursor.hasNext()
```

В этот момент запрос будет отправлен на сервер. Оболочка извлекает первые 100 результатов или первые 4 МБ результатов (в зависимости от того, что меньше), чтобы при последующих вызовах `next` или `hasNext` не нужно было совершать путешествие к серверу. После того как клиент выполнит первый набор результатов, оболочка снова свяжется с базой данных и запросит дополнительные результаты с помощью запроса `getMore`. Запросы `getMore` в основном содержат идентификатор курсора и спрашивают у базы данных, есть ли еще какие-либо результаты, возвращая следующий пакет, если таковые имеются. Этот процесс продолжается до тех пор, пока курсор не будет исчерпан и не будут возвращены все результаты.

Ограничения, пропуск и сортировка

Наиболее распространенные параметры запроса – ограничение количества возвращаемых результатов, пропуск количества результатов и сортировка. Все эти параметры должны быть добавлены до отправки запроса в базу данных.

Чтобы установить предел, включите в свой вызов метода `find` функцию `limit`. Например, чтобы вернуть только три результата, используйте это:

```
> db.c.find().limit(3)
```

Если в коллекции меньше трех документов, соответствующих вашему запросу, будет возвращено только количество совпадающих документов; функция `limit` устанавливает верхний предел, а не нижний.

Функция `skip` работает похожим образом:

```
> db.c.find().skip(3)
```

Первые три совпадающих документа будут пропущены, а оставшиеся совпадения будут возвращены. Если в вашей коллекции меньше трех документов, ни один из документов возвращен не будет.

Функция `sort` принимает объект: набор пар типа «ключ/значение», где ключи – это имена ключей, а значения – направления сортировки. Направление сортировки может быть 1 (по возрастанию) или –1 (по убыванию). Если указано несколько ключей, результаты будут отсортированы в указанном порядке. Например, чтобы отсортировать результаты по `"username"` в порядке возрастания и по `"age"` в порядке убывания, мы делаем следующее:

```
> db.c.find().sort({username : 1, age : -1})
```

Эти три метода можно сочетать, что часто удобно для нумерации страниц. Например, предположим, что вы работаете в интернет-магазине и кто-то ищет *mp3*. Если вы хотите, чтобы 50 результатов на странице были отсортированы по цене от высокой к низкой, можно выполнить следующий запрос:

```
> db.stock.find({"desc" : "mp3"}).limit(50).sort({"price" : -1})
```

Если человек нажимает кнопку «Следующая страница», чтобы увидеть дополнительные результаты, можно просто добавить к запросу функцию `skip`, в результате чего первые 50 совпадений будут пропущены (которые пользователь уже видел на странице 1):

```
> db.stock.find({"desc" : "mp3"}).limit(50).skip(50).sort({"price" : -1})
```

Однако большие пропуски не очень производительны; предложения по поводу того, как избежать их, приводятся в следующем разделе.

Порядок сравнения

В MongoDB существует иерархия относительно того, как типы сравниваются. Иногда у вас будет один ключ с несколькими типами: например,

целые числа и логические типы данных или строки и нули. Если вы выполняете сортировку по ключу со смесью типов, существует предопределенный порядок, в котором они будут отсортированы. Вот как выглядит этот порядок от наименьшего значения к наибольшему:

1) минимальное значение;
2) ноль;
3) числа (целые, длинные, двойные, десятичные);
4) струны;
5) объект/документ;
6) массив;
7) двоичные данные;
8) идентификатор объекта;
9) логический тип данных;
10) дата;
11) временная отметка;
12) регулярное выражение;
13) максимальное значение.

Избегайте больших пропусков

Использование функции `skip` для небольшого количества документов – это прекрасно. Но при большом количестве результатов она может работать медленно, поскольку ей нужно найти и затем отбросить все пропущенные результаты. Большинство СУБД хранят больше метаданных в индексе, чтобы было легче иметь дело с пропусками, но в MongoDB такой функционал пока отсутствует, поэтому больших пропусков следует избегать. Часто можно рассчитать результаты следующего запроса на основе предыдущего.

Разбивка результатов на страницы без пропусков

Самый простой способ сделать нумерацию страниц – это вернуть первую страницу результатов, используя функцию `limit`, а затем вернуть каждую последующую страницу в виде смещения от начала:

```
> // Не используйте этот вариант: в случае с большими пропусками он будет
работать медленно;
> var page1 = db.foo.find(criteria).limit(100)
> var page2 = db.foo.find(criteria).skip(100).limit(100)
> var page3 = db.foo.find(criteria).skip(200).limit(100)
...
```

Однако в зависимости от вашего запроса обычно можно найти способ разбивки на страницы без пропусков. Например, предположим, что мы хотим отображать документы в порядке убывания на основе `"date"`. Мы можем получить первую страницу результатов:

```
> var page1 = db.foo.find().sort({"date" : -1}).limit(100)
```

Затем, предполагая, что дата уникальна, мы можем использовать значение `"date"` последнего документа в качестве критерия для получения следующей страницы:

```
var latest = null;

//Отображаем первую страницу;
while (page1.hasNext()) {
    latest = page1.next();
    display(latest);
}

//Получаем следующую страницу;
var page2 = db.foo.find({"date" : {"$lt" : latest.date}});
page2.sort({"date" : -1}).limit(100);
```

Теперь в запрос не нужно включать функцию `skip`.

Поиск случайного документа

Одна из довольно распространенных проблем состоит в том, как получить случайный документ из коллекции. Наивное (и медленное) решение состоит в том, чтобы подсчитать количество документов, а затем использовать метод `find`, пропуская случайное количество документов от нуля до размера коллекции:

```
> // Не используйте этот вариант;
> var total = db.foo.count()
> var random = Math.floor(Math.random()*total)
> db.foo.find().skip(random).limit(1)
```

На самом деле получать случайный элемент таким способом крайне неэффективно: вам нужно сделать подсчет (что может быть затратно, если вы используете критерии), а пропуск большого количества элементов может занять много времени.

Это займет немного времени, но если вы знаете, что будете искать случайный элемент в коллекции, существует гораздо более эффективный способ сделать это. Хитрость заключается в том, чтобы добавлять дополнительный случайный ключ к каждому документу при его вставке. Напри-

мер, если мы используем оболочку, то могли бы использовать функцию `Math.random()` (которая создает случайное число от 0 до 1):

```
> db.people.insertOne({"name" : "joe", "random" : Math.random()})
> db.people.insertOne({"name" : "john", "random" : Math.random()})
> db.people.insertOne({"name" : "jim", "random" : Math.random()})
```

Теперь, если хотим найти случайный документ из коллекции, мы можем вычислить случайное число и использовать его в качестве критерия запроса, вместо того чтобы применять функцию `skip`:

```
> var random = Math.random()
> result = db.people.findOne({"random" : {"$gt" : random}})
```

Существует небольшая вероятность того, что значение `random` будет больше, чем любое из «случайных» значений в коллекции, и результаты не будут возвращены. От этого можно защититься, просто вернув документ в другом направлении:

```
> if (result == null) {
... result = db.people.findOne({"random" : {"$lte" : random}})
... }
```

Если в коллекции нет никаких документов, этот метод в конечном итоге вернет значение `null`, что не лишено смысла.

Данный метод можно использовать с произвольно сложными запросами; просто убедитесь, что у вас есть индекс, который включает в себя случайный ключ. Например, если мы хотим найти случайного водопроводчика в Калифорнии, можно создать индекс для `"profession"`, `"state"` и `"random"`:

```
> db.people.ensureIndex({"profession" : 1, "state" : 1, "random" : 1})
```

Это позволит нам быстро найти случайный результат (дополнительную информацию по индексации *см.* в главе 5).

Бесконечные курсоры

У курсора есть две стороны: клиентский курсор и курсор базы данных, представляющий клиентскую сторону. Мы уже говорили о клиентской стороне, но кратко рассмотрим, что происходит на сервере.

На стороне сервера курсор занимает память и ресурсы. Как только у курсора заканчиваются результаты или клиент отправляет сообщение, сообщающее ему о прекращении работы, база данных может освободить ресурсы, которые она использовала. Освобождение этих ресурсов позволяет базе данных использовать их для других целей, и это хорошо, поэтому мы хотим обеспечить быстрое освобождение курсоров (в пределах разумного).

Есть пара условий, которые могут вызвать остановку работы (и последующую очистку) курсора. Во-первых, когда курсор завершает итерацию результатов сопоставления, он очищается самостоятельно. Другой способ заключается в том, что когда курсор выходит из области видимости на стороне клиента, драйверы отправляют базе данных специальное сообщение, чтобы уведомить ее, что она может остановить курсор.

Наконец, даже если пользователь не прошел итерацию по всем результатам и курсор по-прежнему находится в области видимости, после 10 минут бездействия курсор базы данных остановится автоматически. Таким образом, если клиент аварийно завершает работу или у него какая-то неисправность, в MongoDB не будет тысяч открытых курсоров.

Эта «смерть по тайм-ауту», как правило, является желаемым поведением: очень немногие приложения ожидают, что их пользователи будут сидеть без дела несколько минут в ожидании результатов. Тем не менее иногда вы можете знать, что вам нужно, чтобы курсора хватило надолго. В этом случае многие драйверы реализовали функцию под названием `immortal`, или аналогичный механизм, который велит базе данных не превышать время ожидания курсора. Если вы отключите тайм-аут курсора, то должны выполнить итерацию всех его результатов или уничтожить его, дабы убедиться, что он закрыт. В противном случае он будет находиться в базе данных, пожирая ресурсы, до тех пор, пока сервер не будет перезагружен.

Часть II

Разработка приложения

Глава 5

Индексы

Эта глава знакомит вас с индексами MongoDB. Индексы позволяют эффективно выполнять запросы. Они являются важной частью разработки приложений и даже необходимы для определенных типов запросов. В этой главе мы рассмотрим:

- что такое индексы и почему нужно их использовать;
- как решить, какие поля индексировать;
- как обеспечить и оценить использование индекса;
- административные сведения касательно создания и удаления индексов.

Как вы увидите, выбор правильных индексов для ваших коллекций имеет решающее значение для производительности.

Знакомство с индексами

Индекс базы данных похож на указатель книги. Вместо того чтобы просматривать всю книгу, СУБД использует ярлык и просто просматривает упорядоченный список со ссылками на содержимое. Это позволяет MongoDB выполнять запросы на порядок быстрее.

Запрос, который не использует индекс, называется *сканированием коллекции*. Это означает, что сервер должен «просмотреть всю книгу», чтобы найти результаты запроса. Этот процесс в основном представляет собой то, что вы делали бы, если бы искали информацию в книге, где нет оглавления: вы бы начали с первой страницы и прочитали все это целиком. В целом не нужно заставлять сервер выполнять сканирование коллекций, поскольку в случае с большими коллекциями этот процесс будет очень медленным.

Давайте посмотрим на пример. Для начала мы создадим коллекцию с 1 млн документов (или 10 млн, или 100, если вы наберетесь терпения):

```
> for (i=0; i<1000000; i++) {
...     db.users.insertOne(
```

```
...         {
...             "i" : i,
...             "username" : "user"+i,
...             "age" : Math.floor(Math.random()*120),
...             "created" : new Date()
...         }
...     );
... }
```

Затем мы рассмотрим различия в производительности запросов к этой коллекции: сначала без индекса, а затем с индексом.

Если мы выполняем запрос к этой коллекции, то можем использовать команду `explain`, чтобы увидеть, что делает MongoDB при выполнении запроса. Предпочтительнее использовать команду `explain` через вспомогательный метод курсора, который обертывает эту команду. Метод курсора `explain` предоставляет информацию о выполнении различных CRUD-операций. Этот метод может быть запущен в нескольких режимах детализации. Мы рассмотрим режим `executionStats`, поскольку это помогает нам понять эффект использования индекса для удовлетворения запросов. Попробуйте запросить конкретное имя пользователя, чтобы увидеть пример:

```
> db.users.find({"username": "user101"}).explain("executionStats")
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        "namespace" : "test.users",
        "indexFilterSet" : false,
        "parsedQuery" : {
            "username" : {
                "$eq" : "user101"
            }
        },
        "winningPlan" : {
            "stage" : "COLLSCAN",
            "filter" : {
                "username" : {
                    "$eq" : "user101"
                }
            },
            "direction" : "forward"
        },
        "rejectedPlans" : [ ]
    },
    "executionStats" : {
    "executionSuccess" : true,
```

```
    "nReturned" : 1,
    "executionTimeMillis" : 419,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 1000000,
    "executionStages" : {
        "stage" : "COLLSCAN",
        "filter" : {
            "username" : {
                "$eq" : "user101"
            }
        },
        "nReturned" : 1,
        "executionTimeMillisEstimate" : 375,
        "works" : 1000002,
        "advanced" : 1,
        "needTime" : 1000000,
        "needYield" : 0,
        "saveState" : 7822,
        "restoreState" : 7822,
        "isEOF" : 1,
        "invalidates" : 0,
        "direction" : "forward",
        "docsExamined" : 1000000
        }
    },
    "serverInfo" : {
        "host" : "eoinbrazil-laptop-osx",
        "port" : 27017,
        "version" : "4.0.12",
        "gitVersion" : "5776e3cbf9e7afe86e6b29e22520ffb6766e95d4"
    },
    "ok": 1
}
```

При рассмотрении вывода метода `"explain Output"` на стр. 150 мы объясним поля вывода; пока что почти все их можно игнорировать. В этом примере мы хотим рассмотреть вложенный документ, который является значением поля `"executionStats"`. В этом документе `"totalDocsExamined"` – количество документов, которые просматривала MongoDB при попытке удовлетворить запрос, а это, как вы видите, каждый документ в коллекции. То есть MongoDB пришлось просмотреть каждое поле в каждом документе. На моем ноутбуке это заняло почти полсекунды (в поле `"executionTimeMillis"` отображается количество миллисекунд, которое было потрачено на выполнение запроса).

Поле `"nReturned"` документа `"executionStats"` показывает количество возвращаемых результатов: 1, что имеет смысл, поскольку существует только один пользователь с именем `"user101"`. Обратите внимание, что MongoDB должна была просмотреть все документы в коллекции на предмет совпадений, потому что она не знала, что имена пользователей уникальны.

Чтобы MongoDB могла эффективно отвечать на запросы, все шаблоны запросов в вашем приложении должны поддерживаться индексом. Под шаблонами запросов мы просто подразумеваем различные типы вопросов, которые ваше приложение задает базе данных. В этом примере мы запросили коллекцию *users* по имени пользователя. Это пример конкретного шаблона запроса. Во многих приложениях один индекс будет поддерживать несколько шаблонов запросов. Мы обсудим настройку индексов для шаблонов запросов в следующем разделе.

Создание индекса

Теперь давайте попробуем создать индекс по полю `"username"`. Для создания индекса мы будем использовать метод коллекции `createIndex`:

```
> db.users.createIndex({"username" : 1})
{
    "createdCollectionAutomatically" : false,
    "numIndexesBefore" : 1,
    "numIndexesAfter" : 2,
    "ok" : 1
}
```

Создание индекса должно занять не более нескольких секунд, если только вы не сделали свою коллекцию очень уж большой. Если спустя несколько секунд после вызова функция `createIndex` не возвращается, запустите метод `db.currentOp()` (в другой оболочке) или проверьте журнал *mongod*, чтобы увидеть прогресс сборки индекса.

После завершения построения индекса попробуйте повторить исходный запрос:

```
> db.users.find({"username": "user101"}).explain("executionStats")
{
    "queryPlanner": {
        "plannerVersion" : 1,
        "namespace" : "test.users",
        "indexFilterSet" : false,
        "parsedQuery" : {
            "username" : {
                "$eq" : "user101"
            }
```

```
        },
        "winningPlan" : {
            "stage" : "FETCH",
            "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                    "username" : 1
                },
                "indexName" : "username_1",
                "isMultiKey" : false,
                "multiKeyPaths" : {
                    "username" : []
                },
                "isUnique" : false,
                "isSparse" : false,
                "isPartial" : false,
                "indexVersion" : 2,
                "direction" : "forward",
                "indexBounds" : {
                    "username" : [
                        "[\"user101\", \"user101\"]"
                    ]
                }
            }
        },
        "rejectedPlans" : []
    },
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 1,
        "executionTimeMillis" : 1,
        "totalKeysExamined" : 1,
        "totalDocsExamined" : 1,
        "executionStages" : {
            "stage" : "FETCH",
            "nReturned" : 1,
            "executionTimeMillisEstimate" : 0,
            "works" : 2,
            "advanced" : 1,
            "needTime" : 0,
            "needYield" : 0,
            "saveState" : 0,
            "restoreState" : 0,
            "isEOF" : 1,
            "invalidates" : 0,
```

```
            "docsExamined" : 1,
            "alreadyHasObj" : 0,
            "inputStage" : {
                "stage" : "IXSCAN",
                "nReturned" : 1,
                "executionTimeMillisEstimate" : 0,
                "works" : 2,
                "advanced" : 1,
                "needTime" : 0,
                "needYield" : 0,
                "saveState" : 0,
                "restoreState" : 0,
                "isEOF" : 1,
                "invalidates" : 0,
                "keyPattern" : {
                    "username" : 1
                },
                "indexName" : "username_1",
                "isMultiKey" : false,
                "multiKeyPaths" : {
                    "username" : []
                },
                "isUnique" : false,
                "isSparse" : false,
                "isPartial" : false,
                "indexVersion" : 2,
                "direction" : "forward",
                "indexBounds" : {
                    "username" : [
                        "[\"user101\", \"user101\"]"
                    ]
                },
                "keysExamined" : 1,
                "seeks" : 1,
                "dupsTested" : 0,
                "dupsDropped" : 0,
                "seenInvalidated" : 0
            }
        }
    },
    "serverInfo" : {
        "host" : "eoinbrazil-laptop-osx",
        "port" : 27017,
        "version" : "4.0.12",
        "gitVersion" : "5776e3cbf9e7afe86e6b29e22520ffb6766e95d4"
```

```
    },
    "ok": 1
}
```

Этот вывод более сложный, но пока вы можете продолжать игнорировать все поля, кроме `"nReturned"`, `"totalDocsExamined"` и `"executionTimeMillis"`, во вложенном документе `"executionStats"`. Как видите, запрос теперь почти мгновенный, и, что еще лучше, здесь мы видим аналогичное время выполнения при запросе, например для любого имени пользователя:

```
> db.users.find({"username" : "user999999"}).explain("executionStats")
```

Индекс может существенно изменить время запроса. Однако у индексов есть своя цена: операции записи (вставки, обновления и удаления), которые изменяют индексированное поле, займут больше времени. Это связано с тем, что в дополнение к обновлению документа MongoDB должна обновлять индексы при изменении ваших данных. Как правило, компромисс стоит того. Сложности начинаются, когда нужно выяснить, какие поля индексировать.

Индексы MongoDB работают почти так же, как и индексы типичных реляционных СУБД, поэтому если вы знакомы с ними, можете просто просмотреть этот раздел, чтобы ознакомиться с особенностями синтаксиса.

Чтобы выбрать, для каких полей создавать индексы, просмотрите ваши частые запросы и запросы, которые должны быть быстрыми, и попробуйте найти общий набор ключей. Например, в предыдущем примере мы запрашивали `"username"`. Если это очень распространенный запрос или он становился препятствием, индексация `"username"` будет хорошим выбором. Однако если это необычный запрос или запрос, выполняемый только администраторами, которым все равно, сколько времени это займет, это не лучший выбор.

Знакомство с составными индексами

Цель индекса – сделать ваши запросы максимально эффективными. Для множества шаблонов запросов необходимо создавать индексы на основе двух или более ключей. Например, индекс хранит все свои значения в отсортированном порядке, что значительно ускоряет сортировку документов по индексируемому ключу. Однако индекс может помочь в сортировке, только если он является префиксом сортировки. Например, индекс по `"username"` не сильно поможет:

```
> db.users.find().sort({"age" : 1, "username" : 1})
```

Здесь мы выполняем сортировку по возрасту (`"age"`), а затем по имени пользователя (`"username"`), поэтому строгая сортировка по имени пользователя не очень помогает. Чтобы оптимизировать это, можно создать индекс по `"age"` и `"username"`:

```
> db.users.createIndex({"age" : 1, "username" : 1})
```

Это называется *составным индексом* и полезно, если ваш запрос имеет несколько направлений сортировки или несколько ключей в критериях. Составной индекс – это индекс по нескольким полям.

Предположим, у нас есть коллекция *users*, которая выглядит примерно так, как показано ниже, если мы запустим запрос без сортировки (естественный порядок):

```
> db.users.find({}, {"_id" : 0, "i" : 0, "created" : 0})
{ "username" : "user0", "age" : 69 }
{ "username" : "user1", "age" : 50 }
{ "username" : "user2", "age" : 88 }
{ "username" : "user3", "age" : 52 }
{ "username" : "user4", "age" : 74 }
{ "username" : "user5", "age" : 104 }
{ "username" : "user6", "age" : 59 }
{ "username" : "user7", "age" : 102 }
{ "username" : "user8", "age" : 94 }
{ "username" : "user9", "age" : 7 }
{ "username" : "user10", "age" : 80 }
...
```

Если мы проиндексируем эту коллекцию по `{"age" : 1, "username" : 1}`, индекс будет иметь форму, которую можно представить следующим образом:

```
[0, "user100020"] -> 8623513776
[0, "user1002"] -> 8599246768
[0, "user100388"] -> 8623560880
...
[0, "user100414"] -> 8623564208
[1, "user100113"] -> 8623525680
[1, "user100280"] -> 8623547056
[1, "user100551"] -> 8623581744
...
[1, "user100626"] -> 8623591344
[2, "user100191"] -> 8623535664
[2, "user100195"] -> 8623536176
```

```
[2, "user100197"] -> 8623536432
...
```

Каждая запись индекса содержит возраст и имя пользователя и указывает на идентификатор записи. Идентификатор записи используется внутренней подсистемой хранения для определения местоположения данных для документа. Обратите внимание, что поля `"age"` располагаются строго в порядке возрастания, и в каждом из них имена пользователей также располагаются в порядке возрастания. В этом примере набора данных в каждом таком поле содержится приблизительно 8000 имен пользователей, ассоциированных с ним. Сюда мы включили только те, что необходимы для передачи общей идеи.

То, как MongoDB использует этот индекс, зависит от типа запроса, который вы делаете. Вот три наиболее распространенных способа:

`db.users.find({"age" : 21}).sort({"username" : -1})`

Это запрос на равенство, с помощью которого мы ищем одно значение. С таким значением документов может быть несколько. Из-за второго поля в индексе результаты уже находятся в правильном порядке для сортировки: MongoDB может начинать с последнего совпадения для `{"age" : 21}` и проходить по индексу в следующем порядке:

```
[21, "user100154"] -> 8623530928
[21, "user100266"] -> 8623545264
[21, "user100270"] -> 8623545776
[21, "user100285"] -> 8623547696
[21, "user100349"] -> 8623555888
...
```

Данный тип запроса очень эффективен: MongoDB может перейти непосредственно к правильному возрасту и не нуждается в сортировке результатов, потому что при обходе индекса данные возвращаются в правильном порядке.

Обратите внимание, что направление сортировки не имеет значения: MongoDB может перемещаться по индексу в любом направлении.

`db.users.find({"age" : {"$gte" : 21, "$lte" : 30}})`

Это запрос по диапазону, который ищет документы, совпадающие с несколькими значениями (в данном случае это все возрасты от 21 до 30). MongoDB будет использовать первый ключ в индексе, `"age"`, чтобы вернуть совпадающие документы, например:

```
[21, "user100154"] -> 8623530928
[21, "user100266"] -> 8623545264
[21, "user100270"] -> 8623545776
...
[21, "user999390"] -> 8765250224
[21, "user999407"] -> 8765252400
[21, "user999600"] -> 8765277104
[22, "user100017"] -> 8623513392
...
[29, "user999861"] -> 8765310512
[30, "user100098"] -> 8623523760
[30, "user100155"] -> 8623531056
[30, "user100168"] -> 8623532720
...
```

В целом если MongoDB использует индекс для запроса, он вернет полученные документы в порядке индексирования.

```
db.users.find({"age" : {"$gte" : 21, "$lte" : 30}}).sort({"username" : 1})
```

Это многозначный запрос, как и предыдущий, но на этот раз у него есть сортировка. Как и прежде, MongoDB будет использовать индекс для соответствия критериям. Однако индекс не возвращает имена пользователей в отсортированном порядке, и запрос запросил сортировку результатов по имени пользователя. Это означает, что MongoDB нужно будет отсортировать результаты в памяти, перед тем как вернуть их, а не просто просматривать индекс, в котором документы уже отсортированы в нужном порядке. Данный тип запроса обычно менее эффективен.

Конечно же, скорость зависит от того, сколько результатов соответствует вашим критериям: если ваш набор результатов представляет собой только пару документов, MongoDB не придется особо трудиться, чтобы отсортировать их, но если результатов будет больше, дело пойдет медленнее или работать вообще ничего не будет. Если у вас более 32 МБ результатов, MongoDB просто выдаст ошибку, отказавшись сортировать такое количество данных:

```
Error: error: {
    "ok" : 0,
    "errmsg" : "Executor error during find command:
OperationFailed: Sort operation used more than the maximum
33554432 bytes of RAM. Add an index, or specify a smaller limit.",
    "code" : 96,
    "codeName" : "OperationFailed"
 }
```

Если вы хотите избежать этой ошибки, нужно создать индекс, поддерживающий операцию сортировки (https://docs.mongodb.com/manual/reference/method/cursor.sort/index.html#sort-index-use), или используйте функцию `sort` в сочетании с функцией `limit`, чтобы уменьшить результаты до уровня ниже 32 МБ.

Еще один индекс, который можно использовать в последнем примере, – это те же ключи в обратном порядке: `{"username" : 1, "age" : 1}`. Затем MongoDB будет проходить по всем элементам индекса, но в том порядке, в котором вы хотите их вернуть. Она выберет совпадающие документы, используя часть индекса `"age"`:

```
[user0, 4]
[user1, 67]
[user10, 11]
[user100, 92]
[user1000, 10]
[user10000, 31]
[user100000, 21] -> 8623511216
[user100001, 52]
[user100002, 69]
[user100003, 27] -> 8623511600
[user100004, 22] -> 8623511728
[user100005, 95]
...
```

Это хорошо тем, что не требует каких-либо гигантских сортировок в памяти. Тем не менее она должна сканировать весь индекс, чтобы найти все совпадения. Ставить ключ сортировки первым – это, как правило, хорошая стратегия при разработке составных индексов. Как мы вскоре увидим, это одна из лучших практик при рассмотрении того, как составлять составные индексы с учетом запросов на равенство, многозначных запросов и сортировки.

Как MongoDB выбирает индекс

Теперь давайте посмотрим, как MongoDB выбирает индекс, чтобы выполнить запрос. Представим, что у нас есть пять индексов. Когда приходит запрос, MongoDB смотрит на *форму* запроса. Форма связана с тем, в каких полях производится поиск, и дополнительной информацией, такой как наличие или отсутствие сортировки. Основываясь на этой информации, система идентифицирует набор индексов-кандидатов, которые она может использовать при удовлетворении запроса.

Предположим, мы получили запрос, и три из пяти наших индексов определены как кандидаты на этот запрос. Затем MongoDB создаст три плана запросов, по одному для каждого из этих индексов, и выполнит запрос в трех параллельных потоках, каждый из которых использует свой индекс. Задача состоит в том, чтобы увидеть, какой из них способен быстрее вернуть результаты.

Визуально можно рассматривать это как гонку, как показано на рис. 5.1. Идея заключается в том, что первым планом выполнения запроса для достижения целевого состояния является победитель. Но что еще более важно, в дальнейшем он будет выбран в качестве индекса для запросов, которые имеют ту же форму запроса. Планы состязаются друг с другом на протяжении некоего периода (называемого пробным периодом), после которого результаты каждой гонки используются для расчета общего плана выигрыша.

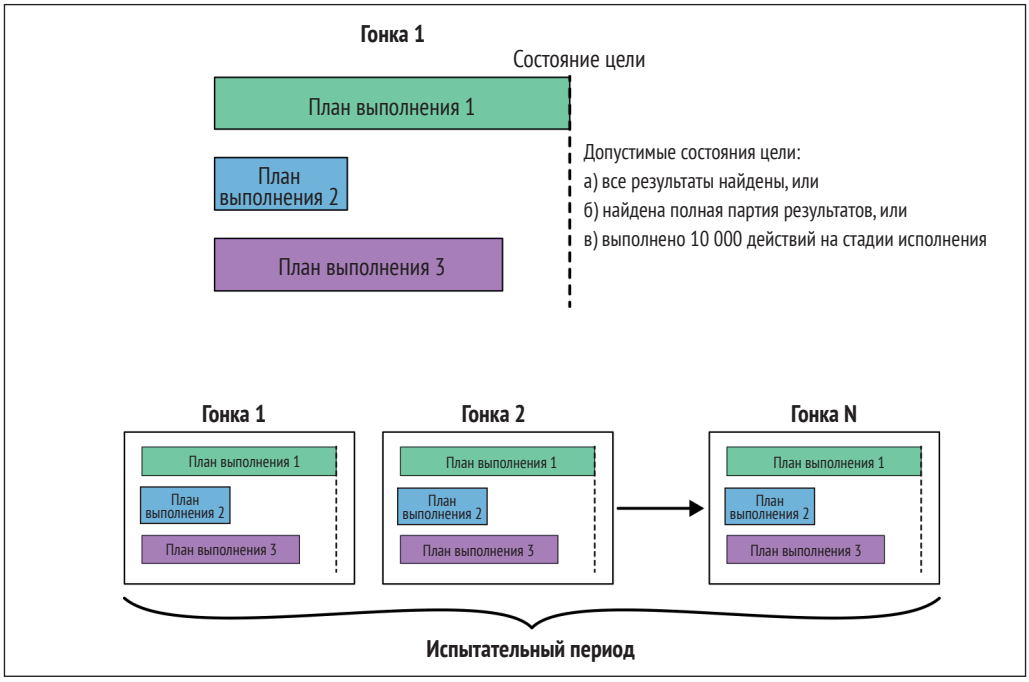

Рис. 5.1. Как планировщик запросов MongoDB выбирает индекс

Чтобы выиграть гонку, поток запросов должен быть первым, чтобы либо вернуть все результаты запроса, либо вернуть пробное число результатов в порядке сортировки. Эта часть порядка сортировки важна, учитывая, насколько затратно выполнять сортировку в памяти.

Реальная ценность состязания нескольких планов запросов друг с другом заключается в том, что для последующих запросов, имеющих одинаковую форму запроса, сервер MongoDB будет знать, какой индекс выбрать. Сер-

вер поддерживает кеш планов выполнения запросов. Выигрышный план хранится в кеше для последующего использования для запросов этой формы. Со временем, по мере изменения коллекции и изменения индексов, в конечном итоге план выполнения запроса может быть удален из кеша, и MongoDB снова будет экспериментировать с возможными планами, чтобы найти тот, который лучше всего подходит для текущей коллекции и набора индексов. Другие события, которые приведут к исключению планов из кеша, если мы перестроим данный индекс, добавим или удалим индекс или явно очистим кеш плана. Наконец, кеш плана выполнения запросов не выдерживает перезапуска (очищается при перезапуске) процесса *mongod*.

Использование составных индексов

В предыдущих разделах мы использовали составные индексы, которые представляют собой индексы с несколькими ключами. Составные индексы немного сложнее рассматривать, чем те, что имеют один ключ, но они очень мощные. В этом разделе они описываются более подробно.

Здесь мы рассмотрим пример, который дает представление о типе мышления, необходимом при разработке составных индексов. Цель состоит в том, чтобы наши операции чтения и записи были максимально эффективными, но, как и во многих других случаях, это требует определенного предварительного мышления и экспериментов.

Чтобы убедиться, что мы получили правильные индексы, необходимо протестировать наши индексы при реальных рабочих нагрузках и внести в них коррективы. Тем не менее существуют некоторые передовые рекомендации, которые можно применять при разработке индексов.

Во-первых, нам нужно рассмотреть селективность индекса. Нас интересует, в какой степени для данного шаблона запроса индекс будет минимизировать количество отсканированных записей. Нам необходимо учитывать избирательность в свете всех операций, необходимых для того, чтобы удовлетворить запрос, а иногда и пойти на компромиссы. Нам нужно будет рассмотреть, например, как обрабатываются сортировки.

Давайте посмотрим на один пример. Мы будем использовать набор данных студентов, содержащий около миллиона записей. Документы в этом наборе данных выглядят примерно так:

```
{
    "_id": ObjectId(«585d817db4743f74e2da067c»),
    "student_id" : 0,
    "scores": [
        {
            "type" : "exam",
            "score" : 38.05000060199827
        },
```

```
        {
            "type" : "quiz",
            "score" : 79.45079445008987
        },
        {
            "type" : "homework",
            "score" : 74.50150548699534
        },
        {
            "type" : "homework",
            "score" : 74.68381684615845
        }
    ],
    "class_id" : 127
}
```

Мы начнем с двух индексов и посмотрим, как MongoDB использует их (или не использует), чтобы выполнить запросы. Эти два индекса создаются следующим образом:

```
> db.students.createIndex({"class_id": 1})
> db.students.createIndex({student_id: 1, class_id: 1})
```

При работе с этим набором данных мы рассмотрим приведенный ниже запрос, поскольку он иллюстрирует некоторые проблемы, которые нам необходимо учитывать при разработке своих индексов:

```
> db.students.find({student_id:{$gt:500000}, class_id:54})
... .sort({student_id:1})
... .explain("executionStats")
```

Обратите внимание, что здесь мы запрашиваем все записи с идентификатором, превышающим 500 000, то есть это примерно половина записей. Мы также ограничиваем поиск записями для класса с ID 54. В этом наборе данных представлено около 500 классов. Наконец, мы выполняем сортировку в порядке возрастания на основе `"student_id"`. Обратите внимание, что это то же поле, для которого мы делаем многозначный запрос. В этом примере мы рассмотрим статистику выполнения, которую предоставляет метод `explain`, чтобы проиллюстрировать, как MongoDB будет обрабатывать этот запрос.

Если мы выполним запрос, вывод метода `explain` сообщит нам, как MongoDB использовала индексы, чтобы выполнить его:

```
{
    "queryPlanner": {
        "plannerVersion": 1,
```

```
"namespace": "school.students",
"indexFilterSet": false,
"parsedQuery": {
    "$and": [
        {
            "class_id": {
                "$eq": 54
            }
        },
        {
            "student_id": {
                "$gt": 500000
            }
        }
    ]
},
"winningPlan": {
    "stage": "FETCH",
    "inputStage": {
        "stage": "IXSCAN",
        "keyPattern": {
            "student_id": 1,
            "class_id": 1
        },
        "indexName": "student_id_1_class_id_1",
        "isMultiKey": false,
        "multiKeyPaths": {
            "student_id": [],
            "class_id": []
        },
        "isUnique": false,
        "isSparse": false,
        "isPartial": false,
        "indexVersion": 2,
        "direction": "forward",
        "indexBounds": {
            "student_id": [
                "(500000.0, inf.0]"
            ],
            "class_id": [
                "[54.0, 54.0]"
            ]
        }
    }
},
```

```
        "rejectedPlans": [
            {
                "stage": "SORT",
                "sortPattern": {
                    "student_id": 1
                },
                "inputStage": {
                    "stage": "SORT_KEY_GENERATOR",
                    "inputStage": {
                        "stage": "FETCH",
                        "filter": {
                            "student_id": {
                                "$gt": 500000
                            }
                        },
                        "inputStage": {
                            "stage": "IXSCAN",
                            "keyPattern": {
                                "class_id": 1
                            },
                            "indexName": "class_id_1",
                            "isMultiKey": false,
                            "multiKeyPaths": {
                                "class_id": []
                            },
                            "isUnique": false,
                            "isSparse": false,
                            "isPartial": false,
                            "indexVersion": 2,
                            "direction": "forward",
                            "indexBounds": {
                                "class_id": [
                                    "[54.0, 54.0]"
                                ]
                            }
                        }
                    }
                }
            }
        ]
    },
    "executionStats": {
        "executionSuccess": true,
        "nReturned": 9903,
        "executionTimeMillis": 4325,
        "totalKeysExamined": 850477,
```

```
"totalDocsExamined": 9903,
"executionStages": {
    "stage": "FETCH",
    "nReturned": 9903,
    "executionTimeMillisEstimate": 3485,
    "works": 850478,
    "advanced": 9903,
    "needTime": 840574,
    "needYield": 0,
    "saveState": 6861,
    "restoreState": 6861,
    "isEOF": 1,
    "invalidates": 0,
    "docsExamined": 9903,
    "alreadyHasObj": 0,
    "inputStage": {
        "stage": "IXSCAN",
        "nReturned": 9903,
        "executionTimeMillisEstimate": 2834,
        "works": 850478,
        "advanced": 9903,
        "needTime": 840574,
        "needYield": 0,
        "saveState": 6861,
        "restoreState": 6861,
        "isEOF": 1,
        "invalidates": 0,
        "keyPattern": {
            "student_id": 1,
            "class_id": 1
        },
        "indexName": "student_id_1_class_id_1",
        "isMultiKey": false,
        "multiKeyPaths": {
            "student_id": [],
            "class_id": []
        },
        "isUnique": false,
        "isSparse": false,
        "isPartial": false,
        "indexVersion": 2,
        "direction": "forward",
        "indexBounds": {
            "student_id": [
                "(500000.0, inf.0]"
```

```
                    ],
                    "class_id": [
                        "[54.0, 54.0]"
                    ]
                },
                "keysExamined": 850477,
                "seeks": 840575,
                "dupsTested": 0,
                "dupsDropped": 0,
                "seenInvalidated": 0
            }
        }
    },
    "serverInfo": {
        "host": "SGB-MBP.local",
        "port": 27017,
        "version": "3.4.1",
        "gitVersion": "5e103c4f5583e2566a45d740225dc250baacfbd7"
    },
    "ok": 1
}
```

Как и большая часть вывода данных из MongoDB, вывод `explain` идет в формате JSON. Давайте сначала посмотрим на нижнюю половину этого вывода, который почти полностью представляет собой статистику выполнения. Поле `"executionStats"` содержит статистику, которая описывает выполнение завершенного запроса для выигрышного плана. Мы рассмотрим планы выполнения запросов и вывод плана `explain` чуть позже.

В рамках `"executionStats"` мы для начала взглянем на `"totalKeysExamined"`. Тут показано, сколько ключей в рамках индекса прошла MongoDB, чтобы сгенерировать набор результатов. Можно сравнить `"totalKeysExamined"` с `"nReturned"`, чтобы понять, какую часть индекса MongoDB пришлось пройти для нахождения только документов, соответствующих запросу. В этом случае было проверено 850 477 индексных ключей, чтобы найти 9,903 совпадающих документа.

Это означает, что индекс, используемый для выполнения этого запроса, был не очень избирательным. Это дополнительно подчеркивается тем фактом, что выполнение данного запроса заняло более 4,3 секунды, как указано в поле `"executionTimeMillis"`. Избирательность является одной из наших ключевых целей при разработке индекса, поэтому давайте выясним, где мы ошиблись с существующими индексами для этого запроса.

В верхней части вывода `explain` находится выигрышный план (*см.* поле `"winning Plan"`). План выполнения запроса описывает шаги, которые использовала MongoDB для его выполнения. Это конкретный ре-

зультат соревнования нескольких разных планов запросов друг с другом в формате JSON. В частности, нас интересует, какие индексы использовались и пришлось ли MongoDB выполнять сортировку в памяти. Ниже выигрышного плана находятся отклоненные планы. Мы рассмотрим оба.

В этом случае в выигрышном плане использовался составной индекс на основе `"student_id"` и `"class_id"`, что явно видно в этой части вывода `explain`:

```
"winningPlan": {
    "stage": "FETCH",
    "inputStage": {
        "stage": "IXSCAN",
        "keyPattern": {
            "student_id": 1,
            "class_id": 1
        },},
```

Вывод представляет план выполнения запроса в виде дерева этапов. Этап может иметь один или несколько входных этапов, в зависимости от количества дочерних этапов. Этап ввода предоставляет документы или индексные ключи своему родителю. В этом случае был один этап ввода, сканирование индекса, и это сканирование предоставило идентификаторы записей для документов, соответствующих запросу к его родителю, этапу `"FETCH"`. После этого на этапе `"FETCH"` будут извлечены сами документы, которые будут возвращаться пакетами по запросу клиента.

Проигрышный план выполнения запроса – он только один – использовал бы индекс на базе `"class_id"`, но тогда ему пришлось бы выполнять сортировку в памяти. Вот что означает следующая часть этого конкретного плана. Когда вы видите стадию `"SORT"` в плане запроса, это означает, что MongoDB не смогла бы отсортировать набор результатов в базе данных с использованием индекса, а вместо этого ей пришлось бы выполнить сортировку в памяти:

```
"rejectedPlans": [
    {
        "stage": "SORT",
        "sortPattern": {
            "student_id": 1
        },
```

В случае с этим запросом индекс, который выиграл, смог вернуть отсортированный вывод. Чтобы выиграть, ему нужно было лишь достигнуть пробного числа отсортированных результирующих документов. Чтобы выиграл другой план, эта цепочка запросов должна была бы сначала вернуть весь набор результатов (почти 10 000 документов), поскольку затем их нужно было бы отсортировать в памяти.

Проблема здесь заключается в избирательности. Многозначный запрос, который мы выполняем, задает широкий диапазон значений `"student_id"`, поскольку он запрашивает записи, для которых `"student_id"` больше 500 000. Это примерно половина записей в нашей коллекции. Опять же, вот запрос, который мы выполняем:

```
> db.students.find({student_id:{$gt:500000}, class_id:54})
...           .sort({student_id:1})
...           .explain("executionStats")
```

Теперь я уверен, что вы видите, куда мы направляемся. Этот запрос содержит как многозначную часть, так и часть, касающуюся равенства. Часть, касающаяся равенства, состоит в том, что мы запрашиваем все записи, в которых `"class_id"` равен 54. В этом наборе данных только около 500 классов, и хотя в этих классах большое количество учащихся с оценками, `"class_id"` будет служить гораздо более избирательной основой для выполнения этого запроса. Именно это значение ограничивает наш набор результатов до 10 000 записей, а не тех приблизительно 850 000, которые были определены многозначной частью этого запроса.

Другими словами, было бы лучше, учитывая имеющиеся у нас индексы, если бы мы использовали индекс, основанный только на `"class_id"` – том, что указан в проигрышном плане. MongoDB предоставляет два способа заставить базу данных использовать определенный индекс. Однако я не могу особо подчеркнуть, что вы должны с осторожностью использовать эти способы переопределения того, что будет результатом работы планировщика запросов. Это не те методы, которые стоит применять при развертывании в среде эксплуатации.

Метод курсора `hint` позволяет нам указать конкретный индекс, который нужно использовать, указав его форму или имя. Фильтр индекса использует форму запроса, которая является комбинацией запроса, сортировки и спецификации проекции. Функция `planCacheSetFilter` может применяться с фильтром индекса, чтобы ограничить оптимизатор запросов только учетом индексов, указанных в индексном фильтре. Если для формы запроса существует индексный фильтр, MongoDB проигнорирует метод `hint`. Индексные фильтры сохраняются только на время процесса сервера *mongod*; после выключения они исчезают.

Если мы слегка изменим наш запрос, чтобы использовать метод `hint`, как в следующем примере, вывод `explain` будет совсем другим:

```
> db.students.find({student_id:{$gt:500000}, class_id:54})
...           .sort({student_id:1})
...           .hint({class_id:1})
...           .explain("executionStats")
```

Полученный вывод показывает, что теперь количество отсканированных индексных ключей сократилось от приблизительно 850 000 до примерно 20 000, чтобы мы могли получить свой набор результатов, состоящий из чуть менее 10 000 ключей. Кроме того, время выполнения составляет всего 272 миллисекунды, а не 4,3 секунды, которые мы видели в плане запроса, где использовался другой индекс:

```
{
    "queryPlanner": {
        "plannerVersion": 1,
        "namespace": "school.students",
        "indexFilterSet": false,
        "parsedQuery": {
            "$and": [
                {
                    "class_id": {
                        "$eq": 54
                    }
                },
                {
                    "student_id": {
                        "$gt": 500000
                    }
                }
            ]
        },
        "winningPlan": {
            "stage": "SORT",
            "sortPattern": {
                "student_id": 1
            },
            "inputStage": {
                "stage": "SORT_KEY_GENERATOR",
                "inputStage": {
                    "stage": "FETCH",
                    "filter": {
                        "student_id": {
                            "$gt": 500000
                        }
                    },
                    "inputStage": {
                        "stage": "IXSCAN",
                        "keyPattern": {
                            "class_id": 1
                        },
```

```
                    "indexName": "class_id_1",
                    "isMultiKey": false,
                    "multiKeyPaths": {
                        "class_id": []
                    },
                    "isUnique": false,
                    "isSparse": false,
                    "isPartial": false,
                    "indexVersion": 2,
                    "direction": "forward",
                    "indexBounds": {
                        "class_id": [
                            "[54.0, 54.0]"
                        ]
                    }
                }
            }
        }
    },
    "rejectedPlans": []
},
"executionStats": {
    "executionSuccess": true,
    "nReturned": 9903,
    "executionTimeMillis": 272,
    "totalKeysExamined": 20076,
    "totalDocsExamined": 20076,
    "executionStages": {
        "stage": "SORT",
        "nReturned": 9903,
        "executionTimeMillisEstimate": 248,
        "works": 29982,
        "advanced": 9903,
        "needTime": 20078,
        "needYield": 0,
        "saveState": 242,
        "restoreState": 242,
        "isEOF": 1,
        "invalidates": 0,
        "sortPattern": {
            "student_id": 1
        },
        "memUsage": 2386623,
        "memLimit": 33554432,
        "inputStage": {
```

```
"stage": "SORT_KEY_GENERATOR",
"nReturned": 9903,
"executionTimeMillisEstimate": 203,
"works": 20078,
"advanced": 9903,
"needTime": 10174,
"needYield": 0,
"saveState": 242,
"restoreState": 242,
"isEOF": 1,
"invalidates": 0,
"inputStage": {
    "stage": "FETCH",
    "filter": {
        "student_id": {
            "$gt": 500000
        }
    },
    "nReturned": 9903,
    "executionTimeMillisEstimate": 192,
    "works": 20077,
    "advanced": 9903,
    "needTime": 10173,
    "needYield": 0,
    "saveState": 242,
    "restoreState": 242,
    "isEOF": 1,
    "invalidates": 0,
    "docsExamined": 20076,
    "alreadyHasObj": 0,
    "inputStage": {
        "stage": "IXSCAN",
        "nReturned": 20076,
        "executionTimeMillisEstimate": 45,
        "works": 20077,
        "advanced": 20076,
        "needTime": 0,
        "needYield": 0,
        "saveState": 242,
        "restoreState": 242,
        "isEOF": 1,
        "invalidates": 0,
        "keyPattern": {
            "class_id": 1
        },
```

```
                        "indexName": "class_id_1",
                        "isMultiKey": false,
                        "multiKeyPaths": {
                            "class_id": []
                        },
                        "isUnique": false,
                        "isSparse": false,
                        "isPartial": false,
                        "indexVersion": 2,
                        "direction": "forward",
                        "indexBounds": {
                            "class_id": [
                                "[54.0, 54.0]"
                            ]
                        },
                        "keysExamined": 20076,
                        "seeks": 1,
                        "dupsTested": 0,
                        "dupsDropped": 0,
                        "seenInvalidated": 0
                    }
                }
            }
        }
    },
    "serverInfo": {
        "host": "SGB-MBP.local",
        "port": 27017,
        "version": "3.4.1",
        "gitVersion": "5e103c4f5583e2566a45d740225dc250baacfbd7"
    },
    "ok": 1
}
```

Однако то, что мы действительно хотим видеть, – это `"nReturned"`, очень близкое к `"totalKeysExamined"`. Кроме того, мы хотели бы избежать использования метода `hint`, чтобы выполнить данный запрос более эффективно. Способ решения обеих этих проблем состоит в том, чтобы разработать более подходящий индекс.

Более подходящим индексом для рассматриваемого шаблона запроса является индекс на базе `"class_id"` и `"student_id"`, в указанном порядке. Используя `"class_id"` в качестве префикса, мы используем фильтр равенства в нашем запросе, чтобы ограничить ключи, рассматриваемые в индексе. Это наиболее избирательный компонент нашего запроса, поэтому он эффективно ограничивает количество ключей, которые MongoDB должна учиты-

вать для удовлетворения этого запроса. Можно построить данный индекс следующим образом:

```
> db.students.createIndex({class_id:1, student_id:1})
```

Хотя этого нельзя сказать абсолютно обо всех наборах данных, в целом нужно разрабатывать составные индексы таким образом, чтобы поля, для которых вы будете использовать фильтры равенства, предшествовали тем, для которых ваше приложение будет использовать многозначные фильтры.

Теперь когда у нас есть новый индекс, если мы повторно запустим свой запрос, на этот раз не нужно использовать метод `hint`, и в поле `"executionStats"` вывода `explain` видно, что у нас есть быстрый запрос (37 миллисекунд), для которого число возвращенных результатов (`"nReturned"`) равно количеству ключей, отсканированных в индексе (`"totalKeysExamined"`). Также видно, что это связано с тем, что `"executionStages"`, которые отражают выигрышный план выполнения запроса, содержат сканирование индекса, которое использует созданный нами новый индекс:

```
...
"executionStats": {
    "executionSuccess": true,
    "nReturned": 9903,
    "executionTimeMillis": 37,
    "totalKeysExamined": 9903,
    "totalDocsExamined": 9903,
    "executionStages": {
        "stage": "FETCH",
        "nReturned": 9903,
        "executionTimeMillisEstimate": 36,
        "works": 9904,
        "advanced": 9903,
        "needTime": 0,
        "needYield": 0,
        "saveState": 81,
        "restoreState": 81,
        "isEOF": 1,
        "invalidates": 0,
        "docsExamined": 9903,
        "alreadyHasObj": 0,
        "inputStage": {
            "stage": "IXSCAN",
            "nReturned": 9903,
            "executionTimeMillisEstimate": 0,
            "works": 9904,
```

```
                "advanced": 9903,
                "needTime": 0,
                "needYield": 0,
                "saveState": 81,
                "restoreState": 81,
                "isEOF": 1,
                "invalidates": 0,
                "keyPattern": {
                    "class_id": 1,
                    "student_id": 1
                },
                "indexName": "class_id_1_student_id_1",
                "isMultiKey": false,
                "multiKeyPaths": {
                    "class_id": [],
                    "student_id": []
                },
                "isUnique": false,
                "isSparse": false,
                "isPartial": false,
                "indexVersion": 2,
                "direction": "forward",
                "indexBounds": {
                    "class_id": [
                        "[54.0, 54.0]"
                    ],
                    "student_id": [
                        "(500000.0, inf.0]"
                    ]
                },
                "keysExamined": 9903,
                "seeks": 1,
                "dupsTested": 0,
                "dupsDropped": 0,
                "seenInvalidated": 0
            }
        }
    },
```

Учитывая то, что мы знаем о создании индексов, вы, вероятно, сможете понять, почему это работает. Индекс `[class_id, student_id]` состоит из пар ключей, как те, что приведены ниже. Поскольку идентификаторы студентов в этих парах идут по порядку, чтобы выполнить нашу сортировку, MongoDB просто нужно пройти все пары ключей, начиная с первой для `class_id 54`:

```
...
[53, 999617]
[53, 999780]
[53, 999916]
[54, 500001]
[54, 500009]
[54, 500048]
...
```

При рассмотрении структуры составного индекса нам нужно знать, как обращаться с фильтрами равенства, многозначными фильтрами и сортировать компоненты общих шаблонов запросов, которые будут использовать индекс. Необходимо учитывать эти три фактора для всех составных индексов, и если вы разрабатываете свой индекс, чтобы правильно сбалансировать эти вещи, то получите максимальную производительность от MongoDB для своих запросов. Несмотря на то что мы рассмотрели все три фактора для нашего примера запроса с использованием индекса `[class_id, student_id]`, написанный запрос представляет собой частный случай проблемы, связанной с составными индексами, поскольку мы выполняем сортировку по одному из полей, по которому мы также выполняем фильтрацию.

Чтобы устранить особый характер этого примера, давайте вместо этого отсортируем итоговую оценку, изменив наш запрос:

```
> db.students.find({student_id:{$gt:500000}, class_id:54})
...           .sort({final_grade:1})
...           .explain("executionStats")
```

Если мы выполним этот запрос и посмотрим на вывод `explain`, то увидим, что теперь мы выполняем сортировку в памяти. Хотя запрос по-прежнему быстрый – всего 136 миллисекунд, – он на порядок медленнее, чем при сортировке по `"student_id"`, потому что сейчас мы выполняем сортировку в памяти. Видно, что мы делаем это, поскольку план выполнения запроса теперь содержит этап `"SORT"`:

```
...
"executionStats": {
    "executionSuccess": true,
    "nReturned": 9903,
    "executionTimeMillis": 136,
    "totalKeysExamined": 9903,
    "totalDocsExamined": 9903,
    "executionStages": {
        "stage": "SORT",
        "nReturned": 9903,
```

```
"executionTimeMillisEstimate": 36,
"works": 19809,
"advanced": 9903,
"needTime": 9905,
"needYield": 0,
"saveState": 315,
"restoreState": 315,
"isEOF": 1,
"invalidates": 0,
"sortPattern": {
    "final_grade": 1
},
"memUsage": 2386623,
"memLimit": 33554432,
"inputStage": {
    "stage": "SORT_KEY_GENERATOR",
    "nReturned": 9903,
    "executionTimeMillisEstimate": 24,
    "works": 9905,
    "advanced": 9903,
    "needTime": 1,
    "needYield": 0,
    "saveState": 315,
    "restoreState": 315,
    "isEOF": 1,
    "invalidates": 0,
    "inputStage": {
        "stage": "FETCH",
        "nReturned": 9903,
        "executionTimeMillisEstimate": 24,
        "works": 9904,
        "advanced": 9903,
        "needTime": 0,
        "needYield": 0,
        "saveState": 315,
        "restoreState": 315,
        "isEOF": 1,
        "invalidates": 0,
        "docsExamined": 9903,
        "alreadyHasObj": 0,
        "inputStage": {
            "stage": "IXSCAN",
            "nReturned": 9903,
            "executionTimeMillisEstimate": 12,
            "works": 9904,
```

```
                    "advanced": 9903,
                    "needTime": 0,
                    "needYield": 0,
                    "saveState": 315,
                    "restoreState": 315,
                    "isEOF": 1,
                    "invalidates": 0,
                    "keyPattern": {
                        "class_id": 1,
                        "student_id": 1
                    },
                    "indexName": "class_id_1_student_id_1",
                    "isMultiKey": false,
                    "multiKeyPaths": {
                        "class_id": [],
                        "student_id": []
                    },
                    "isUnique": false,
                    "isSparse": false,
                    "isPartial": false,
                    "indexVersion": 2,
                    "direction": "forward",
                    "indexBounds": {
                        "class_id": [
                            "[54.0, 54.0]"
                        ],
                        "student_id": [
                            "(500000.0, inf.0]"
                        ]
                    },
                    "keysExamined": 9903,
                    "seeks": 1,
                    "dupsTested": 0,
                    "dupsDropped": 0,
                    "seenInvalidated": 0
                }
            }
        }
    }
},
...
```

Если можно избежать сортировки в памяти, используя более подходящий индекс, так и стоит поступить. Это позволит нам более легко выполнять масштабирование относительно размера набора данных и загрузки системы.

Но для этого придется пойти на компромисс, что обычно имеет место при разработке составных индексов.

Как это часто бывает необходимо при работе с составными индексами, чтобы избежать сортировки в памяти, нужно проверить больше ключей, чем количество документов, которые мы возвращаем. Чтобы использовать индекс для сортировки, MongoDB должен иметь возможность обходить ключи индекса по порядку. Это означает, что нам нужно включить поле сортировки в ключи составных индексов.

Ключи в нашем новом составном индексе должны быть упорядочены следующим образом: `[class_id, final_grade, student_id]`. Обратите внимание, что мы включаем компонент сортировки сразу после фильтра равенства, но перед многозначным фильтром. Этот индекс очень выборочно сузит набор ключей, рассматриваемых для данного запроса. Затем, пройдя триплеты ключей, совпадающие с фильтром равенства в этом индексе, MongoDB может идентифицировать записи, которые соответствуют многозначному фильтру, и эти записи будут упорядочены должным образом по итоговой оценке в порядке возрастания.

Этот составной индекс заставляет MongoDB проверять ключи на наличие большего количества документов, чем то, что окажется в нашем наборе результатов. Однако, используя индекс для обеспечения сортировки документов, мы экономим время выполнения. Можно построить новый индекс, используя следующую команду:

```
> db.students.createIndex({class_id:1, final_grade:1, student_id:1})
```

Теперь, если мы снова отправим наш запрос:

```
> db.students.find({student_id:{$gt:500000}, class_id:54})
...           .sort({final_grade:1})
...           .explain("executionStats")
```

то получим результаты, приведенные ниже. Это будет зависеть от вашего оборудования и от того, что еще происходит в системе, но видно, что выигрышный план больше не включает в себя сортировку в памяти. Вместо этого он использует индекс, который мы только что создали, чтобы выполнить запрос, включая сортировку:

```
"executionStats": {
    "executionSuccess": true,
    "nReturned": 9903,
    "executionTimeMillis": 42,
    "totalKeysExamined": 9905,
    "totalDocsExamined": 9903,
    "executionStages": {
        "stage": "FETCH",
```

```
"nReturned": 9903,
"executionTimeMillisEstimate": 34,
"works": 9905,
"advanced": 9903,
"needTime": 1,
"needYield": 0,
"saveState": 82,
"restoreState": 82,
"isEOF": 1,
"invalidates": 0,
"docsExamined": 9903,
"alreadyHasObj": 0,
"inputStage": {
    "stage": "IXSCAN",
    "nReturned": 9903,
    "executionTimeMillisEstimate": 24,
    "works": 9905,
    "advanced": 9903,
    "needTime": 1,
    "needYield": 0,
    "saveState": 82,
    "restoreState": 82,
    "isEOF": 1,
    "invalidates": 0,
    "keyPattern": {
        "class_id": 1,
        "final_grade": 1,
        "student_id": 1
    },
    "indexName": "class_id_1_final_grade_1_student_id_1",
    "isMultiKey": false,
    "multiKeyPaths": {
        "class_id": [],
        "final_grade": [],
        "student_id": []
    },
    "isUnique": false,
    "isSparse": false,
    "isPartial": false,
    "indexVersion": 2,
    "direction": "forward",
    "indexBounds": {
        "class_id": [
            "[54.0, 54.0]"
        ],
```

```
                "final_grade": [
                    "[MinKey, MaxKey]"
                ],
                "student_id": [
                    "(500000.0, inf.0]"
                ]
            },
            "keysExamined": 9905,
            "seeks": 2,
            "dupsTested": 0,
            "dupsDropped": 0,
            "seenInvalidated": 0
        }
    }
},
```

В этом разделе мы привели конкретный пример некоторых передовых методов создания составных индексов. Хотя эти рекомендации применимы не для каждой ситуации, они подходят для большинства из них и должны рассматриваться в первую очередь при построении составного индекса.

Напомним, что при разработке составного индекса:

- ключи для фильтров равенства должны появляться первыми;
- ключи, используемые для сортировки, должны появляться перед многозначными полями;
- ключи для многозначных фильтров должны появляться последними.

Разработайте свой составной индекс, используя эти рекомендации, а затем протестируйте его в реальных рабочих нагрузках для диапазона шаблонов запросов, поддерживаемых вашим индексом.

Выбор направлений ключей

Пока что все наши индексные записи отсортированы в порядке возрастания или от наименьшего к наибольшему. Однако если вам нужно выполнить сортировку по двум (или более) критериям, вам может потребоваться, чтобы индексные ключи шли в разных направлениях. Например, возвращаясь к нашему более раннему примеру с коллекцией *users*, предположим, что нам нужно было отсортировать коллекцию по возрасту от самых молодых до самых старых и по имени от Z–A. Наши предыдущие индексы были бы не очень эффективными при решении этой проблемы: в каждой возрастной группе пользователи сортировались по имени пользователя в порядке возрастания (A–Z, а не Z–A). Составные индексы, которые мы

использовали до сих пор, не содержат значений в каком-либо полезном порядке, чтобы возраст (`"age"`) шел по возрастанию, а имя пользователя (`"username"`) шло бы в порядке убывания.

Чтобы оптимизировать составные сортировки в разных направлениях, нужно использовать индекс с соответствующими направлениями. В этом примере мы могли бы использовать `{"age" : 1, "username" : -1}`, что позволило бы организовать данные следующим образом:

```
[21, user999600] -> 8765277104
[21, user999407] -> 8765252400
[21, user999390] -> 8765250224
...
[21, user100270] -> 8623545776
[21, user100266] -> 8623545264
[21, user100154] -> 8623530928
...
[30, user100168] -> 8623532720
[30, user100155] -> 8623531056
[30, user100098] -> 8623523760
```

Возрасты располагаются от самого младшего к самому старшему, и в каждом возрасте имена пользователей рассортированы от Z до A (или, скорее, от 9 до 0, учитывая наши имена пользователей).

Если нашему приложению также понадобилось бы оптимизировать сортировку по `{"age" : 1, "username" : 1}`, нам пришлось бы создать второй индекс с этими направлениями. Чтобы выяснить, какие направления использовать для индекса, просто сопоставьте направления, используемые вашей сортировкой. Обратите внимание, что обратные индексы (если умножить каждое направление на –1) равнозначны: `{"age" : 1, "username" : -1}` удовлетворяет тем же запросам, что и `{"age" : -1, "username" : 1}`.

Направление индекса действительно имеет значение только тогда, когда вы выполняете сортировку по нескольким критериям. Если вы сортируете только по одному ключу, MongoDB может так же легко прочитать индекс в обратном порядке. Например, если у вас была сортировка по `{"age" : -1}` и индекс `{"age" : 1}`, MongoDB мог бы оптимизировать ее так же, как если бы у вас был индекс `{"age" : -1 }` (поэтому не создавайте оба!). Направление имеет значение только для сортировок со множеством ключей.

Использование покрытых запросов

В предыдущих примерах индекс всегда использовался для поиска правильного документа, а затем следовал назад за указателем для получения фактического документа. Однако если ваш запрос ищет только те поля, которые включены в индекс, ему не нужно извлекать документ. Когда индекс содержит все значения, запрошенные запросом, запрос считается *покры-*

тым. По возможности используйте покрытые запросы, вместо того чтобы возвращаться к документам. Таким образом, вы можете сделать свой рабочий набор намного меньше.

Чтобы запрос мог использовать только индекс, вы должны использовать проекции (ограничивающие возвращаемые поля лишь теми, что указаны в вашем запросе; *см*. раздел «Указываем, какие ключи нужно вернуть»), чтобы избежать возврата поля `"_id"` (только если это не часть индекса). Вам также может понадобиться проиндексировать поля, по которым вы не делаете запрос, поэтому вам следует сбалансировать свою потребность в более быстрых запросах и издержки, которые добавятся при записи.

Если вы запускаете `explain` для покрытого запроса, результат будет содержать этап `"IXSCAN"`, который не является потомком этапа `"FETCH"`, а в `"executionStats"` значение `"totalDocsExamined"` будет равно 0.

Неявные индексы

Составные индексы могут выполнять «двойную обязанность» и действовать как разные индексы для разных запросов. Если у нас есть индекс `{"age" : 1, "username" : 1}`, поле `"age"` сортируется идентично тому, как если бы у нас был только индекс `{"age" : 1}`. Таким образом, составной индекс можно использовать как сам по себе индекс `{"age" : 1}`.

Это можно распространить на то количество ключей, какое вам необходимо: у индекса *N* ключей, вы получаете «свободный» индекс для любого префикса этих ключей. Например, если у нас есть индекс, который выглядит как `{"a": 1, "b": 1, "c": 1, ..., "z": 1}`, у нас фактически есть индексы для `{"a": 1}`, `{"a": 1, "b": 1}`, `{"a": 1, "b": 1, "c": 1}` и т. д.

Обратите внимание, что это не относится к *любому* подмножеству ключей: запросы, которые будут использовать индекс `{"b": 1}` или `{"a": 1, "c": 1}` (например), не будут оптимизированы. Этим могут воспользоваться только запросы, которые могут использовать префикс индекса.

Как операторы с символом $ используют индексы

Некоторые запросы могут использовать индексы более эффективно, чем другие; иные запросы и вовсе не могут использовать их. В этом разделе описывается, как MongoDB обрабатывает различные операторы запросов.

Неэффективные операторы

В общем, отрицание неэффективно. Запросы с оператором `"$ne"` могут использовать индекс, но делают они это не очень хорошо. Они должны просматривать все записи индекса, кроме той, которая указана `"$ne"`, поэтому в основном им приходится сканировать весь индекс. Например, в случае с коллекцией с индексом по полю `"i"` диапазоны индекса, пройденные для такого запроса, выглядят так:

```
db.example.find({
    "i": {
        "$ne": 3
    }
}).explain()
{
    "queryPlanner": {
...,
        "parsedQuery": {
            "i": {
                "$ne": "3"
            }
        },
        "winningPlan": {
            {
...,
                "indexBounds": {
                    "i": [
                        [
                            {
                                "$minElement": 1
                            },
                            3
                        ],
                        [
                            3,
                            {
                                "$maxElement": 1
                            }
                        ]
                    ]
                }
            }
        },
        "rejectedPlans": []
    },
    "serverInfo": {
...,
    }
}
```

Этот запрос просматривает все записи индекса, которые меньше 3, и все записи индекса, которые больше 3. Это может быть эффективно, если большой набор вашей коллекции равен 3, но в противном случае нужно проверять почти все.

Оператор `"$not"` иногда может использовать индекс, но часто не знает, как. Он может инвертировать базовые диапазоны (`{"ключ" : {"$lt" : 7}}` превращается в `{"ключ" : {"$gte" : 7}}`) и регулярные выражения. Тем не менее большинство других запросов с `"$not"` вернутся к сканированию таблицы. Оператор `"$nin"` всегда использует сканирование таблицы.

Если вам нужно быстро выполнить один из этих типов запросов, выясните, есть ли другой оператор, который можно добавить к запросу, в котором можно использовать индекс для фильтрации набора результатов по небольшому количеству документов, прежде чем MongoDB попытается выполнить неиндексированное сопоставление.

Диапазоны

Составные индексы могут помочь MongoDB эффективно выполнять запросы с несколькими операторами. При разработке индекса с несколькими полями сначала поместите поля, которые будут использоваться в точных совпадениях (например, `"x" : 1`), а диапазоны – последними (например, `"y": {"$gt" : 3, "$lt" : 5}`). Это позволяет запросу найти точное значение первого индексного ключа, а затем выполнить поиск по нему для второго индексного диапазона. Например, предположим, что мы запрашивали определенный возраст и диапазон имен пользователей, используя индекс `{"age" : 1, "username" : 1}`. Мы получили бы довольно точные границы индекса:

```
> db.users.find({
    "age": 47,
    "username":
    ... {
        "$gt": "user5",
        "$lt": "user8"
    }
}).explain('executionStats')
    {
    "queryPlanner": {
        "plannerVersion": 1,
        "namespace": "test.users",
        "indexFilterSet": false,
        "parsedQuery": {
            "$and": [
                {
                    "age": {
                        "$eq": 47
                    }
                },
                {
```

```
                "username": {
                    "$lt": "user8"
                }
            },
            {
                "username": {
                    "$gt": "user5"
                }
            }
        ]
    },
    "winningPlan": {
        "stage": "FETCH",
        "inputStage": {
            "stage": "IXSCAN",
            "keyPattern": {
                "age": 1,
                "username": 1
            },
            "indexName": "age_1_username_1",
            "isMultiKey": false,
            "multiKeyPaths": {
                "age": [],
                "username": []
            },
            "isUnique": false,
            "isSparse": false,
            "isPartial": false,
            "indexVersion": 2,
            "direction": "forward",
            "indexBounds": {
                "age": [
                    "[47.0, 47.0]"
                ],
                "username": [
                    "(\"user5\", \"user8\")"
                ]
            }
        }
    },
    "rejectedPlans": [
        {
            "stage": "FETCH",
            "filter": {
                "age": {
```

```
                    "$eq": 47
                }
            },
            "inputStage": {
                "stage": "IXSCAN",
                "keyPattern": {
                    "username": 1
                },
                "indexName": "username_1",
                "isMultiKey": false,
                "multiKeyPaths": {
                    "username": []
                },
                "isUnique": false,
                "isSparse": false,
                "isPartial": false,
                "indexVersion": 2,
                "direction": "forward",
                "indexBounds": {
                    "username": [
                        "(\"user5\", \"user8\")"
                    ]
                }
            }
        }
    ]
},
"executionStats": {
    "executionSuccess": true,
    "nReturned": 2742,
    "executionTimeMillis": 5,
    "totalKeysExamined": 2742,
    "totalDocsExamined": 2742,
    "executionStages": {
        "stage": "FETCH",
        "nReturned": 2742,
        "executionTimeMillisEstimate": 0,
        "works": 2743,
        "advanced": 2742,
        "needTime": 0,
        "needYield": 0,
        "saveState": 23,
        "restoreState": 23,
        "isEOF": 1,
        "invalidates": 0,
```

```
            "docsExamined": 2742,
            "alreadyHasObj": 0,
            "inputStage": {
                "stage": "IXSCAN",
                "nReturned": 2742,
                "executionTimeMillisEstimate": 0,
                "works": 2743,
                "advanced": 2742,
                "needTime": 0,
                "needYield": 0,
                "saveState": 23,
                "restoreState": 23,
                "isEOF": 1,
                "invalidates": 0,
                "keyPattern": {
                    "age": 1,
                    "username": 1
                },
                "indexName": "age_1_username_1",
                "isMultiKey": false,
                "multiKeyPaths": {
                    "age": [],
                    "username": []
                },
                "isUnique": false,
                "isSparse": false,
                "isPartial": false,
                "indexVersion": 2,
                "direction": "forward",
                "indexBounds": {
                    "age": [
                        "[47.0, 47.0]"
                    ],
                    "username": [
                        "(\"user5\", \"user8\")"
                    ]
                },
                "keysExamined": 2742,
                "seeks": 1,
                "dupsTested": 0,
                "dupsDropped": 0,
                "seenInvalidated": 0
            }
        }
    },
```

```
    "serverInfo": {
        "host": "eoinbrazil-laptop-osx",
        "port": 27017,
        "version": "4.0.12",
        "gitVersion": "5776e3cbf9e7afe86e6b29e22520ffb6766e95d4"
    },
    "ok": 1
}
```

Запрос переходит непосредственно к `"age" : 47`, а затем ищет в нем имена пользователей в промежутке между `"user5"` и `"user8"`.

И наоборот, предположим, что мы используем индекс `{"username" : 1, "age" : 1}`. Это меняет план запроса, поскольку запрос должен смотреть на всех пользователей в промежутке между `"user5"` и `"user8"` и выбирать тех, чей возраст равен 47 (`"age" : 47`):

```
> db.users.find({
    "age": 47,
    "username": {
        "$gt": "user5",
        "$lt": "user8"
    }
})
.explain('executionStats')
{
    "queryPlanner": {
        "plannerVersion": 1,
        "namespace": "test.users",
        "indexFilterSet": false,
        "parsedQuery": {
            "$and": [
                {
                    "age": {
                        "$eq": 47
                    }
                },
                {
                    "username": {
                        "$lt": "user8"
                    }
                },
                {
                    "username": {
                        "$gt": "user5"
                    }
```

```
            }
        ]
    },
    "winningPlan": {
        "stage": "FETCH",
        "filter": {
            "age": {
                "$eq": 47
            }
        },
        "inputStage": {
            "stage": "IXSCAN",
            "keyPattern": {
                "username": 1
            },
            "indexName": "username_1",
            "isMultiKey": false,
            "multiKeyPaths": {
                "username": []
            },
            "isUnique": false,
            "isSparse": false,
            "isPartial": false,
            "indexVersion": 2,
            "direction": "forward",
            "indexBounds": {
                "username": [
                    "(\"user5\", \"user8\")"
                ]
            }
        }
    },
    "rejectedPlans": [
        {
            "stage": "FETCH",
            "inputStage": {
                "stage": "IXSCAN",
                "keyPattern": {
                    "username": 1,
                    "age": 1
                },
                "indexName": "username_1_age_1",
                "isMultiKey": false,
                "multiKeyPaths": {
                    "username": [],
```

```
                    "age": []
                },
                "isUnique": false,
                "isSparse": false,
                "isPartial": false,
                "indexVersion": 2,
                "direction": "forward",
                "indexBounds": {
                    "username": [
                        "(\"user5\", \"user8\")"
                    ],
                    "age": [
                        "[47.0, 47.0]"
                    ]
                }
            }
        }
    ]
},
"executionStats": {
    "executionSuccess": true,
    "nReturned": 2742,
    "executionTimeMillis": 369,
    "totalKeysExamined": 333332,
    "totalDocsExamined": 333332,
    "executionStages": {
        "stage": "FETCH",
        "filter": {
            "age": {
                "$eq": 47
            }
        },
        "nReturned": 2742,
        "executionTimeMillisEstimate": 312,
        "works": 333333,
        "advanced": 2742,
        "needTime": 330590,
        "needYield": 0,
        "saveState": 2697,
        "restoreState": 2697,
        "isEOF": 1,
        "invalidates": 0,
        "docsExamined": 333332,
        "alreadyHasObj": 0,
        "inputStage": {
```

```
                "stage": "IXSCAN",
                "nReturned": 333332,
                "executionTimeMillisEstimate": 117,
                "works": 333333,
                "advanced": 333332,
                "needTime": 0,
                "needYield": 0,
                "saveState": 2697,
                "restoreState": 2697,
                "isEOF": 1,
                "invalidates": 0,
                "keyPattern": {
                    "username": 1
                },
                "indexName": "username_1",
                "isMultiKey": false,
                "multiKeyPaths": {
                    "username": []
                },
                "isUnique": false,
                "isSparse": false,
                "isPartial": false,
                "indexVersion": 2,
                "direction": "forward",
                "indexBounds": {
                    "username": [
                        "(\"user5\", \"user8\")"
                    ]
                },
                "keysExamined": 333332,
                "seeks": 1,
                "dupsTested": 0,
                "dupsDropped": 0,
                "seenInvalidated": 0
            }
        }
    },
    "serverInfo": {
        "host": "eoinbrazil-laptop-osx",
        "port": 27017,
        "version": "4.0.12",
        "gitVersion": "5776e3cbf9e7afe86e6b29e22520ffb6766e95d4"
    },
    "ok": 1
}
```

Это заставляет MongoDB сканировать в 100 раз больше записей индекса, чем при использовании предыдущего индекса. Использование двух диапазонов в запросе в основном всегда приводит к появлению менее эффективного плана запроса такого рода.

Оператор $or (запросы, содержащие оператор OR)

На момент написания этого раздела MongoDB может использовать только по одному индексу на запрос. То есть если вы создадите один индекс `{"x": 1}` и еще один индекс `{"y": 1}`, а затем выполните запрос для `{"x": 123, "y": 456}`, MongoDB будет использовать один из созданных вами индексов, но не оба. Единственное исключение из этого правила – оператор `"$or"`. `"$or"` может использовать по одному индексу на каждый оператор, поскольку он выполняет два запроса, а затем объединяет результаты:

```
db.foo.find({
    "$or": [
        {
            "x": 123
        },
        {
            "y": 456
        }
    ]
}).explain()
{
    "queryPlanner": {
        "plannerVersion": 1,
        "namespace": "foo.foo",
        "indexFilterSet": false,
        "parsedQuery": {
            "$or": [
                {
                    "x": {
                        "$eq": 123
                    }
                },
                {
                    "y": {
                        "$eq": 456
                    }
                }
            ]
        },
        "winningPlan": {
```

```
"stage": "SUBPLAN",
"inputStage": {
    "stage": "FETCH",
    "inputStage": {
        "stage": "OR",
        "inputStages": [
            {
                "stage": "IXSCAN",
                "keyPattern": {
                    "x": 1
                },
                "indexName": "x_1",
                "isMultiKey": false,
                "multiKeyPaths": {
                    "x": []
                },
                "isUnique": false,
                "isSparse": false,
                "isPartial": false,
                "indexVersion": 2,
                "direction": "forward",
                "indexBounds": {
                    "x": [
                        "[123.0, 123.0]"
                    ]
                }
            },
            {
                "stage": "IXSCAN",
                "keyPattern": {
                    "y": 1
                },
                "indexName": "y_1",
                "isMultiKey": false,
                "multiKeyPaths": {
                    "y": []
                },
                "isUnique": false,
                "isSparse": false,
                "isPartial": false,
                "indexVersion": 2,
                "direction": "forward",
                "indexBounds": {
                    "y": [
                        "[456.0, 456.0]"
```

```
                                ]
                            }
                        }
                    ]
                }
            }
        },
        "rejectedPlans": []
    },
    "serverInfo": {
...,
    },
    "ok": 1
}
```

Как видите, здесь нам потребовалось два отдельных запроса по двум индексам (как обозначено двумя этапами `"IXSCAN"`). В целом делать два запроса и объединять результаты гораздо менее эффективно, чем выполнять один запрос; таким образом, по возможности, лучше используйте оператор `"$in"` вместо `"$or"`.

Если вам необходимо использовать оператор `"$or"`, помните, что MongoDB нужно просмотреть результаты обоих запросов и удалить все дубликаты (документы, которые соответствуют более чем одному оператору `"$or"`).

При выполнении запросов с `"$in"` другого способа, кроме сортировки, нет, чтобы контролировать порядок возврата документов. Например, `{"x": {"$in": [1, 2, 3]}}` вернет документы в том же порядке, что и `{"x": {"$in": [3, 2, 1]}}`.

Индексирование объектов и массивов

MongoDB позволяет вам проникать в ваши документы и создавать индексы для вложенных полей и массивов. Поля вложенного объекта и массива можно сочетать с полями верхнего уровня в составных индексах, и хотя они в некотором смысле являются особыми, в основном они ведут себя так же, как «обычные» индексные поля.

Индексирование вложенных документов

Индексы можно создавать для ключей во вложенных документах так же, как их создают для обычных ключей. Если бы у нас была коллекция, в которой каждый документ обозначает пользователя, у нас мог бы быть вложенный документ, описывающий местоположение каждого пользователя:

```
{
    "username": "sid",
```

```
    "loc": {
        "ip": "1.2.3.4",
        "city": "Springfield",
        "state": "NY"
    }
}
```

Мы могли бы поместить индекс в одно из вложенных полей `"loc"`, скажем `"loc.city"`, чтобы ускорить запросы, использующие это поле:

```
> db.users.createIndex({"loc.city" : 1})
```

Вы можете зайти настолько глубоко, насколько захотите: вы можете проиндексировать `"x.y.z.w.a.b.c"` (и так далее), если хотите.

Обратите внимание, что индексирование самого вложенного документа (`"loc"`) ведет себя совершенно иначе, чем индексирование поля этого документа (`"loc.city"`). Индексирование вложенного документа целиком поможет только запросам, которые запрашивают весь вложенный документ. Оптимизатор запросов может использовать индекс `"loc"` только для запросов, описывающих весь вложенный документ с полями в правильном порядке (например, `db.users.find({"loc": {"ip": "123.456.789.000", "city": "Shelbyville", "state": "NY"}})`). Он не мог бы использовать индекс для запросов типа `db.users.find({"loc.city" : "Shelbyville"})`.

Индексирование массивов

Вы также можете индексировать массивы, что позволяет использовать индекс для эффективного поиска определенных элементов массива.

Предположим, у нас есть коллекция постов в блоге, где каждый документ – это пост. У каждого поста есть поле `"comments"`, которое представляет собой массив вложенных документов `"comment"`. Если бы нам понадобилось найти самые последние прокомментированные посты в блоге, мы могли бы создать индекс по ключу `"date"` в массиве вложенных документов `"comments"` нашей коллекции постов:

```
> db.blog.createIndex({"comments.date" : 1})
```

В ходе индексирования массива для каждого элемента массива создается индексная запись, поэтому, если у поста 20 комментариев, он будет иметь 20 индексных записей. Это делает индексы массива более затратными, по сравнению с индексами, имеющими одно значение: при одной вставке, обновлении или удалении может потребоваться обновление каждой записи массива (возможно, тысяч индексных записей).

В отличие от примера с `"loc"` из предыдущего раздела, нельзя проиндексировать весь массив как единую сущность: при индексировании поля массива индексируется каждый элемент массива, а не сам массив.

Индексы элементов массива не сохраняют никакого понятия расположения: нельзя использовать индекс для запроса, который ищет определенный элемент массива, например `"comments.4"`.

Кстати, вы можете проиндексировать конкретную запись массива, например:

```
> db.blog.createIndex({"comments.10.votes": 1})
```

Однако этот индекс будет полезен только для запросов точно по 11-му элементу массива (первый элемент массива имеет нулевой индекс).

Только одно поле в индексной записи может быть из массива. Это сделано для того, чтобы избежать огромного количества записей в индексе, которые вы получаете из нескольких индексов с большим количеством ключей: каждая возможная пара элементов должна быть проиндексирована, в результате чего индексы будут составлять $n*m$ записей на документ. Например, предположим, что у нас был индекс `{"x" : 1, "y" : 1}`:

```
> // x - это массив - допустимо;
> db.multi.insert({"x" : [1, 2, 3], "y" : 1})
>
> // y - это массив - также допустимо;
> db.multi.insert({"x" : 1, "y" : [4, 5, 6]})
>
> // x и y - это массивы - недопустимо!
> db.multi.insert({"x" : [1, 2, 3], "y" : [4, 5, 6]})
cannot index parallel arrays [y] [x]
```

Если бы MongoDB индексировала последний пример, ей пришлось бы создать индексные записи для `{"x": 1, "y": 4}`, `{"x": 1, "y": 5}`, `{"x": 1, "y": 6}`, `{"x": 2, " y ": 4}`, `{"x": 2, "y": 5}`, `{"x": 2, "y": 6}`, `{"x": 3, "y": 4}`, `{"x": 3, "y": 5}` и `{"x": 3, "y": 6}` (а длина этих массивов составляет всего три элемента).

Последствия многоключевых (мультиключевых) индексов

Если в каком-либо документе есть поле массива для индексированного ключа, индекс немедленно помечается как многоключевой (мультиключевой). Можно увидеть, является ли индекс многоключевым (мультиключевым), из вывода `explain`: если использовался многоключевой (мультиключевой) индекс, поле `"isMultikey"` будет иметь значение `true`. После того как индекс был помечен как многоключевой, он ни при каких обстоятельствах не может лишиться этого свойства, даже если все документы, содержащие массивы в этом поле, удалены. Единственный способ отменить это свойство – сбросить индекс и создать снова.

Многоключевые (мультиключевые) индексы могут быть немного медленнее. Многие индексные записи могут указывать на один документ, по-

этому MongoDB может потребоваться выполнить дедупликацию, прежде чем вернуть результаты.

Кардинальность индекса

Под *кардинальностью* понимается количество различных значений для поля в коллекции. Некоторые поля, такие как `"gender"` или `"newsletter opt-out"`, могут иметь только два возможных значения, что считается очень низкой кардинальностью. Другие поля, такие как `"username"` или `"email"`, могут иметь уникальное значение для каждого документа в коллекции, что является высокой кардинальностью. Третьи находятся где-то посередине, например `"age"` или `"zip code"`.

В целом чем больше кардинальность поля, тем полезнее может быть индекс для этого поля. Это объясняется тем, что индекс может быстро сузить пространство поиска до гораздо меньшего набора результатов. Для поля с низкой кардинальностью индекс, как правило, не может исключить как можно больше совпадений.

Например, предположим, что у нас был индекс по полю `"gender"`, и мы искали женщин по имени Сьюзен. Мы могли бы сузить пространство результатов примерно лишь на 50 %, прежде чем обращаться к отдельным документам для поиска «имени». И наоборот, если бы мы выполняли индексацию по «имени», мы могли бы сразу же сузить наш набор результатов до крошечной доли пользователей по имени Сьюзен, а затем обратиться к этим документам, чтобы проверить пол.

В качестве практического правила попробуйте создать индексы для ключей с высокой кардинальностью или, по крайней мере, сначала поместить ключи с высокой кардинальностью в составные индексы (перед ключами с низкой кардинальностью).

Вывод explain

Как вы уже видели, `explain` дает вам большое количество информации о ваших запросах. Это один из самых важных инструментов диагностики для медленных запросов. Вы можете узнать, какие индексы используются и как, посмотрев на вывод `explain`. Для любого запроса вы можете добавить вызов `explain` в конце (так же, как и в случае с функциями `sort` или `limit`, но вызов `explain` должен идти последним).

Существует два типа вывода `explain`, которые вы будете встречать чаще всего: для индексированных запросов и неиндексированных. Специальные типы индексов могут создавать несколько разные планы запросов, но большинство полей должны быть похожими. Кроме того, в процессе шардинга возвращается скопление `explain` (как описано в главе 14), поскольку запрос выполняется на нескольких серверах.

Самый простой тип `explain` – для запроса, который не использует индекс. Можно сказать, что запрос не использует индекс, потому что он использует `"COLLSCAN"`.

Вывод `explain` по запросу, использующему индекс, различается, но в самом простом случае это выглядит примерно так, если мы добавим индекс `imdb.rating`:

```
> db.users.find({
    "age": 42
}).explain('executionStats')
{
    "queryPlanner": {
        "plannerVersion": 1,
        "namespace": "test.users",
        "indexFilterSet": false,
        "parsedQuery": {
            "age": {
                "$eq": 42
            }
        },
        "winningPlan": {
            "stage": "FETCH",
            "inputStage": {
                "stage": "IXSCAN",
                "keyPattern": {
                    "age": 1,
                    "username": 1
                },
                "indexName": "age_1_username_1",
                "isMultiKey": false,
                "multiKeyPaths": {
                    "age": [],
                    "username": []
                },
                "isUnique": false,
                "isSparse": false,
                "isPartial": false,
                "indexVersion": 2,
                "direction": "forward",
                "indexBounds": {
                    "age": [
                        "[42.0, 42.0]"
                    ],
                    "username": [
                        "[MinKey, MaxKey]"
```

```
                    ]
                }
            }
        },
        "rejectedPlans": []
    },
    "executionStats": {
        "executionSuccess": true,
        "nReturned": 8449,
        "executionTimeMillis": 15,
        "totalKeysExamined": 8449,
        "totalDocsExamined": 8449,
        "executionStages": {
            "stage": "FETCH",
            "nReturned": 8449,
            "executionTimeMillisEstimate": 10,
            "works": 8450,
            "advanced": 8449,
            "needTime": 0,
            "needYield": 0,
            "saveState": 66,
            "restoreState": 66,
            "isEOF": 1,
            "invalidates": 0,
            "docsExamined": 8449,
            "alreadyHasObj": 0,
            "inputStage": {
                "stage": "IXSCAN",
                "nReturned": 8449,
                "executionTimeMillisEstimate": 0,
                "works": 8450,
                "advanced": 8449,
                "needTime": 0,
                "needYield": 0,
                "saveState": 66,
                "restoreState": 66,
                "isEOF": 1,
                "invalidates": 0,
                "keyPattern": {
                    "age": 1,
                    "username": 1
                },
                "indexName": "age_1_username_1",
                "isMultiKey": false,
                "multiKeyPaths": {
```

```
                    "age": [],
                    "username": []
                },
                "isUnique": false,
                "isSparse": false,
                "isPartial": false,
                "indexVersion": 2,
                "direction": "forward",
                "indexBounds": {
                    "age": [
                        "[42.0, 42.0]"
                    ],
                    "username": [
                        "[MinKey, MaxKey]"
                    ]
                },
                "keysExamined": 8449,
                "seeks": 1,
                "dupsTested": 0,
                "dupsDropped": 0,
                "seenInvalidated": 0
            }
        }
    },
    "serverInfo": {
        "host": "eoinbrazil-laptop-osx",
        "port": 27017,
        "version": "4.0.12",
        "gitVersion": "5776e3cbf9e7afe86e6b29e22520ffb6766e95d4"
    },
    "ok": 1
}
```

Этот вывод сначала сообщает вам, какой индекс использовался: `imdb.rating`. Далее указано, сколько документов фактически было возвращено в результате: `"nReturned"`. Обратите внимание, что это не обязательно отражает объем работы MongoDB по ответу на запрос (т. е. сколько индексов и документов ей пришлось искать). `"totalKeysExamined"` сообщает о количестве просканированных записей индекса, а `"totalDocsExamined"` указывает, сколько документов было отсканировано. Количество отсканированных документов отображено в `"nscannedObjects"`.

Вывод также показывает, что `rejectedPlans` не было и что он использовал ограниченный поиск по индексу в пределах значения 42.0.

`"executionTimeMillis"` сообщает, насколько быстро был выполнен запрос, с момента, когда сервер получил запрос, до момента отправки ответа. Однако

это не всегда то число, которое вы ищете. Если MongoDB опробовала несколько планов запросов, `"executionTimeMillis"` будет показывать, сколько времени потребовалось для их выполнения, а не тот, что был выбран лучшим.

Теперь, когда вы знакомы с основами, ниже приводится более подробное описание некоторых наиболее важных полей:

`"isMultiKey" : false`
: Если в этом запросе использовался многоключевой индекс (*см.* раздел «Индексирование объектов и массивов»).

`"nReturned": 8449`
: Количество документов, возвращаемых по запросу.

`"totalDocsExamined": 8449`
: Сколько раз MongoDB должна была следовать указателю индекса на фактический документ на диске. Если запрос содержит критерии, которые не являются частью индекса, или поля запросов, которые не содержатся в индексе, MongoDB должна найти документ, на который указывает каждая индексная запись.

`"totalKeysExamined": 8449`
: Количество просмотренных индексных записей, если индекс использовался. Если это было сканирование таблицы, это количество проверенных документов.

`"stage": "IXSCAN"`
: Была ли MongoDB в состоянии выполнить этот запрос, используя индекс; в противном случае `"COLSCAN"` будет означать, что для выполнения запроса необходимо выполнить сканирование коллекции.

  В этом примере MongoDB обнаружила все соответствующие документы, используя индекс, который мы знаем, потому что `"totalKeysExamined"` – то же самое, что и `"totalDocsExamined"`. Однако в запросе было указано, что необходимо возвращать все поля в соответствующих документах, а индекс содержал только поля `"age"` или `"username"`.

`"needYield": 0`
: Сколько раз этот запрос уступал (делал паузу), чтобы разрешить выполнение запроса на запись. Если есть записи, ожидающие отправки, запросы будут периодически снимать блокировку и позволять им действовать дальше. В этой системе ничего подобного не было, поэтому запрос не уступил ни разу.

`"executionTimeMillis": 15`
: Количество миллисекунд, которое потребовалось базе данных для выполнения запроса. Чем ниже это число, тем лучше.

`"indexBounds": {...}`
: Описание того, как использовался индекс, с указанием диапазонов пройденного индекса. В этом примере, поскольку первое предложение в запросе было точным соответствием, индексу нужно было только посмотреть на это значение: 42. Второй ключ индекса был свободной переменной, потому что в запросе не было указано никаких ограничений. Таким образом, база данных искала значения между отрицательной бесконечностью (`"$minElement": 1`) и бесконечностью (`"$maxElement": 1`) для имен пользователей в пределах `"age": 42`.

Посмотрим на немного более сложный пример. Предположим, у вас есть индексы `{"username": 1, "age": 1}` и `{"age": 1, "username": 1}`. Что произойдет, если вы запросите `"username"` и `"age"`? Зависит от запроса:

```
"queryPlanner": {
    "plannerVersion": 1,
    "namespace": "test.users",
    "indexFilterSet": false,
    "parsedQuery": {
        "$and": [
            {
                "username": {
                    "$eq": "user2134"
                }
            },
            {
                "age": {
                    "$gt": 10
                }
            }
        ]
    },
    "winningPlan": {
        "stage": "FETCH",
        "filter": {
            "age": {
                "$gt": 10
            }
        },
        "inputStage": {
            "stage": "IXSCAN",
            "keyPattern": {
                "username": 1
            },
            "indexName": "username_1",
```

```
            "isMultiKey": false,
            "multiKeyPaths": {
                "username": []
            },
            "isUnique": false,
            "isSparse": false,
            "isPartial": false,
            "indexVersion": 2,
            "direction": "forward",
            "indexBounds": {
                "username": [
                    "[\"user2134\", \"user2134\"]"
                ]
            }
        }
    },
    "rejectedPlans": [
        {
            "stage": "FETCH",
            "inputStage": {
                "stage": "IXSCAN",
                "keyPattern": {
                    "age": 1,
                    "username": 1
                },
                "indexName": "age_1_username_1",
                "isMultiKey": false,
                "multiKeyPaths": {
                    "age": [],
                    "username": []
                },
                "isUnique": false,
                "isSparse": false,
                "isPartial": false,
                "indexVersion": 2,
                "direction": "forward",
                "indexBounds": {
                    "age": [
                        "(10.0, inf.0]"
                    ],
                    "username": [
                        "[\"user2134\", \"user2134\"]"
                    ]
                }
            }
```

```
        }
    ]
},
"serverInfo": {
    "host": "eoinbrazil-laptop-osx",
    "port": 27017,
    "version": "4.0.12",
    "gitVersion": "5776e3cbf9e7afe86e6b29e22520ffb6766e95d4"
},
"ok": 1
}
```

Мы запрашиваем точное совпадение для `"username"` и диапазона значений для `"age"`, поэтому база данных решает использовать индекс `{"username" : 1, "age" : 1}`, полностью изменяя условия запроса. Если, с другой стороны, мы запрашиваем точный возраст и диапазон имен, MongoDB будет использовать другой индекс:

```
> db.users.find({
    "age": 14,
    "username": /.*/
}).explain()
{
    "queryPlanner": {
        "plannerVersion": 1,
        "namespace": "test.users",
        "indexFilterSet": false,
        "parsedQuery": {
            "$and": [
                {
                    "age": {
                        "$eq": 14
                    }
                },
                {
                    "username": {
                        "$regex": ".*"
                    }
                }
            ]
        },
        "winningPlan": {
            "stage": "FETCH",
            "inputStage": {
```

```
                "stage": "IXSCAN",
                "filter": {
                    "username": {
                        "$regex": ".*"
                    }
                },
                "keyPattern": {
                    "age": 1,
                    "username": 1
                },
                "indexName": "age_1_username_1",
                "isMultiKey": false,
                "multiKeyPaths": {
                    "age": [],
                    "username": []
                },
                "isUnique": false,
                "isSparse": false,
                "isPartial": false,
                "indexVersion": 2,
                "direction": "forward",
                "indexBounds": {
                    "age": [
                        "[14.0, 14.0]"
                    ],
                    "username": [
                        "[\"\", {})",
                        "[/.*/, /.*/]"
                    ]
                }
            }
        },
        "rejectedPlans": [
            {
                "stage": "FETCH",
                "filter": {
                    "age": {
                        "$eq": 14
                    }
                },
                "inputStage": {
                    "stage": "IXSCAN",
                    "filter": {
```

```
                        "username": {
                            "$regex": ".*"
                        }
                    },
                    "keyPattern": {
                        "username": 1
                    },
                    "indexName": "username_1",
                    "isMultiKey": false,
                    "multiKeyPaths": {
                        "username": []
                    },
                    "isUnique": false,
                    "isSparse": false,
                    "isPartial": false,
                    "indexVersion": 2,
                    "direction": "forward",
                    "indexBounds": {
                        "username": [
                            "[\"\", {})",
                            "[/.*/, /.*/]"
                        ]
                    }
                }
            }
        ]
    },
    "serverInfo": {
        "host": "eoinbrazil-laptop-osx",
        "port": 27017,
        "version": "4.0.12",
        "gitVersion": "5776e3cbf9e7afe86e6b29e22520ffb6766e95d4"
    },
    "ok": 1
}
```


Если вы обнаружите, что Mongo использует для запроса другие индексы, а не те, что вы хотите, можно заставить ее использовать определенный индекс с помощью метода `hint`. Например, если вы хотите убедиться, что MongoDB использует индекс `{"username": 1, "age": 1}` в предыдущем запросе, можно написать следующее:

```
> db.users.find({"age": 14, "username": /.*/}).hint({"username": 1, "age": 1})
```

Если запрос не использует нужный вам индекс и вы используете метод `hint`, чтобы изменить его, запустите `explain` перед развертыванием. Если вы заставите MongoDB использовать индекс для запроса, для которого она не знает, как использовать индекс, в итоге ваш запрос может получиться менее эффективным, чем без индекса.

Когда не стоит прибегать к индексированию

Индексы наиболее эффективны при извлечении небольших подмножеств данных, а некоторые типы запросов быстрее работают без индексов. Индексы становятся все менее и менее эффективными, по мере того как вам нужно получать бóльший процент коллекции, потому что использование индекса требует выполнять поиск дважды: один раз для просмотра индексной записи, и второй раз – следуя за указателем индекса на документ. Для сканирования коллекции требуется только одно: просмотр документа. В худшем случае (возврат всех документов в коллекции) использование индекса заняло бы вдвое больше поисков и, как правило, было бы значительно медленнее, чем сканирование коллекции.

К сожалению, не существует строгого правила относительно того, когда индекс помогает, а когда мешает, поскольку в действительности это зависит от размера ваших данных, индексов, документов и среднего набора результатов (табл. 5.1). Как правило, индекс часто ускоряет работу, если запрос возвращает 30 % коллекции или более. Однако это число может варьироваться от 2 % до 60 %. В табл. 5.1 собраны условия, при которых сканирования индексов или коллекций, как правило, работают лучше.

Таблица 5.1. Свойства, влияющие на эффективность индексов

Индексы часто хорошо подходят для	Сканирование коллекции часто хорошо подходит для
Больших коллекций	Маленьких коллекций
Больших документов	Маленьких документов
Выборочных запросов	Неселективных запросов

Допустим, у нас есть информационно-аналитическая система, которая собирает статистику. Наше приложение запрашивает у системы все документы для данной учетной записи, чтобы сгенерировать хороший график всех данных за час до начала времени:

```
> db.entries.find({"created_at" : {"$lt" : hourAgo}})
```

Мы индексируем `"created_at"`, чтобы ускорить этот запрос.

При первом запуске набор результатов очень маленький, и запрос возвращается мгновенно. Но через пару недель данных становится много, и через месяц этот запрос уже занимает слишком много времени для выполнения.

Для большинства приложений это, вероятно, «неправильный» запрос: вам действительно нужен запрос, который возвращает бóльшую часть вашего набора данных? Большинству приложений, особенно с большими наборами данных, это не нужно. Однако в некоторых допустимых случаях вам может потребоваться бóльшая их часть или все данные. Например, вы можете экспортировать эти данные в систему отчетов или использовать их для пакетного задания. В этих случаях вам бы хотелось вернуть большую часть набора данных как можно быстрее.

Типы индексов

Существует несколько параметров, которые можно указать при создании индекса, изменяющего его поведение. Наиболее распространенные варианты описаны в последующих разделах, а более сложные или особые варианты – в следующей главе.

Уникальные индексы

Уникальные индексы гарантируют, что каждое значение будет отображаться в индексе не более одного раза. Например, если вы хотите убедиться в том, что два документа не могут иметь одинаковое значение в ключе `"username"`, то можете создать уникальный индекс с помощью параметра `partialFilterExpression` только для документов с полем `firstname` (подробнее об этом будет рассказано позже в этой главе):

```
> db.users.createIndex({"firstname": 1},
... {"unique": true,"partialFilterExpression": {
        "firstname": {$exists: true } } } )
{
    "createdCollectionAutomatically": false,
    "numIndexesBefore": 3,
    "numIndexesAfter": 4,
    "ok": 1
}
```

Например, предположим, что вы попытались вставить приведенные ниже документы в коллекцию *users*:

```
> db.users.insert({firstname: "bob"})
WriteResult({ "nInserted": 1 })
```

```
> db.users.insert({firstname: "bob"})
WriteResult({
    "nInserted": 0,
    "writeError": {
        "code": 11000,
        "errmsg": "E11000 duplicate key error collection: test.users index:
                  firstname_1 dup key: {
            : \"bob\" }"
        }
    })
```

Если вы проверите коллекцию, то увидите, что был сохранен только первый вариант с именем `"bob"`. Создание исключений дубликатов ключей не очень эффективно, поэтому используйте уникальное ограничение для случайного дубликата, чтобы не отфильтровывать по миллиону дубликатов в секунду.

Уникальный индекс, с которым вы, вероятно, уже знакомы, – это индекс `"_id"`, который создается автоматически при создании коллекции. Это обычный уникальный индекс (помимо того факта, что его нельзя удалить, как другие уникальные индексы).

Если ключа не существует, индекс сохраняет его значение как `null` для этого документа. Это означает, что если вы создаете уникальный индекс и пытаетесь вставить более одного документа, в котором отсутствует индексированное поле, вставки завершатся неудачно, поскольку у вас уже есть документ со значением `null`. *См.* раздел «Частичные индексы», где приводится совет, как с этим справиться.

В некоторых случаях значение не будет проиндексировано. Индексные сегменты имеют ограниченный размер, и если индексная запись превышает его, он просто не будет включен в индекс. Это может вызвать путаницу, поскольку это делает документ «невидимым» для запросов, использующих индекс. До появления MongoDB версии 4.2 поле должно было быть меньше 1024 байтов, чтобы его можно было включить в индекс. В MongoDB версии 4.2 и более поздних версиях это ограничение было снято. MongoDB не возвращает никаких ошибок или предупреждений, если поля документа нельзя проиндексировать из-за размера. Это означает, что ключи длиной более 8 КВ не будут подвергаться ограничениям уникального индекса: например, можно вставить одинаковые строки размером 8 КБ.

Составные уникальные индексы

Вы также можете создать составной уникальный индекс. Если вы сделаете это, отдельные ключи могут иметь одинаковые значения, но комбинация значений для всех ключей в индексной записи может появляться в индексе не более одного раза.

Например, если бы у нас был уникальный индекс `{"username" : 1, "age" : 1}`, приведенные ниже вставки были бы допустимы:

```
> db.users.insert({"username" : "bob"})
> db.users.insert({"username" : "bob", "age" : 23})
> db.users.insert({"username" : "fred", "age" : 23})
```

Однако попытка вставить вторую копию любого из этих документов приведет к исключению дубликата ключа.

GridFS, стандартный метод хранения больших файлов в MongoDB (*см.* раздел «Хранение файлов с помощью GridFS»), использует составной уникальный индекс. Коллекция, которая содержит содержимое файла, имеет уникальный индекс `{"files_id" : 1, "n" : 1}`, который позволяет документам выглядеть (частично) так, как показано ниже:

```
{"files_id" : ObjectId("4b23c3ca7525f35f94b60a2d"), "n" : 1}
{"files_id" : ObjectId("4b23c3ca7525f35f94b60a2d"), "n" : 2}
{"files_id" : ObjectId("4b23c3ca7525f35f94b60a2d"), "n" : 3}
{"files_id" : ObjectId("4b23c3ca7525f35f94b60a2d"), "n" : 4}
```

Обратите внимание, что все значения `"files_id"` одинаковы, но `"n"` разные.

Удаление дубликатов

Если вы попытаетесь создать уникальный индекс для существующей коллекции, он не будет создан, если имеются какие-либо повторяющиеся значения:

```
> db.users.createIndex({"age" : 1}, {"unique" : true})
WriteResult({
    "nInserted": 0,
    "writeError": {
        "code": 11000,
        "errmsg": "E11000 duplicate key error collection:
                   test.users index: age_1 dup key: {: 12 }"
    }
})
```

Как правило, вам придется обработать свои данные (может помочь фреймворк агрегации) и выяснить, где находятся дубликаты и что с ними делать.

Частичные индексы

Как уже упоминалось в предыдущем разделе, уникальные индексы считают `null` значением, поэтому у вас не может быть уникального индекса с несколькими документами, в которых отсутствует ключ. Однако во многих случаях вам может понадобиться, чтобы уникальный индекс применялся только при наличии ключа. Если у вас есть поле, которое может или не может существовать, но должно быть уникальным, когда оно существует, можно сочетать параметры «`unique`" и `"partial"`.

Частичные индексы в MongoDB создаются только для подмножества данных. Это не похоже на разреженные индексы в реляционных СУБД, которые создают меньше индексных записей, указывающих на блок данных, однако все блоки данных будут иметь ассоциированную запись разреженного индекса в реляционной системе управления базами данных.

Чтобы создать частичный индекс, используйте параметр `"partalFilterExpression"`. Частичные индексы представляют собой расширенный набор функций, предлагаемых разреженными индексами, с документом, представляющим выражение фильтра, для которого вы хотите его создать. Например, если указание адреса электронной почты не является обязательным, но, если это предусмотрено, должно быть уникальным, можно сделать следующее:

```
> db.users.ensureIndex({"email" : 1}, {"unique" : true, "partialFilterExpression" :
... { email: { $exists: true } }})
```

Частичные индексы не обязательно должны быть уникальными. Чтобы создать неуникальный частичный индекс, просто не используйте параметр `"unique"`.

Следует помнить, что один и тот же запрос может возвращать разные результаты в зависимости от того, использует он частичный индекс или нет. Например, предположим, что у нас есть коллекция, в которой у большинства документов есть поле `"x"`, а у одного – нет:

```
> db.foo.find()
{ "_id" : 0 }
{ "_id" : 1, "x" : 1 }
{ "_id" : 2, "x" : 2 }
{ "_id" : 3, "x" : 3 }
```

Когда мы сделаем запрос для `"x"`, он вернет все соответствующие документы:

```
> db.foo.find({"x" : {"$ne" : 2}})
{ "_id" : 0 }
{ "_id" : 1, "x" : 1 }
{ "_id" : 3, "x" : 3 }
```

Если мы создадим частичный индекс `"x"`, документ `"_id" : 0` не будет включен в индекс. Так что теперь, если мы запросим `"x"`, MongoDB будет использовать индекс и не вернет документ `{"_id": 0}`:

```
> db.foo.find({"x" : {"$ne" : 2}})
{ "_id" : 1, "x" : 1 }
{ "_id" : 3, "x" : 3 }
```

Можно использовать метод `hint`, чтобы заставить его выполнить сканирование таблицы, если вам нужны документы с пропущенными полями.

Управление индексами

Как было показано в предыдущем разделе, можно создавать новые индексы, используя функцию `createIndex`. Индекс нужно создавать только один раз для каждой коллекции. Если вы попытаетесь создать тот же индекс снова, ничего не произойдет.

Вся информация об индексах базы данных хранится в коллекции *system.indexes*. Это зарезервированная коллекция, поэтому нельзя изменять ее документы или удалять их оттуда. Вы можете манипулировать ей только с помощью команд `createIndex`, `createIndexes` и `dropIndexes`.

При создании индекса можно увидеть его метаинформацию в *system.indexes*. Вы также можете выполнить команду `db.`*`Имяколлекции`*`.getIndexes()`, чтобы увидеть информацию обо всех индексах для данной коллекции:

```
> db.students.getIndexes()
[
    {
        "v": 2,
        "key": {
            "_id": 1
        },
        "name": "_id_",
        "ns": "school.students"
    },
    {
        "v": 2,
        "key": {
            "class_id": 1
        },
```

```
        "name": "class_id_1",
        "ns": "school.students"
    },
    {
        "v": 2,
        "key": {
            "student_id": 1,
            "class_id": 1
        },
        "name": "student_id_1_class_id_1",
        "ns": "school.students"
    }
]
```

Важными полями являются `"key"` и `"name"`. Ключ можно использовать для подсказок и других мест, где должен быть указан индекс. Здесь важен порядок полей: индекс `{"class_id": 1, "student_id": 1}` не совпадает с индексом `{"student_id": 1, "class_id": 1}`. Имя индекса используется в качестве идентификатора для множества операций административного индекса, таких как `dropIndexes`. Является ли индекс мультиключевым, в его спецификации не указано.

Поле `"v"` используется внутри для управления версиями индекса. Если у вас есть индексы, у которых нет хотя бы поля `"v": 1`, они хранятся в более старом, менее эффективном формате. Можно обновить их, убедившись, что вы работаете с MongoDB по крайней мере версии 2.0, а также удаляя и заново создавая индексы.

Идентификация индексов

У каждого индекса в коллекции есть имя, которое уникальным образом идентифицирует этот индекс и используется сервером для удаления или манипулирования им. Имена индекса по умолчанию имеют вид *имяключа 1_dir1_имяключа 2_dir2_..._имяключа N_dirN*, где *имяключа X* – это ключ индекса, а *dirX* – его направление (1 или –1). Оно может выглядеть громоздко, если индексы содержат более пары ключей, поэтому можно указать свое имя в качестве одного из параметров функции `createIndex`:

```
> db.soup.createIndex({"a" : 1, "b" : 1, "c" : 1, ..., "z" : 1},
... {"name" : "alphabet"})
```

Количество символов в имени индекса ограничено, поэтому для создания сложных индексов могут потребоваться настраиваемые имена. Вызов `getLastError` покажет, удалось ли создать индекс или почему это не удалось.

Замена индексов

По мере роста и изменения вашего приложения вы можете обнаружить, что ваши данные или запросы изменились и что индексы, которые раньше работали хорошо, теперь не работают. Ненужные индексы можно удалить с помощью команды `dropIndex`:

```
> db.people.dropIndex("x_1_y_1")
{ "nIndexesWas" : 3, "ok" : 1 }
```

Используйте поле `"name"` из описания индекса, чтобы указать, какой индекс удалить.

Создание новых индексов отнимает много времени и ресурсов. До появления версии 4.2 MongoDB создавала индекс как можно быстрее, блокируя все операции чтения и записи в базе данных до завершения построения индекса. Если вы хотите, чтобы ваша база данных по-прежнему чутко реагировала на чтение и запись, используйте опцию `"background"` при создании индекса. Это вынуждает сборку индекса иногда уступать другим операциям, но все же может оказывать серьезное влияние на ваше приложение (*см.* раздел «Создание индексов» для получения дополнительной информации). Индексация в фоновом режиме также намного медленнее по сравнению с индексацией переднего плана. В версии 4.2 появился новый подход – гибридная сборка индекса. Он использует эксклюзивную блокировку только в начале и в конце сборки индекса. Оставшаяся часть процесса сборки приводит к чередованию операций чтения и записи, что выступает в качестве замены создания индексов в фоновом режиме и в режиме foreground в MongoDB версии 4.2.

Если у вас есть выбор, создание индексов для существующих документов выполняется немного быстрее, чем первоначальное создание индекса, а затем вставка всех документов.

Подробнее об эксплуатационных аспектах построения индексов читайте в главе 19.

Глава 6

Специальные типы индексов и коллекций

В этой главе рассматриваются специальные коллекции и типы индексов, которые есть в MongoDB, в том числе:

- ограниченные коллекции для данных, напоминающих очереди;
- индексы TTL для кешей;
- полнотекстовые индексы для простого поиска строк;
- геопространственные индексы для двухмерной и сферической геометрии;
- GridFS для хранения больших файлов.

Геопространственные индексы

В MongoDB существует два типа геопространственных индексов: `2dsphere` и `2d`. Индексы `2dsphere` работают с элементами сферической геометрии, которые моделируют поверхность Земли на основе данных WGS84. Эти данные моделируют поверхность Земли в виде сплюснутого сфероида, что означает некоторое выравнивание на полюсах. Таким образом, при расчете расстояния с использованием индексов `2dsphere` принимается во внимание форма Земли и обеспечивается более точная обработка расстояния, например между двумя городами по сравнению с индексами `2d`. Используйте индексы `2d` для точек, находящихся на двумерной плоскости.

`2dsphere` позволяет задавать элементы для точек, линий и полигонов в формате GeoJSON (http://www.geojson.org/). Точка задается двухэлементным массивом, обозначающим [*долготу*, *широту*]:

```
{
    "name": "New York City",
    "loc": {
        "type": "Point",
        "coordinates": [ 50, 2]
```

```
    }
}
```

Линия задается массивом точек:

```
{
    "name": "Hudson River",
    "loc": {
        "type": "LineString",
        "coordinates": [ [ 0 , 1 ], [ 0, 2 ], [ 1, 2 ] ]
    }
}
```

Полигон задается так же, как и линия (массив точек), но с другим «типом»:

```
{
    "name": "New England",
    "loc": {
        "type": "Polygon",
        "coordinates" : [[0,1], [0,2], [1,2]]
    }
}
```

Поле, которое мы называем в этом примере `"loc"`, может называться как угодно, но имена полей во вложенном объекте определяются форматом GeoJSON, и их нельзя изменить.

Геопространственный индекс можно создать, используя тип `"2dsphere"` с помощью функции `createIndex`:

```
> db.openStreetMap.createIndex({"loc" : "2dsphere"})
```

Чтобы создать индекс `2dsphere`, передайте документ функции `createIndex`, в котором указано поле, содержащее элементы, которые вы хотите проиндексировать для рассматриваемой коллекции, и укажите `"2dsphere"` в качестве значения.

Типы геопространственных запросов

Существует три типа геопространственных запросов, которые можно выполнять: пересечение, в пределах и близость. Вы указываете то, что ищете, как объект GeoJSON, который выглядит как `{"$ geometry": geoJsonDesc}`.

Например, можно найти документы, которые пересекают местоположение запроса, используя оператор `"$geoIntersects"`:

```
> var eastVillage = {
... "type" : "Polygon",
... "coordinates" : [
```

```
...  [
...    [ -73.9732566, 40.7187272 ],
...    [ -73.9724573, 40.7217745 ],
...    [ -73.9717144, 40.7250025 ],
...    [ -73.9714435, 40.7266002 ],
...    [ -73.975735, 40.7284702 ],
...    [ -73.9803565, 40.7304255 ],
...    [ -73.9825505, 40.7313605 ],
...    [ -73.9887732, 40.7339641 ],
...    [ -73.9907554, 40.7348137 ],
...    [ -73.9914581, 40.7317345 ],
...    [ -73.9919248, 40.7311674 ],
...    [ -73.9904979, 40.7305556 ],
...    [ -73.9907017, 40.7298849 ],
...    [ -73.9908171, 40.7297751 ],
...    [ -73.9911416, 40.7286592 ],
...    [ -73.9911943, 40.728492 ],
...    [ -73.9914313, 40.7277405 ],
...    [ -73.9914635, 40.7275759 ],
...    [ -73.9916003, 40.7271124 ],
...    [ -73.9915386, 40.727088 ],
...    [ -73.991788, 40.7263908 ],
...    [ -73.9920616, 40.7256489 ],
...    [ -73.9923298, 40.7248907 ],
...    [ -73.9925954, 40.7241427 ],
...    [ -73.9863029, 40.7222237 ],
...    [ -73.9787659, 40.719947 ],
...    [ -73.9772317, 40.7193229 ],
...    [ -73.9750886, 40.7188838 ],
...    [ -73.9732566, 40.7187272 ]
...  ]
... ]}
> db.openStreetMap.find(
... {"loc" : {"$geoIntersects" : {"$geometry" : eastVillage}}})
```

Это позволит найти все документы, содержащие точки, линии и полигоны, у которых есть точка в районе Ист-Виллидж в Нью-Йорке.

Можно использовать оператор `"$geoWithin"`, чтобы запрашивать объекты, которые целиком находятся в одном районе (например, "Какие рестораны есть в Ист-Виллидже?"):

```
> db.openStreetMap.find({"loc" : {"$geoWithin" : {"$geometry" : eastVillage}}})
```

В отличие от нашего первого запроса, он не будет возвращать объекты, которые просто проходят через Ист-Виллидж (например, улицы) или частично перекрывают его (например, полигон, описывающий Манхэттен).

Наконец, можно запросить близлежащее местоположение с помощью оператора `"$near"`:

```
> db.openStreetMap.find({"loc" : {"$near" : {"$geometry" : eastVillage}}})
```

Обратите внимание на то, что `"$near"` – единственный геопространственный оператор, который подразумевает сортировку: результаты всегда возвращаются в порядке расстояния от ближайшего к дальнему.

Использование геопространственных индексов

Геопространственная индексация в MongoDB позволяет эффективно выполнять пространственные запросы к коллекции, которая содержит геопространственные формы и точки. Чтобы продемонстрировать возможности геопространственных функций и сравнить различные подходы, мы пройдем процесс написания запросов для простого геопространственного приложения. Мы немного подробнее рассмотрим несколько концепций, имеющих ключевое значение для геопространственных индексов, а затем продемонстрируем их применение с помощью операторов `"$geoWithin"`, `"$geoIntersects"` и `"$geoNear"`.

Предположим, что мы разрабатываем мобильное приложение, которое должно помочь пользователям находить рестораны в Нью-Йорке. Приложение должно:

- определить район, в котором находится пользователь;
- показать количество ресторанов в этом районе;
- найти рестораны в пределах указанного расстояния.

Мы будем использовать индекс `2dsphere` для запроса этих данных.

Двухмерная геометрия и сферическая геометрия в запросах

Геопространственные запросы могут использовать либо сферическую, либо двухмерную (плоскую) геометрию, в зависимости от запроса и типа применяемого индекса. В табл. 6.1 показано, какую геометрию использует каждый геопространственный оператор.

Также обратите внимание на то, что индексы `2d` поддерживают как элементы плоской геометрии, так и расчет расстояний для сфер (то есть с использованием оператора `$nearSphere`). Однако запросы, использующие сферическую геометрию, будут более производительными и точными с индексом `2dsphere`.

Также обратите внимание на то, что оператор `$geoNear` – это оператор агрегации. Фреймворк агрегации обсуждается в главе 7. В дополнение к операции запроса с помощью `$near` оператор `$geoNear` и специальная команда `geoNear` позволяют запрашивать близлежащие местоположения.

Имейте в виду, что оператор запроса `$near` не будет работать с коллекциями, которые распространяются с использованием шардинга, способа масштабирования в MongoDB (*см.* главу 15).

Команда `geoNear` и оператор `$geoNear` требуют, чтобы у коллекции был максимум один индекс `2dsphere` и максимум один индекс `2d`, тогда как операторы геопространственных запросов (например, `$near` и `$geoWithin`) позволяют коллекциям иметь несколько геопространственных индексов.

Ограничение по геопространственным индексам для команды `geoNear` и оператора агрегации `$geoNear` существует, поскольку ни команда `geoNear`, ни синтаксис `$geoNear` не содержат поле местоположения. Таким образом, выбор индекса среди индексов `2d` или `2dsphere` неоднозначен.

Данное ограничение не применяется к операторам геопространственных запросов; эти операторы принимают поле местоположения, устраняя неоднозначность.

Таблица 6.1. Типы запросов и геометрия в MongoDB

Тип запроса	Тип геометрии
`$near` (точка GeoJSON, индекс `2dsphere`)	Сферическая
`$near` (унаследованные координаты, индекс `2d`)	Плоская
$ geoNear (точка GeoJSON, индекс `2dsphere`)	Сферическая
$ geoNear (устаревшие координаты, индекс `2d`)	Плоская
`$nearSphere` (точка GeoJSON, индекс `2dsphere`)	Сферическая
`$nearSphere` (унаследованные координаты, индекс `2d`)*	Сферическая
`$geoWithin : { $geometry: ...}`	Сферическая
`$geoWithin : { $box: ... }`	Плоская
`$geoWithin : { $polygon: ... }`	Плоская
`$geoWithin : { $center: ... }`	Плоская
`$geoWithin : { $centerSphere: ... }`	Сферическая
`$geoIntersects`	Сферическая

* Вместо этого используйте точки GeoJSON.

Искажение

Сферическая геометрия будет выглядеть искаженной при визуализации на карте по причине характера проецирования трехмерной сферы, такой как земля, на плоскую плоскость.

Возьмем, к примеру, спецификацию сферического квадрата, определяемого точками долготы и широты (0,0), (80,0), (80,80) и (0,80). На рис. 6.1 изображена область, покрытая этим регионом.

Рис. 6.1. Сферический квадрат, определенный точками (0,0), (80,0), (80, 80) и (0,80)

Поиск ресторанов

В этом примере мы будем работать с наборами данных окрестностей (https://oreil.ly/rpGna) и ресторанов (https://oreil.ly/JXYd-) Нью-Йорка. Скачать примеры наборов данных можно на сайте GitHub.

Мы можем импортировать наборы данных в нашу базу данных, используя утилиту `mongoimport`:

```
$ mongoimport <path to neighborhoods.json> -c neighborhoods
$ mongoimport <path to restaurants.json> -c restaurants
```

Можно создать индекс `2dsphere` для каждой коллекции, используя команду `createIndex` в оболочке *mongo* (https://oreil.ly/NMUhn).

```
> db.neighborhoods.createIndex({location:"2dsphere"})
> db.restaurants.createIndex({location:"2dsphere"})
```

Изучение данных

Мы можем получить представление о схеме, используемой для документов в этих коллекциях, с помощью пары быстрых запросов в оболочке *mongo*:

```
> db.neighborhoods.find({name: "Clinton"
})
{
    "_id": ObjectId("55cb9c666c522cafdb053a4b"),
    "geometry": {
        "coordinates": [
            [
                [-73.99,40.77],
                .
                .
                .
                [-73.99,40.77],
                [-73.99,40.77]
            ]
        ],
        "type": "Polygon"
    },
    "name": "Clinton"
}

> db.restaurants.find({name: "Little Pie Company"})
{
    "_id": ObjectId("55cba2476c522cafdb053dea"),
    "location": {
        "coordinates": [
            -73.99331699999999,
            40.7594404
        ],
        "type": "Point"
    },
    "name": "Little Pie Company"
}
```

Документ района из предыдущего кода соответствует области в Нью-Йорке, показанной на рис. 6.2.

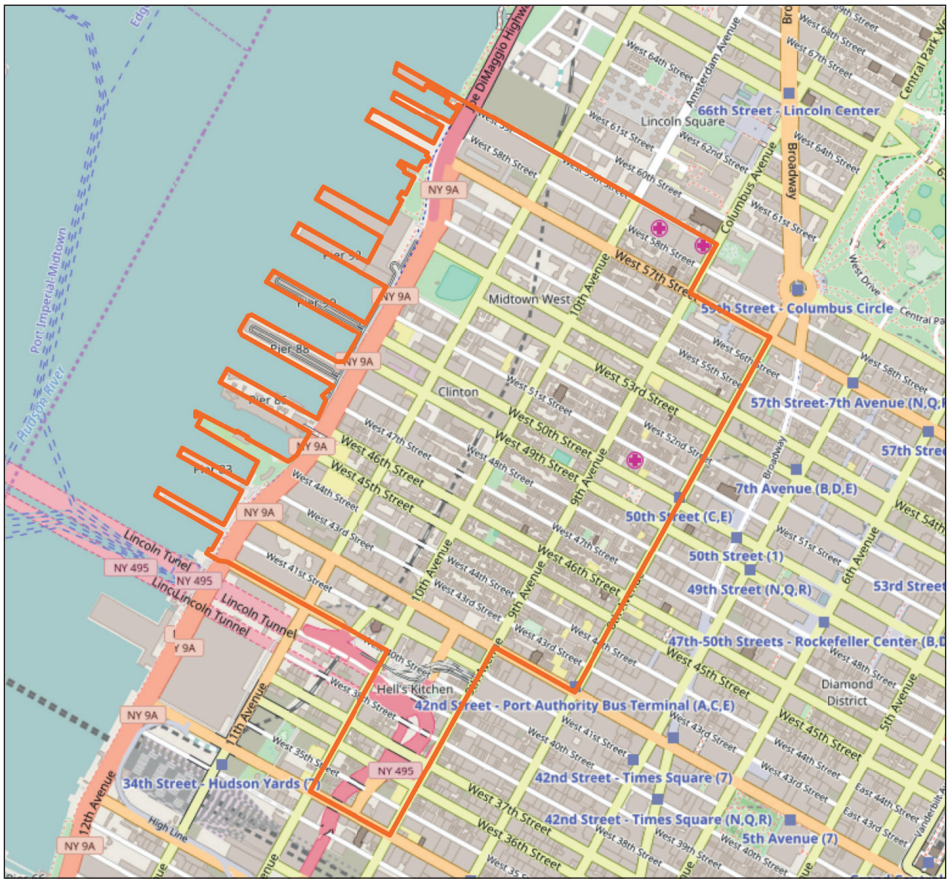

Рис. 6.2. Район Адской кухни (Клинтон) Нью-Йорка

Пекарня соответствует местоположению, показанному на рис. 6.3.

Поиск нужного района

Предполагая, что мобильное устройство пользователя может предоставить достаточно точное местоположение, можно легко найти нужный район с помощью оператора `$geoIntersects`.

Предположим, что пользователь находится в месте с координатами –73,93414657 (долгота) и 40,82302903 (широта). Чтобы найти текущий район (Hell's Kitchen), мы можем указать точку, используя специальное поле `$geometry` в формате GeoJSON:

```
> db.neighborhoods.findOne({geometry:{$geoIntersects:{$geometry:{type:"Point",
... coordinates:[-73.93414657,40.82302903]}}}})
```

Этот запрос вернет следующий результат:

```
{
    "_id":ObjectId("55cb9c666c522cafdb053a68"),
    "geometry": {
        "type": "Polygon",
        "coordinates":[[[-73.93383000695911,40.81949109558767],...]]},
        "name": "Central Harlem North-Polo Grounds"
}
```

Рис. 6.3. Компания Little Pie по адресу 424 West 43rd Street

Поиск всех ресторанов по соседству

Мы также можем сделать запрос, чтобы найти все рестораны в данном районе. Для этого можно выполнить приведенный ниже код в оболочке *mongo*, чтобы найти район, в котором находится пользователь, а затем посчитать рестораны в этом районе. Например, чтобы найти все рестораны в районе Адской кухни:

```
> var neighborhood = db.neighborhoods.findOne(
{
    geometry: {
        $geoIntersects: {
            $geometry: {
                type: "Point",
                coordinates: [-73.93414657,40.82302903]
            }
        }
    }
});
> db.restaurants.find({
    location: {
        $geoWithin: {
          // Используем геометрию из соседнего объекта, который мы получили выше;
          $geometry: neighborhood.geometry
        }
    }
  },
  // Проектируем только название ресторана, по которому есть совпадение;
  {name: 1, _id: 0});
}
```

Этот запрос сообщит вам, что в запрошенном районе есть 127 ресторанов со следующими названиями:

```
{
    "name": "White Castle"
}
{
    "name": "Touch Of Dee'S"
}
{
    "name": "Mcdonald'S"
}
{
    "name": "Popeyes Chicken & Biscuits"
}
{
    "name": "Make My Cake"
}
{
    "name": "Manna Restaurant Ii"
}
...
```

```
{
    "name": "Harlem Coral Llc"
}
```

Поиск ресторанов, находящихся на расстоянии

Чтобы найти рестораны на указанном расстоянии от точки, можно использовать операторы `"$geoWithin"` и `"$centerSphere"` для возврата результатов в несортированном порядке либо операторы `"$nearSphere"` и `"$maxDistance"`, если вам нужны результаты, отсортированные по расстоянию.

Чтобы найти рестораны в круговой области, используйте операторы `"$geoWithin"` и `"$centerSphere"`. `"$centerSphere"` – специфический для MongoDB синтаксис для обозначения круговой области путем указания центра и радиуса в радианах. Оператор `"$geoWithin"` не возвращает документы в каком-либо определенном порядке, поэтому может сначала вернуть самые дальние документы.

С помощью приведенного ниже кода мы найдем все рестораны, находящиеся в пределах пяти миль от пользователя:

```
> db.restaurants.find({
    location: {
        $geoWithin: {
            $centerSphere: [
                [-73.93414657,40.82302903],
                5/3963.2
            ]
        }
    }
})
```

Второй аргумент оператора `"$centerSphere"` принимает радиус в радианах. Запрос преобразует расстояние в радианы путем выполнения деления на приблизительный экваториальный радиус Земли, 3963,2 мили.

Приложения могут использовать `"$centerSphere"` без геопространственного индекса. Однако геопространственные индексы поддерживают гораздо более быстрые запросы по сравнению со своими неиндексированными аналогами. Геопространственные индексы `2dsphere` и `2d` поддерживают `"$centerSphere"`.

Вы также можете использовать оператор `"$nearSphere"` и указать для оператора `"$maxDistance"` значение в метрах, в результате чего получите все рестораны, находящиеся в пределах пяти миль от пользователя в отсортированном порядке от ближайшего к дальнему:

```
> var METERS_PER_MILE = 1609.34;
db.restaurants.find({
    location: {
        $nearSphere: {
            $geometry: {
                type: "Point",
                coordinates: [
                    -73.93414657,
                    40.82302903
                ]
            },
        $maxDistance: 5*METERS_PER_MILE
        }
    }
});
```

Составные геопространственные индексы

Как и в случае с другими типами индексов, можно комбинировать геопространственные индексы с другими полями для оптимизации более сложных запросов. Возможный запрос, упомянутый ранее, звучал так: «Какие рестораны есть в Адской кухне?» Используя только геопространственный индекс, мы могли бы сузить поле и искать все, что находится в этом районе, но, чтобы ограничить поиск словами «рестораны» или «пицца», потребуется еще одно поле в индексе:

```
> db.openStreetMap.createIndex({"tags" : 1, "location" : "2dsphere"})
```

Тогда мы сможем быстро найти пиццерию:

```
> db.openStreetMap.find({"loc" : {"$geoWithin" :
... {"$geometry" : hellsKitchen.geometry}},
... "tags" : "pizza"})
```

У нас может быть индексное поле «vanilla» либо до, либо после поля `"2dsphere"`, в зависимости от того, хотим ли мы сначала делать фильтрацию по полю vanilla или по местоположению. Выберите то вариант, который является более избирательным (т. е. отфильтрует больше результатов в качестве первого термина).

Индексы 2d

Для несферических карт (карт видеоигр, данных временных рядов и т. д.) можно использовать индекс `"2d"` вместо `"2dsphere"`:

```
> db.hyrule.createIndex({"tile" : "2d"})
```

Индексы 2d предполагают совершенно плоскую поверхность вместо сферы. Таким образом, их не следует использовать со сферами, только если вы не возражаете против массовых искажений вокруг полюсов.

Документы должны использовать двухэлементный массив для своего 2d-индексированного поля. Элементы в этом массиве должны отражать координаты долготы и широты соответственно. Вот как может выглядеть пример такого документа:

```
{
     "name" : "Water Temple",
     "tile" : [ 32, 22 ]
}
```

Не используйте индекс 2d, если вы планируете сохранять данные в формате GeoJSON – они могут индексировать только точки. Вы можете сохранить массив точек, но он и будет сохранен как массив точек, а не линия. В частности, это важное различие для запросов с использованием оператора "$geoWithin". Если вы сохраняете улицу как массив точек, документ будет соответствовать "$geoWithin", если одна из этих точек находится в заданной форме. Однако линия, созданная этими точками, может не полностью находиться в форме.

По умолчанию индексы 2d предполагают, что ваши значения будут находиться в диапазоне от –180 до 180. Если вы ожидаете бóльших или меньших границ, вы можете указать, какие минимальные и максимальные значения будут использованы в качестве опций для команды createIndex:

```
> db.hyrule.createIndex({"light-years" : "2d"}, {"min" : -1000, "max" : 1000})
```

В результате будет создан пространственный индекс, откалиброванный для квадрата 2000×2000.

Индексы 2d поддерживают селекторы запросов с операторами "$geoWithin", "$nearSphere" и "$near". Используйте оператор "$geoWithin" для запроса точек внутри фигуры, определенной на плоской поверхности. Этот оператор может запрашивать все точки внутри прямоугольника, многоугольника, круга или сферы; он использует оператор "$geometry" для указания объекта GeoJSON. Вернемся к нашей сетке, проиндексированной следующим образом:

```
> db.hyrule.createIndex({"tile" : "2d"})
```

Ниже приведены запросы документов в прямоугольнике, определенном координатами [10, 10] в нижнем левом углу и координатами [100, 100] в верхнем правом углу:

```
> db.hyrule.find({
    tile: {
```

```
        $geoWithin: {
            $box: [[10, 10], [100, 100]]
        }
    }
})
```

`$box` принимает массив из двух элементов: первый элемент задает координаты левого нижнего угла, а второй элемент – правого верхнего.

Чтобы запросить документы, которые находятся внутри круга с координатами [–17, 20.5] и радиусом 25, можно выполнить следующую команду:

```
> db.hyrule.find({
    tile: {
        $geoWithin: {
            $center: [[-17, 20.5] , 25]
        }
    }
})
```

Следующий запрос возвращает все документы с координатами, которые есть в полигоне с координатами [0, 0], [3, 6] и [6, 0]:

```
> db.hyrule.find({
    tile: {
        $geoWithin: {
            $polygon: [[0, 0], [3, 6], [6, 0]]
        }
    }
})
```

Вы указываете полигон как массив точек. Конечная точка в списке будет «соединена» с первой точкой, образующей полигон. В этом примере все документы, содержащие точки, будут размещены в данном треугольнике.

MongoDB также поддерживает элементарные сферические запросы для плоских индексов `2d` по причинам унаследованных координат. Как правило, в сферических вычислениях следует использовать индекс 2dsphere, как описано в разделе «Двухмерная геометрия и сферическая геометрия в запросах». Однако для запроса унаследованных пар координат в рамках сферы используйте операторы `"$geoWithin"` и `"$centerSphere"`. Укажите массив, который содержит:

- координаты сетки центральной точки круга;
- радиус круга в радианах.

Например:

```
> db.hyrule.find({
    loc: {
        $geoWithin: {
            $centerSphere: [[88, 30], 10/3963.2]
        }
    }
})
```

Для запроса ближайших точек используйте оператор `"$near"`. Запросы на близость возвращают документы с парами координат, ближайшими к определенной точке, и сортируют результаты по расстоянию. Так можно найти все документы в коллекции *hyrule* в порядке расстояния от точки (20, 21):

```
> db.hyrule.find({"tile" : {"$near" : [20, 21]}})
```

Если ограничение не указано, по умолчанию применяется ограничение в 100 документов. Если вам не нужно так много результатов, нужно установить ограничение для экономии ресурсов сервера. Например, приведенный ниже код возвращает 10 ближайших к (20, 21) документов:

```
> db.hyrule.find({"tile" : {"$near" : [20, 21]}}).limit(10)
```

Индексы для полнотекстового поиска

Текстовые индексы в MongoDB поддерживают требования полнотекстового поиска. Данный тип индекса не следует путать с индексами полнотекстового поиска MongoDB Atlas, которые используют библиотеку Apache Lucene для обеспечения дополнительных возможностей текстового поиска по сравнению с текстовыми индексами MongoDB. Используйте текстовый индекс, если ваше приложение должно позволить пользователям отправлять запросы по ключевым словам, которые должны соответствовать заголовкам, описаниям и тексту в других полях коллекции.

В предыдущих главах мы запрашивали строки, используя точные совпадения и регулярные выражения, но у этих методов есть ограничения. Поиск большого блока текста для регулярного выражения выполняется медленно, и трудно принять во внимание морфологию (например, слово «запись» должно совпадать с «записями») и другие проблемы, связанные с естественным языком. Текстовые индексы дают возможность быстрого поиска текста и обеспечивают поддержку распространенных требований поисковой системы, таких как токенизация с учетом языка, стоп-слова и стемминг.

Текстовым индексам требуется некое количество ключей, пропорциональных словам в индексируемых полях. Как следствие при создании

текстового индекса может потребляться большое количество системных ресурсов. Вы должны создавать такой индекс тогда, когда он не будет негативно влиять на производительность вашего приложения для пользователей, или строить индекс в фоновом режиме, если это возможно. Чтобы обеспечить хорошую производительность, как и для всех индексов, также следует позаботиться о том, чтобы любой созданный вами текстовый индекс помещался в ОЗУ. *См.* главу 19 для получения дополнительной информации о создании индексов с минимальным воздействием на ваше приложение.

Запись в коллекцию требует обновления всех индексов. Если вы используете текстовый поиск, строки будут разбиты на токены, и будет выполнен поиск основы слова, а индекс обновлен, возможно, во многих местах. По этой причине записи с использованием текстовых индексов обычно обходятся дороже, чем записи в индексы с одним полем, составные индексы или даже многоключевые. Таким образом, вы будете склонны видеть худшую производительность записи в коллекциях с текстовыми индексами, по сравнению с другими. Они также будут замедлять перемещение данных, если вы используете шардинг: весь текст должен быть проиндексирован заново при переносе на новые серверы, содержащие разные части данных (шарды).

Создание текстового индекса

Предположим, у нас есть коллекция статей из «Википедии», которые мы хотим проиндексировать. Чтобы выполнить поиск по тексту, сначала нужно создать текстовый индекс. В результате приведенного ниже вызова метода `createIndex` будет создан индекс на основе термов в полях `"title"` и `"body"`:

```
> db.articles.createIndex({"title": "text",
                           "body" : "text"})
```

Это не похоже на «обычный» составной индекс. По умолчанию каждому полю уделяется одинаковое внимание в текстовом индексе. Можно контролировать относительную важность, которую MongoDB придает каждому полю, указав веса:

```
> db.articles.createIndex({"title": "text",
                           "body": "text"
                          },
                          {
                           "weights": {
                           "title": 3,
                           "body": 2
                           }
})
```

Здесь мы видим соотношение весов 3:2 для полей `"title"` и `"body"`.

После создания индекса веса полей изменять нельзя (без удаления индекса и его повторного создания), поэтому вы можете поэкспериментировать с весами в образце набора данных перед созданием индекса для своих производственных данных.

В случае с некоторыми коллекциями вы можете не знать, какие поля будет содержать документ. Вы можете создать полнотекстовый индекс для всех строковых полей в документе, создав индекс `"$**"` – он не только индексирует все строковые поля верхнего уровня, но также ищет во вложенных документах и массивах строковые поля:

```
> db.articles.createIndex({"$**" : "text"})
```

Поиск по тексту

Используйте оператор запроса `"$text"` для выполнения текстового поиска в коллекции с текстовым индексом. Оператор `"$text"` будет токенизировать строку поиска, используя пробелы и большинство знаков препинания в качестве разделителей, и логический оператор «ИЛИ» для всех таких токенов в строке поиска. Например, можно использовать приведенный ниже запрос, чтобы найти все статьи, содержащие любой из термов – «impact», «crater» или «lunar». Обратите внимание, что поскольку наш индекс основан на термах как в заголовке, так и в теле статьи, этот запрос будет соответствовать документам, в которых эти термы находятся в любом поле. В этом примере мы спроектируем заголовок таким образом, чтобы у нас была возможность разместить больше результатов на странице:

```
> db.articles.find({"$text": {"$search": "impact crater lunar"}},
                   {title: 1}
                     ).limit(10)
{ "_id" : "170375", "title" : "Chengdu" }
{ "_id" : "34331213", "title" : "Avengers vs. X-Men" }
{ "_id" : "498834", "title" : "Culture of Tunisia" }
{ "_id" : "602564", "title" : "ABC Warriors" }
{ "_id" : "40255", "title" : "Jupiter (mythology)" }
{ "_id" : "80356", "title" : "History of Vietnam" }
{ "_id" : "22483", "title" : "Optics" }
{ "_id" : "8919057", "title" : "Characters in The Legend of Zelda series" }
{ "_id" : "20767983", "title" : "First inauguration of Barack Obama" }
{ "_id" : "17845285", "title" : "Kushiel's Mercy" }
```

Видно, что результаты нашего первоначального запроса не очень релевантны. Как и во всех технологиях, важно хорошо понимать, как работают текстовые индексы в MongoDB, чтобы эффективно их использовать.

В этом случае есть две проблемы с отправкой запроса. Во-первых, он довольно широкий, учитывая, что MongoDB отпрвляет запрос, используя логический оператор «ИЛИ» с термами «impact», «crater» и «lunar». Вторая проблема заключается в том, что по умолчанию поиск по тексту не сортирует результаты по релевантности.

Мы можем начать решать проблему самого запроса, используя фразу в нашем запросе. Можно искать точные фразы, заключив их в двойные кавычки. Например, в приведенном ниже примере будут найдены все документы, содержащие фразу «impact crater». Возможно, удивительно то, что MongoDB отправит этот запрос как «impact crater» И «lunar»:

```
> db.articles.find({$text: {$search: "\"impact crater\" lunar"}},
                   {title: 1}
                     ).limit(10)
{ "_id" : "2621724", "title" : "Schjellerup (crater)" }
{ "_id" : "2622075", "title" : "Steno (lunar crater)" }
{ "_id" : "168118", "title" : "South Pole-Aitken basin" }
{ "_id" : "1509118", "title" : "Jackson (crater)" }
{ "_id" : "10096822", "title" : "Victoria Island structure" }
{ "_id" : "968071", "title" : "Buldhana district" }
{ "_id" : "780422", "title" : "Puchezh-Katunki crater" }
{ "_id" : "28088964", "title" : "Svedberg (crater)" }
{ "_id" : "780628", "title" : "Zeleny Gai crater" }
{ "_id" : "926711", "title" : "Fracastorius (crater)" }
```

Чтобы убедиться, что семантика вам понятна, давайте рассмотрим расширенный пример. Для следующего запроса MongoDB сделает запрос как «impact crater» И («lunar» ИЛИ «meteor»). MongoDB использует логический оператор «И» фразы с отдельными термами в строке поиска и логический оператор «ИЛИ» отдельных термов друг с другом:

```
> db.articles.find({$text: {$search: "\"impact crater\" lunar meteor"}},
                   {title: 1}
                     ).limit(10)
```

Если вы хотите ввести логический оператор «И» между отдельными термами в запросе, обрабатывайте каждый терм как фразу, заключая его в кавычки. В результате приведенного ниже запроса будут возвращены документы, содержащие «impact crater» И «lunar» И «meteor»:

```
> db.articles.find({$text: {$search: "\"impact crater\" \"lunar\" \"meteor\""}},
                   {title: 1}
                     ).limit(10)
{ "_id" : "168118", "title" : "South Pole-Aitken basin" }
{ "_id" : "330593", "title" : "Giordano Bruno (crater)" }
```

```
{ "_id" : "421051", "title" : "Opportunity (rover)" }
{ "_id" : "2693649", "title" : "Pascal Lee" }
{ "_id" : "275128", "title" : "Tektite" }
{ "_id" : "14594455", "title" : "Beethoven quadrangle" }
{ "_id" : "266344", "title" : "Space debris" }
{ "_id" : "2137763", "title" : "Wegener (lunar crater)" }
{ "_id" : "929164", "title" : "Dawes (lunar crater)" }
{ "_id" : "24944", "title" : "Plate tectonics" }
```

Теперь, когда вы лучше понимаете, как использовать фразы и логические операторы «И» в своих запросах, давайте вернемся к проблеме сортировки результатов по релевантности. Хотя предыдущие результаты, безусловно, релевантны, в основном это связано с довольно строгим запросом, который мы делали. Мы можем добиться большего успеха, отсортировав результаты по релевантности.

Текстовые запросы приводят к тому, что некоторые метаданные становятся ассоциированными со всеми результатами запроса. Метаданные не отображаются в этих результатах, если мы явно не проецируем их с помощью оператора `$meta`. Таким образом, в дополнение к заголовку мы спроецируем показатель релевантности, рассчитанный для каждого документа. Показатель релевантности хранится в поле метаданных с именем `"textScore"`. В этом примере мы вернемся к нашему запросу «impact crater» И «lunar»:

```
> db.articles.find({$text: {$search: "\"impact crater\" lunar"}},
                   {title: 1, score: {$meta: "textScore"}}
                     ).limit(10)
{"_id": "2621724", "title": "Schjellerup (crater)", "score": 2.852987132352941}
{"_id": "2622075", "title": "Steno (lunar crater)", "score": 2.4766639610389607}
{"_id": "168118", "title": "South Pole–Aitken basin", "score": 2.980198136295181}
{"_id": "1509118", "title": "Jackson (crater)", "score": 2.3419137286324787}
{"_id": "10096822", "title": "Victoria Island structure","score": 1.782051282051282}
{"_id": "968071", "title": "Buldhana district", "score": 1.6279783393501805}
{"_id": "780422", "title": "Puchezh-Katunki crater", "score": 1.9295977011494254}
{"_id": "28088964", "title": "Svedberg (crater)", "score": 2.497767857142857}
{"_id": "780628", "title": "Zeleny Gai crater", "score": 1.4866071428571428}
{"_id": "926711", "title": "Fracastorius (crater)", "score": 2.7511877111486487}
```

Теперь можно увидеть показатель релевантности, спроецированный с заголовком для каждого результата. Обратите внимание, что они не отсортированы. Чтобы отсортировать результаты в порядке показателя релевантности, нужно добавить вызов функции `sort`, снова используя оператор `$meta`, дабы указать значение поля `"textScore"`. Обратите внимание, что мы

должны использовать то же имя поля в нашей сортировке, которое мы использовали в нашей проекции. В этом случае мы использовали имя поля `"score"` для значения показателя релевантности, отображаемого в наших результатах поиска. Как видно, результаты теперь сортируются в порядке убывания релевантности:

```
> db.articles.find({$text: {$search: "\"impact crater\" lunar"}},
                   {title: 1, score: {$meta: "textScore"}}
                     ).sort({score: {$meta: "textScore"}}).limit(10)
{"_id": "1621514", "title": "Lunar craters", "score": 3.1655242042922014}
{"_id": "14580008", "title": "Kuiper quadrangle", "score": 3.0847527829208814}
{"_id": "1019830", "title": "Shackleton (crater)", "score": 3.076471119932001}
{"_id": "2096232", "title": "Geology of the Moon", "score": 3.064981949458484}
{"_id": "927269", "title": "Messier (crater)", "score": 3.0638183133686008}
{"_id": "206589", "title": "Lunar geologic timescale", "score": 3.062029540854157}
{"_id": "14536060", "title": "Borealis quadrangle", "score": 3.0573010719646687}
{"_id": "14609586", "title": "Michelangelo quadrangle", "score": 3.057224063486582}
{"_id": "14568465", "title": "Shakespeare quadrangle", "score": 3.0495256481056443}
{"_id": "275128", "title": "Tektite", "score" : 3.0378807169646915}
```

Текстовый поиск также доступен в конвейере агрегации, который мы обсудим в главе 7.

Оптимизация полнотекстового поиска

Существует несколько способов оптимизировать полнотекстовый поиск. Если сначала вы можете сузить результаты поиска по другим критериям, можно создать составной индекс с префиксом этих критериев, а затем полнотекстовые поля:

```
> db.blog.createIndex({"date" : 1, "post" : "text"})
```

Это называется *разделением* полнотекстового индекса, поскольку так мы разбиваем его на несколько более мелких деревьев на основе даты (в этом примере), что значительно ускоряет полнотекстовый поиск по определенной дате или диапазону дат.

Вы также можете использовать постфикс других критериев для покрытия запросов индексом. Например, если бы мы возвращали только поля «author» и «post», то могли бы создать составной индекс для обоих:

```
> db.blog.createIndex({"post" : "text", "author" : 1})
```

Эти варианты можно сочетать:

```
> db.blog.createIndex({"date" : 1, "post" : "text", "author" : 1})
```

Поиск на других языках

Когда документ вставляется (или сперва создается индекс), MongoDB просматривает поля индекса и осуществляет поиск основы каждого слова, превращая его в существенную единицу. Однако в разных языках поиск основы осуществляется по-разному, поэтому нужно указать язык индекса или документа. Текстовые индексы позволяют указывать параметр `"default_language"`, который по умолчанию имеет значение `"english"`, но его можно поменять на ряд других языков (*см.* документацию по адресу https://oreil.ly/eUt0Z, где приводится актуальный список языков).

Например, чтобы создать франкоязычный индекс, можно использовать такой код:

```
> db.users.createIndex({"profil" : "text",
                        "interets" : "text"},
                       {"default_language" : "french"})
```

Теперь для стемминга будет использоваться французский язык, если не указано иное. Для каждого документа можно указывать язык с помощью поля `"language"`, которое описывает язык документа:

```
> db.users.insert({"username" : "swedishChef",
... "profile" : "Bork de bork", language : "swedish"})
```

Ограниченные коллекции

«Обычные» коллекции в MongoDB создаются динамически и автоматически увеличиваются в размере для размещения дополнительных данных. MongoDB также поддерживает другой тип коллекции, именуемый *ограниченной коллекцией,* которая создается заранее и имеет фиксированный размер (*см.* рис. 6.4).

Наличие коллекций фиксированного размера поднимает интересный вопрос: что произойдет, когда мы попытаемся вставить что-либо в ограниченную коллекцию, которая уже заполнена? Ответ заключается в том, что ограниченные коллекции ведут себя как циклические очереди: если у нас не хватает места, самый старый документ будет удален, а новый займет его место (см. рис. 6.5). Это означает, что ограниченные коллекции автоматически прекращают срок действия самых старых документов при вставке новых документов.

В ограниченных коллекциях некоторые операции запрещены. Документы нельзя перемещать или удалить (за исключением автоматического истечения срока действия, описанного ранее), а обновления, которые могут привести к увеличению размера документов, запрещены. Предотвращая две эти операции, мы гарантируем, что документы в ограниченной кол-

лекции хранятся в порядке вставки и нет необходимости поддерживать свободный список для пространства из удаленных документов.

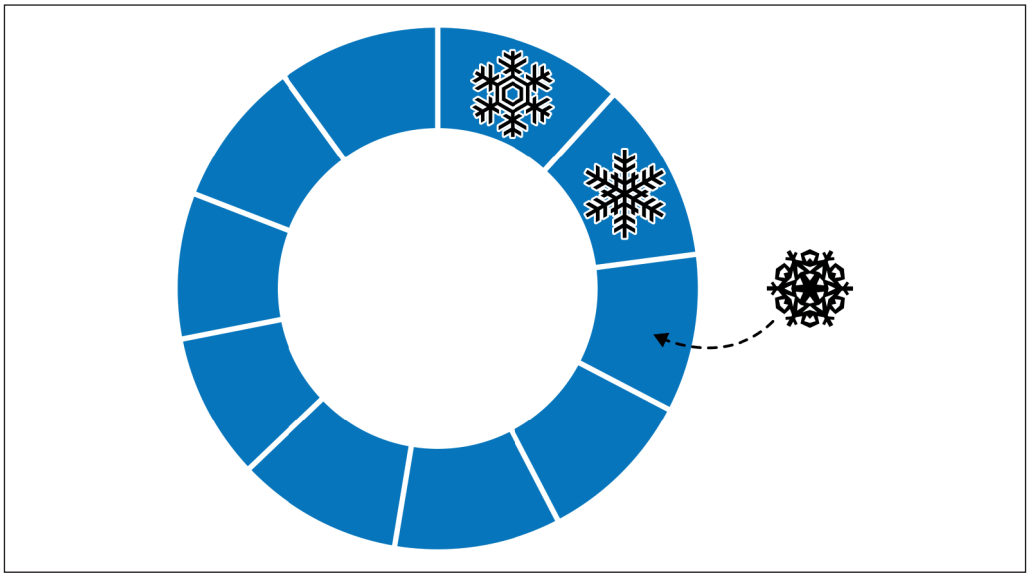

Рис. 6.4. Новые документы вставляются в конец очереди

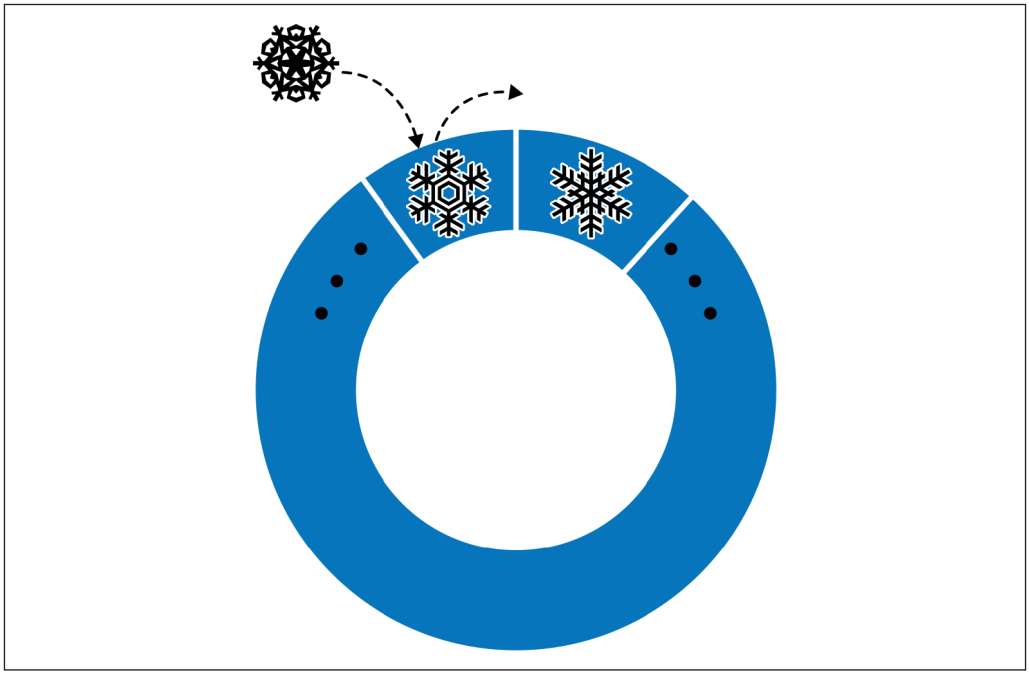

Рис. 6.5. Когда очередь заполнится, самый старый элемент будет заменен самым новым

Ограниченные коллекции имеют другой шаблон доступа, в отличие от большинства коллекций MongoDB: данные записываются последовательно по фиксированной части диска. Это заставляет их стремиться быстро выполнять запись на вращающиеся диски, особенно если им может быть предоставлен собственный диск (чтобы их не «прерывали» случайные записи других коллекций).

Как правило, в MongoDB для ограниченных коллекций рекомендуется использовать индексы TTL, поскольку они лучше работают с подсистемой хранения WiredTiger. Индексы TTL истекают и удаляют данные из обычных коллекций на основе значения поля с типом даты и TTL-значения индекса. Более подробно речь о них пойдет далее в этой главе.

Ограниченные коллекции нельзя распределять по разным серверам. Если операция по обновлению или замене изменяет размер документа в ограниченной коллекции, такая операция окончится неудачей.

Ограниченные коллекции, как правило, полезны при ведении журналов, хотя им не хватает гибкости: нельзя контролировать время устаревания данных, кроме установки размера при создании коллекции.

Создание ограниченных коллекций

В отличие от обычных коллекций, ограниченные коллекции должны быть созданы явно перед тем, как они будут использоваться. Чтобы создать ограниченную коллекцию, используйте команду `create`. Из оболочки это можно сделать с помощью метода `createCollection`:

```
> db.createCollection("my_collection", {"capped" : true, "size" : 100000});
```

С помощью предыдущей команды мы создали ограниченную коллекцию *my_collection* с фиксированным размером 100 000 байт.

Метод `createCollection` также может указывать ограничение на количество документов в ограниченной коллекции:

```
> db.createCollection("my_collection2",
                      {"capped" : true, "size" : 100000, "max" : 100});
```

Можно использовать это, чтобы сохранить, скажем, последние 10 новостных статей или ограничить количество документов для пользователя до 1000.

После того как ограниченная коллекция будет создана, ее нельзя изменить (ее необходимо удалить и создать заново, если вы хотите изменить ее

свойства). Таким образом, нужно тщательно продумать размер большой коллекции, прежде чем создавать ее.

При ограничении количества документов в ограниченной коллекции также необходимо указать ограничение размера. Срок действия будет зависеть от того, какой лимит достигнут первым: нельзя хранить больше максимального количества документов `"max"` или занимать больше места, чем указано в `"size"`.

Еще один вариант создания ограниченной коллекции – преобразовать существующую обычную коллекцию в ограниченную. Это можно сделать с помощью команды `convertToCapped` – в следующем примере мы преобразуем коллекцию *test* в ограниченную коллекцию из 10 000 байт:

```
> db.runCommand({"convertToCapped" : "test", "size" : 10000});
                { "ok" : true }
```

Нельзя сделать ограниченную коллекцию обычной (кроме как удалить ее).

Настраиваемые курсоры

Настраиваемые курсоры – особый тип курсоров, которые остаются открытыми, когда результаты исчерпаны. Они напоминают Unix-команду `tail -f`, и, подобно этой команде, они будут продолжать получать вывод как можно дольше. Поскольку курсоры не останавливаются, когда у них заканчиваются результаты, они могут продолжать получать новые результаты, по мере того как документы добавляются в коллекцию. Настраиваемые курсоры можно использовать только для ограниченных коллекций, поскольку в случае с обычными коллекциями порядок вставки не отслеживается. В подавляющем большинстве случаев потоки изменений, описанные в главе 16, рекомендуется использовать с настраиваемыми курсорами, поскольку они предлагают гораздо больше контроля и конфигурации, а также работают с обычными коллекциями.

Настраиваемые курсоры часто используются для обработки документов по мере их вставки в «рабочую очередь» (ограниченная коллекция). Поскольку настраиваемые курсоры будут отключаться через 10 минут, если результатов нет, важно использовать логику для повторного запроса коллекции, если они умирают. Оболочка *mongo* не позволяет использовать настраиваемые курсоры, но использование такого курсора в PHP выглядит примерно так:

```
$cursor = $collection->find([],[
    'cursorType' => MongoDB\Operation\Find: :TAILABLE_AWAIT,
    'maxAwaitTimeMS' => 100,
]);


    while (true) {
        if ($iterator->valid()) {
            $document = $iterator->current();
            printf("Consumed document created at: %s\n", $document->createdAt);
        }
        $iterator->next();
    }
```

Курсор будет обрабатывать результаты или ждать получения новых, пока не истечет время ожидания или кто-то не завершит операцию запроса.

Индексы TTL

Как упоминалось в предыдущем разделе, ограниченные коллекции дают вам ограниченный контроль над тем, когда их содержимое перезаписывается. Если вам нужна более гибкая система устаревания, индексы TTL позволяют установить тайм-аут для каждого документа. Когда документ достигает предварительно настроенного возраста, он будет удален. Данный тип индекса полезен для кеширования вариантов использования, таких как хранилище сессий.

Можно создать индекс TTL, указав параметр `"expireAfterSeconds"` во втором аргументе метода `createIndex`:

```
> // 24-часовой тайм-аут;
> db.sessions.createIndex({"lastUpdated" : 1}, {"expireAfterSeconds" : 60*60*24})
```

Мы создали индекс TTL в поле `"lastUpdated"`. Если поле документа `"lastUpdated"` существует и является датой, документ будет удален, как только время сервера будет на `"expireAfterSeconds"` секунд опережать время документа.

Чтобы предотвратить удаление активного сеанса, можно обновить поле `"lastUpdated"` и указать текущее время, когда есть активность. Как только возраст будет составлять 24 часа, документ будет удален.

MongoDB сканирует индекс TTL раз в минуту, поэтому не стоит зависеть от степени детализации с точностью до секунды. Можно изменить опцию `"expireAfterSeconds"` с помощью команды `collMod`:

```
> db.runCommand( {"collMod" : "someapp.cache" , "index" : { "keyPattern" :
... {"lastUpdated" : 1} , "expireAfterSeconds" : 3600 } } );
```

У вас может быть несколько индексов TTL для данной коллекции. Они не могут быть составными индексами, но их можно использовать как «обычные» индексы с целью сортировки и оптимизации запросов.

Хранение файлов с помощью GridFS

GridFS – механизм хранения больших двоичных файлов в MongoDB. Существует несколько причин, по которым вы можете рассмотреть возможность использования GridFS для хранения файлов:

- использование GridFS может упростить ваш стек. Если вы уже используете MongoDB, можно применять GridFS вместо отдельного инструмента для хранения файлов;
- GridFS будет использовать любую существующую репликацию или автоматический шардинг, которые вы настроили для MongoDB, поэтому добиться отказоустойчивости и масштабирования для хранения файлов будет легче;
- GridFS может облегчить некоторые проблемы, которые могут возникать у определенных файловых систем, когда они используются для хранения пользовательских загрузок. Например, у GridFS нет проблем с хранением большого количества файлов в одном каталоге.

Есть и некоторые недостатки:

- производительность будет ниже. Доступ к файлам из MongoDB не будет таким быстрым, как напрямую через файловую систему;
- вы можете изменить документы, только удалив их и сохранив все заново. MongoDB хранит файлы как множество документов, поэтому она не может заблокировать все фрагменты в файле одновременно.

GridFS, как правило, лучше всего подходит, когда у вас есть большие файлы, к которым вы будете получать доступ в последовательной манере, которая не будет сильно меняться.

Начало работы с GridFS: mongofiles

Самый простой способ опробовать GridFS – использовать утилиту *mongofiles*. Она входит в состав всех дистрибутивов MongoDB и может использоваться для загрузки, скачивания, просмотра, поиска или удаления файлов в GridFS.

Как и в случае с любым другим инструментом командной строки, выполните команду `mongofiles --help`, чтобы увидеть параметры, доступные для *mongofiles*.

В приведенном ниже примере показано, как использовать *mongofiles* для загрузки файла из файловой системы в GridFS, перечисления всех файлов в GridFS и скачивания файла, который мы предварительно загрузили:

```
$ echo "Hello, world" > foo.tx
$ mongofiles put foo.txt
2019-10-30T10:12:06.588+0000 connected to: localhost
2019-10-30T10:12:06.588+0000 added file: foo.txt
$ mongofiles list
2019-10-30T10:12:41.603+0000 connected to: localhost
foo.txt 13
$ rm foo.txt
$ mongofiles get foo.txt
2019-10-30T10:13:23.948+0000 connected to: localhost
2019-10-30T10:13:23.955+0000 finished writing to foo.txt
$ cat foo.txt
Hello, world
```

В предыдущем примере мы выполняем три основные операции с использованием *mongofiles*: `put`, `list` и `get`. Используя операцию `put`, мы берем файл в файловой системе и добавляем его в GridFS. С помощью операции `list` будут перечислены все файлы, которые были добавлены в GridFS. Используя операцию `get`, мы выполняем обратное действие: берем файл из GridFS и записываем его в файловую систему. *mongofiles* также поддерживает еще две операции: `search` для поиска файлов в GridFS по имени файла и `delete` для удаления файла из GridFS.

Работа с GridFS из драйверов MongoDB

У всех клиентских библиотек есть API GridFS. Например, с помощью PyMongo (драйвер Python для MongoDB) можно выполнять те же серии операций (что предполагает использование Python версии 3 и локально работающего *mongod* на порту 27017), что и с помощью *mongofiles*:

```
>>> import pymongo
>>> import gridfs
>>> client = pymongo.MongoClient()
>>> db = client.test
>>> fs = gridfs.GridFS(db)
>>> file_id = fs.put(b"Hello, world", filename="foo.txt")
>>> fs.list()
['foo.txt']
>>> fs.get(file_id).read()
b'Hello, world'
```

API для работы с GridFS из PyMongo очень похож на API *mongofiles*: вы можете с легкостью выполнять основные операции `put`, `list` и `get`. Почти все драйверы MongoDB следуют этому базовому шаблону при работе с GridFS, в то же время часто предоставляя более продвинутую функциональность. Для получения информации о драйверах в GridFS, пожалуйста, ознакомьтесь с документацией по конкретному драйверу, который вы используете.

Что под капотом

GridFS – легковесная спецификация для хранения файлов, созданная поверх обычных документов MongoDB. Сервер MongoDB практически ничего не делает для «особого случая» обработки запросов GridFS; вся работа выполняется драйверами на стороне клиента и инструментами.

Основная идея GridFS заключается в том, что мы можем хранить большие файлы, разбивая их на *чанки* и сохраняя каждый такой фрагмент в виде отдельного документа. Поскольку MongoDB поддерживает хранение двоичных данных в документах, можно свести накладные расходы на сохранение чанков к минимуму. Помимо хранения каждого чанка, мы храним единый документ, который группирует их вместе и содержит метаданные о файле.

Чанки хранятся в собственной коллекции. По умолчанию они будут использовать коллекцию *fs.chunks*, но ее можно переопределить. Внутри этой коллекции структура отдельных документов довольно проста:

```
{
    "_id": ObjectId("..."),
    "n": 0,
    "data": BinData("..."),
    "files_id": ObjectId("...")
}
```

Как и любой другой документ MongoDB, у чанка есть свой уникальный `"_id"`. Кроме того, у него есть пара других ключей:

`"files_id"`
: `"_id"` файлового документа, который содержит метаданные файла, из которого этот чанк (chunk).

`"n"`
: Расположение чанка в файле относительно других чанков.

`"data"`
: Байты этого чанка в файле.

Метаданные каждого файла хранятся в отдельной коллекции, по умолчанию это `fs.files`. Каждый документ в коллекции файлов представляет

собой отдельный файл в GridFS и может содержать любые пользовательские метаданные, которые должны быть связаны с этим файлом. В дополнение к пользовательским ключам есть еще несколько ключей, которые предписаны спецификацией GridFS:

`"_id"`
: Уникальный идентификатор файла – то, что будет храниться в каждом чанке в качестве значения ключа `"files_id"`.

`"length"`
: Общее число байтов, составляющих содержимое файла.

`"chunkSize"`
: Размер каждого чанка, содержащего файл, в байтах. По умолчанию установлено значение 255 КБ, но при необходимости его можно изменить.

`"uploadDate"`
: Временная отметка, обозначающая, когда этот файл хранился в GridFS.

`"md5"`
: Контрольная сумма MD5 содержимого этого файла, сгенерированная на стороне сервера.

Из всех необходимых ключей, пожалуй, наиболее интересным (или наименее понятным) является `"md5"`. Значение `"md5"` генерируется сервером MongoDB с помощью команды `filemd5`, которая вычисляет контрольную сумму MD5 загруженных чанков. Это означает, что пользователи могут проверить значение ключа `"md5"`, чтобы убедиться, что файл был загружен правильно.

Как упоминалось ранее, вы не ограничены обязательными полями в файле *fs.files*: вы также можете свободно хранить любые другие метаданные файла в этой коллекции. Возможно, вы захотите сохранить информацию, такую как количество скачиваний, тип MIME или рейтинг пользователя, с метаданными файла.

Как только вы освоите базовую спецификацию GridFS, реализация функций, для которых используемый драйвер может не предоставлять помощников, станет тривиальной задачей. Например, можно использовать команду `distinct`, чтобы получить список уникальных имен файлов, хранящихся в GridFS:

```
> db.fs.files.distinct("filename")
[ "foo.txt" , "bar.txt" , "baz.txt" ]
```

Это дает вашему приложению большую гибкость при загрузке и сборе информации о файлах. Мы немного изменим направление в следующей главе, когда будем знакомиться с фреймворком агрегации. Он предлагает ряд инструментов для анализа данных для обработки данных в вашей базе данных.

Глава 7

Знакомство с фреймворком агрегации

Многие приложения требуют анализа данных в той или иной форме. MongoDB предоставляет мощную поддержку для естественного запуска аналитики с использованием фреймворка агрегации. В этой главе мы познакомимся с этим фреймворком и некоторыми основными инструментами, которые он предоставляет.

Мы рассмотрим:

- фреймворк агрегации;
- этапы агрегации;
- выражения агрегации;
- аккумуляторы агрегации.

В следующей главе мы подробнее рассмотрим более продвинутые функции агрегации, в том числе возможность выполнения объединений коллекций.

Конвейеры, этапы и настраиваемые параметры

Фреймворк агрегации представляет собой набор аналитических инструментов, которые позволяют работать с документами в одной или нескольких коллекциях.

Он основан на концепции конвейера. С помощью конвейера агрегации мы берем входные данные из коллекции MongoDB и пропускаем документы из этой коллекции через один или несколько этапов, каждый из которых выполняет свою операцию (рис. 7.1). Каждый этап рассматривается в качестве входных данных, независимо от этапа, прежде чем будут созданы выходные данные. Входные и выходные данные всех этапов – это документы, поток документов, если хотите.

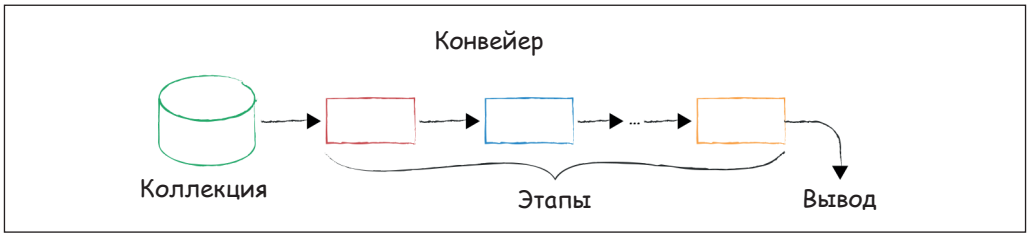

Рис. 7.1. Конвейер агрегации

Если вы знакомы с конвейерами в оболочке Linux, такими как bash, это примерно то же самое. У каждого этапа есть определенное задание, которое он выполняет. Он ожидает конкретную форму документа и производит определенный вывод, который сам по себе является потоком документов. В конце конвейера мы получаем доступ к выводу почти так же, как если бы мы выполнили запрос на поиск. То есть мы получаем поток документов, который затем можем использовать для выполнения дополнительной работы, будь то создание какого-либо рода отчета, создание веб-сайта или какая-то иная задача.

Теперь давайте пойдем дальше и рассмотрим отдельные этапы. Отдельным этапом конвейера агрегации является блок обработки данных. Он принимает поток входных документов по одному за раз, обрабатывает каждый документ по одному за раз и создает выходной поток документов по одному за раз (рис. 7.2).

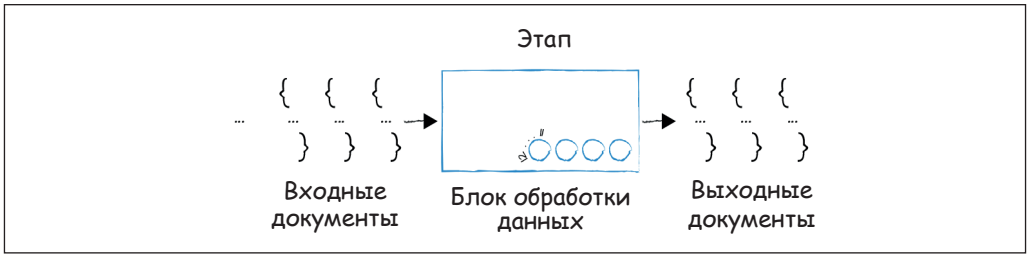

Рис. 7.2. Этапы конвейера агрегации

Каждый этап предоставляет набор настраиваемых параметров, которыми мы можем управлять для параметризации этапа для выполнения любой задачи, которая нас интересует. Этап выполняет какую-то обобщенную задачу общего назначения, и мы параметризируем этап для конкретной коллекции, с которой мы работаем, и конкретно то, что мы хотели бы, чтобы этот этап делал с этими документами.

Эти настраиваемые параметры обычно принимают форму операторов, которые мы можем предоставить и которые будут изменять поля, выполнять арифметические операции, изменять форму документов или выполнять какую-либо задачу по аккумулированию и множество других вещей.

Прежде чем приступить к рассмотрению конкретных примеров, особенно важно помнить еще об одном аспекте конвейеров, когда вы начинаете работать с ними. Часто нам нужно включить один и тот же тип этапа несколько раз в один конвейер (рис. 7.3). Например, нам может понадобиться выполнить начальную фильтрацию, чтобы не нужно было передавать всю коллекцию в конвейер. Позже, после дополнительной обработки, мы могли бы выполнять фильтрацию дальше, применяя другой набор критериев.

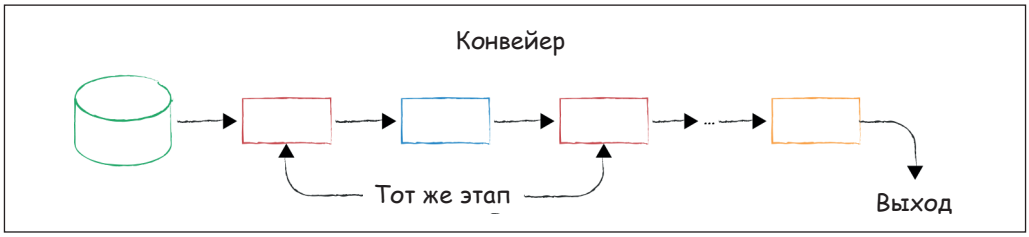

Рис. 7.3. Повторные этапы в конвейере агрегации

Напомним, что конвейеры работают с коллекциями MongoDB. Они состоят из этапов, каждый из которых выполняет свою задачу обработки данных на входе и создает документы в качестве выходных данных для передачи их следующему этапу. Наконец, в конце обработки конвейер создает выходные данные, с которыми мы можем что-то сделать в нашем приложении или которые можно отправить в коллекцию для дальнейшего использования. Во многих случаях, чтобы выполнить необходимый нам анализ, мы будем использовать один и тот же тип этапа несколько раз в рамках отдельного конвейера.

Начало работы с этапами: знакомые операции

Чтобы приступить к разработке конвейеров агрегации, мы рассмотрим создание конвейеров, которые включают в себя уже знакомые вам операции. Для этого мы рассмотрим этапы *match*, *project*, *sort*, *skip* и *limit*.

Для работы с этими примерами агрегации мы будем использовать коллекцию данных компании. В этой коллекции есть несколько полей, в которых указаны сведения о компаниях, такие как название, краткое описание компании и время ее основания.

Есть также поля, описывающие этапы финансирования, которые прошла компания, важные вехи в истории компании, имело ли место проведение первичного публичного размещения акций (IPO) или нет, и, если такое было, подробности IPO. Вот пример документа, содержащего данные о компании Facebook, Inc.:

```
{
    "_id": "52cdef7c4bab8bd675297d8e",
```

```
"name": "Facebook",
"category_code": "social",
"founded_year": 2004,
"description": "Social network",
"funding_rounds": [
    {
        "id": 4,
        "round_code": "b",
        "raised_amount": 27500000,
        "raised_currency_code": "USD",
        "funded_year": 2006,
        "investments": [
            {
                "company": null,
                "financial_org": {
                    "name": "Greylock Partners",
                    "permalink": "greylock"
                },
                "person": null
            },
            {
                "company": null,
                "financial_org": {
                    "name": "Meritech Capital Partners",
                    "permalink": "meritech-capital-partners"
                },
                "person": null
            },
            {
                "company": null,
                "financial_org": {
                    "name": "Founders Fund",
                    "permalink": "founders-fund"
                },
                "person": null
            },
            {
                "company": null,
                "financial_org": {
                    "name": "SV Angel",
                    "permalink": "sv-angel"
                },
                "person": null
            }
        ]
```

```
        },
        {
            "id": 2197,
            "round_code": "c",
            "raised_amount": 15000000,
            "raised_currency_code": "USD",
            "funded_year": 2008,
            "investments": [
                {
                    "company": null,
                    "financial_org": {
                        "name": "European Founders Fund",
                        "permalink": "european-founders-fund"
                    },
                    "person": null
                }
            ]
        }
    ],
    "ipo": {
        "valuation_amount": NumberLong("104000000000"),
        "valuation_currency_code": "USD",
        "pub_year": 2012,
        "pub_month": 5,
        "pub_day": 18,
        "stock_symbol": "NASDAQ:FB"
    }
}
```

В качестве нашего первого примера агрегации давайте выполним простую фильтрацию, разыскивая все компании, которые были основаны в 2004 году:

```
db.companies.aggregate([
    {$match: {founded_year: 2004}},
])
```

Это эквивалентно приведенной ниже операции с использованием метода `find`:

```
db.companies.find({founded_year: 2004})
```

Теперь давайте добавим в наш конвейер этап `$project`, чтобы уменьшить вывод до нескольких полей на документ. Мы уберем поле `"_id"` и добавим поля `"name"` и `"based_year"`. Наш конвейер будет выглядеть следующим образом:

```
db.companies.aggregate([
    {$match: {founded_year: 2004}},
    {$project: {
       _id: 0,
       name: 1,
       founded_year: 1
       }
    }
])
```

Если выполнить этот код, мы получим вывод, который выглядит так:

```
{"name": "Digg", "founded_year": 2004 }
{"name": "Facebook", "founded_year": 2004 }
{"name": "AddThis", "founded_year": 2004 }
{"name": "Veoh", "founded_year": 2004 }
{"name": "Pando Networks", "founded_year": 2004 }
{"name": "Jobster", "founded_year": 2004 }
{"name": "AllPeers", "founded_year": 2004 }
{"name": "blinkx", "founded_year": 2004 }
{"name": "Yelp", "founded_year": 2004 }
{"name": "KickApps", "founded_year": 2004 }
{"name": "Flickr", "founded_year": 2004 }
{"name": "FeedBurner", "founded_year": 2004 }
{"name": "Dogster", "founded_year": 2004 }
{"name": "Sway", "founded_year": 2004 }
{"name": "Loomia", "founded_year": 2004 }
{"name": "Redfin", "founded_year": 2004 }
{"name": "Wink", "founded_year": 2004 }
{"name": "Techmeme", "founded_year": 2004 }
{"name": "Eventful", "founded_year": 2004 }
{"name": "Oodle", "founded_year": 2004 }
...
```

Давайте рассмотрим этот конвейер агрегации немного подробнее. Первое, что вы заметите, – это то, что мы используем метод `aggregate`. Это метод, который мы вызываем, когда хотим выполнить запрос на агрегацию. Для того чтобы осуществить агрегацию, мы переходим в конвейер агрегации. Конвейер – это массив с документами в качестве элементов. Каждый из документов должен предусматривать оператор конкретного этапа. В этом примере у нас есть конвейер, у которого два этапа: этап `$match` для фильтрации и этап `$project`, с помощью которого мы ограничиваем вывод только двумя полями на документ.

Этап `$match` фильтрует коллекцию и передает полученные документы в этап проектирования по одному. Затем этап `$project` выполняет свою ра-

боту, изменяя форму документов, и передает результаты из конвейера нам обратно.

Теперь давайте немного расширим наш конвейер, чтобы включить в него этап `$limit`. Мы хотим выполнить сопоставление, используя один и тот же запрос, но мы ограничим наш набор результатов пятью результатами, а затем спроецируем нужные поля. Чтобы было проще, давайте ограничим наш вывод только названиями компаний:

```
db.companies.aggregate([
    {$match: {founded_year: 2004 } },
    {$limit: 5},
    {$project: {
       _id: 0,
       name: 1}
    }
])
```

Результат выглядит следующим образом:

```
{"name": "Digg"}
{"name": "Facebook"}
{"name": "AddThis"}
{"name": "Veoh"}
{"name": "Pando Networks"}
```

Обратите внимание, что мы построили этот конвейер таким образом, чтобы ограничение шло до этапа `project`. Если бы мы сначала запустили этап `project`, а затем этап `limit`, как в следующем запросе, то получили бы точно такие же результаты, но нам пришлось бы пропустить через этап `project` сотни документов, прежде чем окончательно ограничить результаты пятью вариантами:

```
db.companies.aggregate([
    {$match: {founded_year: 2004 } },
    {$project: {
       _id: 0,
       name: 1}
    },
    {$limit: 5}
])
```

Независимо от того, на какие типы оптимизаций может рассчитывать планировщик запросов MongoDB в данном выпуске, всегда нужно учитывать эффективность конвейера агрегации. Убедитесь, что вы ограничиваете количество документов, которые должны быть переданы от одного этапа к другому при создании конвейера.

Это требует тщательного рассмотрения всего потока документов, идущих через конвейер. В случае с предыдущим запросом нас интересуют только первые пять документов, которые соответствуют нашему запросу, независимо от того, как они отсортированы, поэтому, если мы ограничимся вторым этапом, все в порядке.

Однако, если порядок имеет значение, нам нужно будет выполнить сортировку до этапа `$limit`. Сортировка работает аналогично тому, что мы уже видели, за исключением того, что во фреймворке агрегации мы определяем сортировку как этап в рамках конвейера (в этом случае мы будем делать сортировку по имени в порядке возрастания):

```
db.companies.aggregate([
    { $match: { founded_year: 2004 } },
    { $sort: { name: 1} },
    { $limit: 5},
    { $project: {
       _id: 0,
       name: 1}
     }
])
```

Мы получаем следующий результат от нашей коллекции *companies*:

```
{"name": "1915 Studios"}
{"name": "1Scan"}
{"name": "2GeeksinaLab"}
{"name": "2GeeksinaLab"}
{"name": "2threads"}
```

Обратите внимание, что сейчас перед нами другой набор из пяти компаний, и теперь мы получаем первые пять документов в алфавитно-цифровом порядке по названию.

Наконец, давайте посмотрим на работу этапа `$skip`. Здесь мы сначала сортируем, затем пропускаем первые 10 документов и снова ограничиваем наш набор результатов пятью документами:

```
db.companies.aggregate([
    {$match: {founded_year: 2004 } },
    {$sort: {name: 1 } },
    {$skip: 10},
    {$limit: 5},
    {$project: {
       _id: 0,
       name: 1}
    },
])
```

Давайте еще раз рассмотрим наш конвейер. У нас есть пять этапов. Сначала мы отфильтровываем коллекцию *companies*, ища только те документы, у которых значение `"based_year"` равно 2004. Затем мы делаем сортировку по имени в порядке возрастания, пропуская первые 10 совпадений, и ограничиваем наши конечные результаты пятью документами. Наконец, мы передаем эти пять документов в этап `$project`, где изменяем форму документов таким образом, чтобы наши выходные документы содержали только название компании.

Здесь мы рассмотрели создание конвейеров с использованием этапов, которые выполняют операции, которые должны быть вам уже знакомы. Эти операции предоставляются фреймворком агрегации, потому что они необходимы для тех типов аналитики, которые мы хотим выполнить, используя этапы, о которых пойдет речь в последующих разделах. По мере прохождения оставшейся части этой главы мы подробно рассмотрим и другие операции, которые предоставляет этот фреймворк.

Выражения

По мере того как мы углубляемся в обсуждение фреймворка агрегации, важно иметь представление о различных типах выражений, доступных для использования при создании конвейеров агрегации. Фреймворк агрегации поддерживает множество различных классов выражений:

- *логические* выражения позволяют нам использовать выражения И, ИЛИ и НЕ;
- выражения *множеств* дают нам возможность работать с массивами в качестве множеств. В частности, мы можем получить пересечение или объединение двух или более множеств, а также можем взять разницу двух множеств и выполнить ряд других операций над множествами;
- выражения *сравнения* позволяют нам выражать множество различных типов фильтров диапазонов;
- *арифметические* выражения позволяют нам вычислять целую часть («пол» и «потолок»), натуральный логарифм и логарифм, а также выполнять простые арифметические операции, такие как умножение, деление, сложение и вычитание. Можно даже выполнять более сложные операции, такие как вычисление квадратного корня из значения;
- *строковые* выражения позволяют нам объединять, находить подстроки и выполнять операции, связанные с регистром и операциями текстового поиска;
- выражения *массивов* предоставляют множество возможностей для манипулирования массивами, включая возможность фильтровать

элементы массива, разрезать массив или просто получать диапазон значений из определенного массива;

- выражения *переменных*, в которые мы не будем вдаваться слишком подробно, позволяют работать с литералами, выражениями для парсинга значений дат и условными выражениями;
- *аккумуляторы* предоставляют возможность расчета сумм, описательной статистики и многих других типов значений.

$project

Теперь мы более подробно рассмотрим этап `$project` и изменения структуры документов, изучив типы операций преобразования, которые должны быть наиболее распространены в разрабатываемых вами приложениях. Мы уже видели несколько простых этапов в конвейерах агрегации, а теперь рассмотрим те, что немного сложнее.

Сперва давайте посмотрим на передачу вложенных полей в следующий этап. В приведенном ниже конвейере мы используем оператор `$match`:

```
db.companies.aggregate([
    {$match: { "funding_rounds.investments.financial_org.permalink": "greylock" } },
    {$project: {
       _id: 0,
       name: 1,
       ipo: "$ipo.pub_year",
       valuation: "$ipo.valuation_amount",
       funders: "$funding_rounds.investments.financial_org.permalink"
       }
    }
]).pretty()
```

В качестве примера релевантных полей документов в нашей коллекции *companies* давайте еще раз посмотрим на фрагмент документа Facebook:

```
{
    "_id": "52cdef7c4bab8bd675297d8e",
    "name": "Facebook",
    "category_code": "social",
    "founded_year": 2004,
    "description": "Social network",
    "funding_rounds": [
        {
            "id": 4,
            "round_code": "b",
            "raised_amount": 27500000,
```

```
        "raised_currency_code": "USD",
        "funded_year": 2006,
        "investments": [
            {
                "company": null,
                "financial_org": {
                    "name": "Greylock Partners",
                    "permalink": "greylock"
                },
                "person": null
            },
            {
                "company": null,
                "financial_org": {
                    "name": "Meritech Capital Partners",
                    "permalink": "meritech-capital-partners"
                },
                "person": null
            },
            {
                "company": null,
                "financial_org": {
                    "name": "Founders Fund",
                    "permalink": "founders-fund"
                },
                "person": null
            },
            {
                "company": null,
                "financial_org": {
                    "name": "SV Angel",
                    "permalink": "sv-angel"
                },
                "person": null
            }
        ]
    },
    {
        "id": 2197,
        "round_code": "c",
        "raised_amount": 15000000,
        "raised_currency_code": "USD",
        "funded_year": 2008,
        "investments": [
            {
```

```
                    "company": null,
                    "financial_org": {
                        "name": "European Founders Fund",
                        "permalink": "european-founders-fund"
                    },
                    "person": null
                }
            ]
        }
    ],
    "ipo": {
        "valuation_amount": NumberLong("104000000000"),
        "valuation_currency_code": "USD",
        "pub_year": 2012,
        "pub_month": 5,
        "pub_day": 18,
        "stock_symbol": "NASDAQ:FB"
    }
}
```

Вернемся к нашему оператору `$match`:

```
db.companies.aggregate([
    {$match: { "funding_rounds.investments.financial_org.permalink": "greylock" } },
    {$project: {
       _id: 0,
       name: 1,
       ipo: "$ipo.pub_year",
       valuation: "$ipo.valuation_amount",
       funders: "$funding_rounds.investments.financial_org.permalink"
       }
    }
]).pretty()
```

Мы проводим фильтрацию для всех компаний, прошедших раунд финансирования, в котором участвовала компания Greylock Partners. Значение постоянной ссылки, `"greylock"`, – это уникальный идентификатор таких документов. Вот еще один вид документа Facebook, где отображены только релевантные поля:

```
{
    ...
    "name": "Facebook",
    ...
    "funding_rounds": [
        {
```

```
        ...
        "investments": [
            {
            ...
            "financial_org": {
                        "name": "Greylock Partners",
                        "permalink": "greylock"
            },
            ...
        },
        {
            ...
            "financial_org": {
                        "name": "Meritech Capital Partners",
                        "permalink": "meritech-capital-partners"
            },
            ...
         },
         {
            ...
            "financial_org": {
                        "name": "Founders Fund",
                        "permalink": "founders-fnd"
            },
            ...
         },
         {
            "company": null,
            "financial_org": {
            "name": "SV Angel",
            "permalink": "sv-angel"
            },
            ...
         }],
         ...
    ]},
    {
    ...
    "investments": [
            {
            ...
            "financial_org": {
                    "name": "European Founders Fund",
                    "permalink": "european-founders-fund"
            },
```

```
            ...
            }]
    }],
    "ipo": {
        "valuation_amount": NumberLong("104000000000"),
        "valuation_currency_code": "USD",
        "pub_year": 2012,
        "pub_month": 5,
        "pub_day": 18,
        "stock_symbol": "NASDAQ:FB"
    }
}
```

Этап `$project`, который мы определили в этом конвейере агрегации, будет подавлять `"_id"` и использовать `"name"`, а также передавать некоторые вложенные поля в следующий этап. Здесь используется точечная нотация для обозначения путей, которые ведут к полям `"ipo"` и `"funding_rounds"` для выбора значений из этих вложенных документов и массивов. Данный этап сделает их значениями полей верхнего уровня в документах, которые он выдает в качестве вывода, как показано здесь:

```
{
    "name": "Digg",
    "funders": [
        [
            "greylock",
            "omidyar-network"
        ],
        [
            "greylock",
            "omidyar-network",
            "floodgate",
            "sv-angel"
        ],
        [
            "highland-capital-partners",
            "greylock",
            "omidyar-network",
            "svb-financial-group"
        ]
    ]
}
    {
    "name": "Facebook",
    "ipo": 2012,
```

```
        "valuation": NumberLong("104000000000"),
        "funders": [
            [
                "accel-partners"
            ],
            [
                "greylock",
                "meritech-capital-partners",
                "founders-fund",
                "sv-angel"
            ],
            ...
            [
                "goldman-sachs",
                "digital-sky-technologies-fo"
            ]
        ]
    }
            {
        "name": "Revision3",
        "funders": [
            [
                "greylock",
                "sv-angel"
            ],
            [
                "greylock"
            ]
        ]
    }
    ...
```

В выводе у каждого документа есть поле `"name"` и поле `"funders"`. Для тех компаний, которые прошли процесс проведения первоначального публичного предложения акций, поле `"ipo"` содержит год, когда компания стала продавать акции на бирже, а поле `"valuation"` содержит стоимость компании на момент проведения IPO. Обратите внимание, что во всех этих документах это поля верхнего уровня, а значения этих полей были переданы из вложенных документов и массивов.

Символ `$`, используемый для указания значений полей `"ipo"`, `"valuation"` и `"funders"` в нашем этапе, указывает на то, что значения должны интерпретироваться как пути к полям и применяться для выбора значения, которое должно быть спроецировано для каждого поля соответственно.

Вы, возможно, заметили, что мы видим несколько значений, выведенных для поля `"funders"`. На самом деле мы видим массив массивов. Основы-

ваясь на нашем обзоре примера с документом Facebook, мы знаем, что все спонсоры перечислены в массиве под названием `"investments"`. Наш этап указывает, что мы хотим спроецировать значение `financial_org.permalink` для каждой записи в массиве `"investments"`, для каждого раунда финансирования. Таким образом, создается массив, состоящий из массивов имен спонсоров.

В последующих разделах мы рассмотрим, как выполнять арифметические и другие операции со строками, датами и рядом других типов значений для проецирования документов всех форм и размеров. Практически единственное, чего мы не можем сделать в этом этапе, – это изменить тип данных для значения.

$unwind

При работе с полями массива в конвейере агрегации часто возникает необходимость использовать один или несколько этапов с использованием оператора `$unwind`. Это позволяет нам производить вывод таким образом, что для каждого элемента в указанном поле массива имеется один выходной документ.

В примере, изображенном на рис. 7.4, у нас есть входной документ, у которого есть три ключа и соответствующие им значения. Третий ключ имеет в качестве значения массив из трех элементов. Оператор `$unwind`, если он используется для входного документа данного типа и настроен на работу с полем `key3`, создаст документы, похожие на те, что показаны в нижней части рис. 7.4. Вам может показаться непонятным, что в каждом из этих выходных документов будет поле `key3`, но это поле будет содержать одно значение, а не значение массива, и для каждого из элементов этого массива, которые находились в массиве, будет отдельный документ. Другими словами, если бы в массиве было 10 элементов, на этапе `$unwind` получилось бы 10 выходных документов.

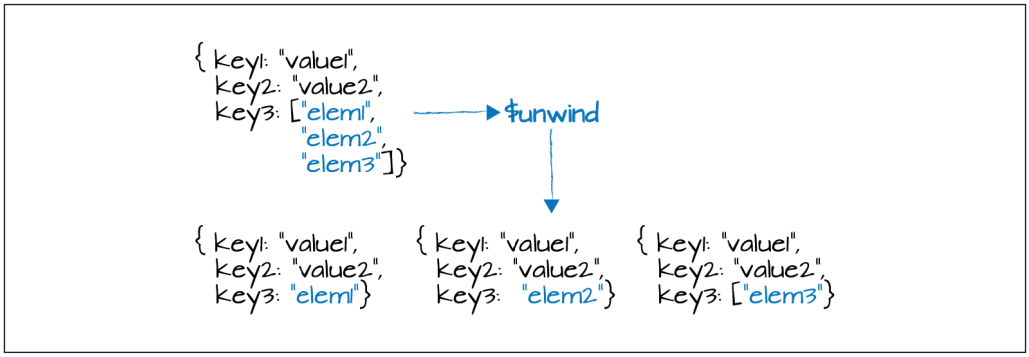

Рис. 7.4. Оператор $unwind берет массив из входного документа и создает выходной документ для каждого элемента в этом массиве

Давайте вернемся к нашему примеру с коллекцией *companies* и рассмотрим использование этого этапа. Начнем с приведенного ниже конвейера агрегации. Обратите внимание, что в этом конвейере, как и в предыдущем разделе, мы работаем с конкретным спонсором (`{$match: {"funding_rounds.investments.financial_org.permalink": "greylock"} }`) и передаем значения из вложенных документов `funding_rounds`, используя этап `$project`:

```
db.companies.aggregate([
    {$match: { "funding_rounds.investments.financial_org.permalink": "greylock" } },
    {$project: {
        _id: 0,
        name: 1,
        amount: "$funding_rounds.raised_amount",
        year: "$funding_rounds.funded_year"
        }
    }
])
```

Еще раз, вот пример модели данных для документов в этой коллекции:

```
{
    "_id": "52cdef7c4bab8bd675297d8e",
    "name": "Facebook",
    "category_code": "social",
    "founded_year": 2004,
    "description": "Social network",
    "funding_rounds": [
        {
            "id": 4,
            "round_code": "b",
            "raised_amount": 27500000,
            "raised_currency_code": "USD",
            "funded_year": 2006,
            "investments": [
                {
                    "company": null,
                    "financial_org": {
                        "name": "Greylock Partners",
                        "permalink": "greylock"
                    },
                    "person": null
                },
                {
                    "company": null,
                    "financial_org": {
```

```
                        "name": "Meritech Capital Partners",
                        "permalink": "meritech-capital-partners"
                    },
                    "person": null
                },
                {
                    "company": null,
                    "financial_org": {
                        "name": "Founders Fund",
                        "permalink": "founders-fund"
                    },
                    "person": null
                },
                {
                    "company": null,
                    "financial_org": {
                        "name": "SV Angel",
                        "permalink": "sv-angel"
                    },
                    "person": null
                }
            ]
        },
        {
            "id": 2197,
            "round_code": "c",
            "raised_amount": 15000000,
            "raised_currency_code": "USD",
            "funded_year": 2008,
            "investments": [
                {
                    "company": null,
                    "financial_org": {
                        "name": "European Founders Fund",
                        "permalink": "european-founders-fund"
                    },
                    "person": null
                }
            ]
        }
    ],
    "ipo": {
        "valuation_amount": NumberLong("104000000000"),
        "valuation_currency_code": "USD",
        "pub_year": 2012,
        "pub_month": 5,
        "pub_day": 18,
```

```
            "stock_symbol": "NASDAQ:FB"
        }
    }
```

Наш запрос на агрегацию даст следующие результаты:

```
{
    "name" : "Digg",
    "amount" : [
        8500000,
        2800000,
        28700000,
        5000000
    ],
    "year" : [
        2006,
        2005,
        2008,
        2011
    ]
}
{
    "name" : "Facebook",
    "amount" : [
        500000,
        12700000,
        27500000,
        ...
```

В ходе запроса создаются документы, у которых есть массивы и для `"amount"`, и для `"year"`, потому что мы выполняем доступ к `"raised_amount"` и `"funded_year"` для каждого элемента в массиве `"funding_rounds"`.

Чтобы исправить это, можно включить этап `$unwind` перед этапом `$project` в этот конвейер агрегации и параметризировать его, указав, что к массиву `"funding_rounds"` должен быть применен оператор `$unwind` (рис. 7.5).

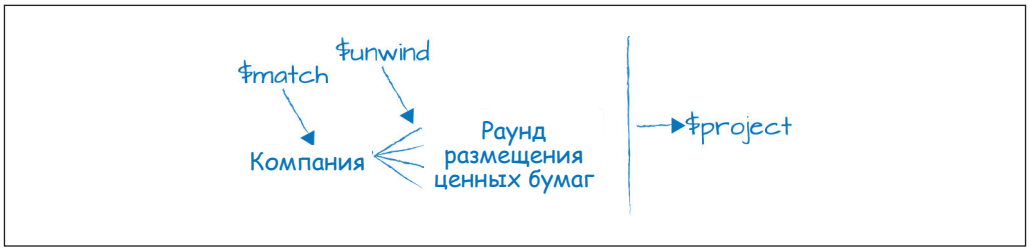

Рис. 7.5. Схема конвейера агрегации на данный момент, соответствует `"greylock"`, затем раскручивает `"funding_rounds"` и, наконец, проецирует имя, сумму и год для каждого раунда финансирования

Снова возвращаясь к нашему примеру с Facebook, мы видим, что для каждого раунда размещения ценных бумаг есть поле `"raised_amount"` и поле `"funded_year"`.

На этапе с оператором `$unwind` будет создан выходной документ для каждого элемента массива `"funding_rounds"`. В этом примере наши значения являются строками, но независимо от типа значения данный этап создаст выходной документ для каждого из них. Вот обновленный запрос на агрегацию:

```
db.companies.aggregate([
    { $match: { "funding_rounds.investments.financial_org.permalink":
"greylock" } },
    { $unwind: "$funding_rounds"
    },
    { $project: {
    _id: 0,
    name: 1,
    amount: "$funding_rounds.raised_amount",
    year: "$funding_rounds.funded_year"
        }
    }
])
```

На этапе с оператором `$unwind` создается точная копия каждого из документов, которые он получает в качестве входных данных. У всех полей будут одинаковый ключ и значение, за исключением поля `"funding_rounds"`. Вместо того чтобы являться массивом документов `"funding_rounds"`, он будет иметь значение, представляющее собой один документ, соответствующий индивидуальному раунду размещения ценных бумаг:

```
{"name": "Digg", "amount": 8500000, "year": 2006 }
{"name": "Digg", "amount": 2800000, "year": 2005 }
{"name": "Digg", "amount": 28700000, "year": 2008 }
{"name": "Digg", "amount": 5000000, "year": 2011 }
{"name": "Facebook", "amount": 500000, "year": 2004 }
{"name": "Facebook", "amount": 12700000, "year": 2005 }
{"name": "Facebook", "amount": 27500000, "year": 2006 }
{"name": "Facebook", "amount": 240000000, "year": 2007 }
{"name": "Facebook", "amount": 60000000, "year": 2007 }
{"name": "Facebook", "amount": 15000000, "year": 2008 }
{"name": "Facebook", "amount": 100000000, "year": 2008 }
{"name": "Facebook", "amount": 60000000, "year": 2008 }
{"name": "Facebook", "amount": 200000000, "year": 2009 }
{"name": "Facebook", "amount": 210000000, "year": 2010 }
{"name": "Facebook", "amount": 1500000000, "year": 2011 }
```

```
{"name": "Revision3", "amount": 1000000, "year": 2006 }
{"name": "Revision3", "amount": 8000000, "year": 2007 }
...
```

Теперь давайте добавим в наши выходные документы дополнительное поле. При этом мы фактически выявим небольшую проблему, связанную с этим конвейером агрегации:

```
db.companies.aggregate([
    { $match: { "funding_rounds.investments.financial_org.permalink": "greylock" } },
    { $unwind: "$funding_rounds"
    },
    { $project: {
        _id: 0,
        name: 1,
        funder: "$funding_rounds.investments.financial_org.permalink",
        amount: "$funding_rounds.raised_amount",
        year: "$funding_rounds.funded_year"
        }
    }
])
```

При добавлении поля `"funder"` у нас теперь есть значение пути к полю, которое будет обращаться к полю `"investments"` вложенного документа `"funding_rounds"`, получаемому на этапе с оператором `$unwind`, и в случае с финансовой организацией выбирает значение постоянной ссылки. Обратите внимание, что это очень похоже на то, что мы делаем в нашем фильтре соответствий. Давайте посмотрим на результат:

```
{
    "name": "Digg",
    "funder": [
        "greylock",
        "omidyar-network"
    ],
    "amount": 8500000,
    "year": 2006
}
    {
    "name": "Digg",
    "funder": [
        "greylock",
        "omidyar-network",
        "floodgate",
        "sv-angel"
```

```
    ],
    "amount": 2800000,
    "year": 2005
}
    {
    "name": "Digg",
    "funder": [
        "highland-capital-partners",
        "greylock",
        "omidyar-network",
        "svb-financial-group"
    ],
    "amount": 28700000,
    "year": 2008
}
    ...
    {
    "name": "Farecast",
    "funder": [
        "madrona-venture-group",
        "wrf-capital"
    ],
    "amount": 1500000,
    "year": 2004
}
    {
    "name": "Farecast",
    "funder": [
        "greylock",
        "madrona-venture-group",
        "wrf-capital"
    ],
    "amount": 7000000,
    "year": 2005
}
    {
    "name": "Farecast",
    "funder": [
        "greylock",
        "madrona-venture-group",
        "par-capital-management",
        "pinnacle-ventures",
        "sutter-hill-ventures",
        "wrf-capital"
    ],
```

```
    "amount": 12100000,
    "year": 2007
}
```

Чтобы понять то, что мы видим здесь, нам нужно вернуться к нашему документу и взглянуть на поле `"investments"`.

Поле `"funding_rounds.investments"` само по себе является массивом. В каждом раунде размещения ценных бумаг могут участвовать несколько инвесторов, поэтому в поле `"investments"` будет указан каждый из них. Глядя на результаты, как мы изначально видели с полями `"raised_amount"` и `"funded_year"`, теперь мы видим массив для `"funder"`, поскольку `"investments"` – это поле со значением массива.

Еще одна проблема заключается в том, что из-за способа написания нашего конвейера многие документы, которые передаются в этап с оператором `$project`, обозначают раунды размещения ценных бумаг, в которых компания Greylock не участвовала. Это можно увидеть, посмотрев на раунды размещения ценных бумаг компании Farecast. Эта проблема связана с тем, что наш этап с оператором `$match` выбирает все компании, где Greylock участвовала как минимум в одном раунде. Если нас интересуют только те раунды, в которых фактически участвовала компания Greylock, нам нужно найти иной способ фильтрации.

Одна из возможностей – изменить порядок этапов с операторами `$unwind` и `$match`, то есть сначала использовать оператор `$unwind`, а затем `$match`. Это гарантирует, что мы будем сопоставлять только документы, выходящие из этапа с `$unwind`. Но, обдумывая данный подход, быстро становится ясно, что если мы вначале используем этап с оператором `$unwind`, то просканируем всю коллекцию.

Для эффективности нам нужно, чтобы совпадение в нашем конвейере было как можно раньше. Это позволяет, например, фреймворку агрегации использовать индексы. Итак, чтобы выбрать только те раунды размещения ценных бумаг, в которых участвовала компания Greylock, мы можем включить второй этап с оператором `$match`:

```
db.companies.aggregate([
    { $match: { "funding_rounds.investments.financial_org.permalink": "greylock" } },
    { $unwind: "$funding_rounds" },
    { $match: { "funding_rounds.investments.financial_org.permalink": "greylock" } },
    { $project: {
        _id: 0,
        name: 1,
        individualFunder: "$funding_rounds.investments.person.permalink",
        fundingOrganization: "$funding_rounds.investments.financial_org.permalink",
        amount: "$funding_rounds.raised_amount",
        year: "$funding_rounds.funded_year"
```

```
        }
    }
])
```

Этот конвейер сначала отфильтрует компании, в которых Greylock участвовала хотя бы в одном раунде. Затем он использует оператор `$unwind` для раундов размещения ценных бумаг и снова выполнит фильтрацию, таким образом, только те документы, которые представляют раунды, в которых фактически участвовала компания Greylock, будут переданы в этап с `$project`.

Как уже упоминалось в начале этой главы, часто бывает необходимо включить несколько этапов одного типа. Это хороший пример: мы выполняем фильтрацию, чтобы уменьшить количество документов, которые мы рассматриваем изначально, сузив наш набор документов для рассмотрения до тех, где Greylock участвовала как минимум в одном раунде размещения ценных бумаг. Затем, на этапе с оператором `$unwind`, мы получаем ряд документов, представляющих раунды размещения ценных бумаг от компаний, в которых Greylock фактически участвовала, но отдельные раунды, в которых Greylock не участвовала. Мы можем избавиться ото всех раундов, которые нас не интересуют, просто включив еще один фильтр, используя второй этап с оператором `$match`.

Выражения массивов

Теперь давайте обратим наше внимание на выражения массивов. В рамках нашего глубокого погружения мы рассмотрим использование выражений массивов в этапах с оператором `$project`.

Первое выражение, которое мы рассмотрим, – это выражение фильтра. Выражение фильтра выбирает подмножество элементов в массиве на основе критериев фильтра.

Работая снова с нашим набором данных *companies*, мы обратимся к оператору `$match`, используя те же критерии для раундов размещения ценных бумаг, в которых участвовала компания Greylock. Взгляните на поле `rounds` в этом конвейере:

```
db.companies.aggregate([
    { $match: { "funding_rounds.investments.financial_org.permalink": "greylock" } },
    { $project: {
        _id: 0,
        name: 1,
        founded_year: 1,
        rounds: { $filter: {
            input: "$funding_rounds",
            as: "round",
```

```
            cond: { $gte: [
                        "$$round.raised_amount",
                        100000000 ]
                  }
            }
        }
    }
    },
    { $match: { "rounds.investments.financial_org.permalink": "greylock" } },
]).pretty()
```

В поле `rounds` используется выражение фильтра. Оператор `$filter` предназначен для работы с полями массива и определяет параметры, которые мы должны предоставить. Первый параметр `$filter` – это `input`. Для нее мы просто указываем массив. В этом случае мы используем спецификатор пути к полю для определения массива `"funding_rounds"`, обнаруженного в документах нашей коллекции *companies*. Далее мы указываем имя, которое хотели бы использовать для этого массива в остальной части нашего выражения фильтра. Затем, в качестве третьего параметра, нам нужно указать условие. Условие должно предоставлять критерии, используемые для фильтрации любого массива, который мы предоставили в качестве входных данных, выбирая подмножество. В этом случае мы выполняем фильтрацию таким образом, что выбираем только те элементы, для которых значение `"raised_amount"` для `"funding_round"` больше или равно 100 млн.

При указании условия мы использовали операторы с символами для ссылки на переменную, определенную в рамках выражения, с которым мы работаем. Оператор `as` определяет переменную в нашем выражении фильтра. Эта переменная носит название `"round"`, потому что так мы ее пометили, чтобы устранить неоднозначность ссылки на переменную из пути к полю. В этом случае наше выражение сравнения принимает массив из двух значений и возвращает значение `true`, если первое предоставленное значение больше или равно второму.

Теперь давайте рассмотрим, какие документы даст этап `$project`, учитывая этот фильтр. Выходные документы будут иметь поля `"name"`, `"based_year"` и `"rounds"`. Значения `"rounds"` будут массивами, состоящими из элементов, которые соответствуют условию нашего фильтра: полученная сумма превышает 100 000 000 долларов.

В следующем этапе с оператором `$match`, как мы это делали ранее, мы просто отфильтруем входные документы для тех, которые каким-то образом финансировались компанией Greylock. Документы, выводимые этим конвейером, будут выглядеть следующим образом:

```
{
    "name": "Dropbox",
```

```
"founded_year": 2007,
"rounds": [
    {
        "id": 25090,
        "round_code": "b",
        "source_description":
            "Dropbox Raises $250M In Funding, Boasts 45 Million Users",
        "raised_amount": 250000000,
        "raised_currency_code": "USD",
        "funded_year": 2011,
        "investments": [
            {
                "financial_org": {
                    "name": "Index Ventures",
                    "permalink": "index-ventures"
                }
            },
            {
                "financial_org": {
                    "name": "RIT Capital Partners",
                    "permalink": "rit-capital-partners"
                }
            },
            {
                "financial_org": {
                    "name": "Valiant Capital Partners",
                    "permalink": "valiant-capital-partners"
                }
            },
            {
                "financial_org": {
                    "name": "Benchmark",
                    "permalink": "benchmark-2"
                }
            },
            {
                "company": null,
                "financial_org": {
                    "name": "Goldman Sachs",
                    "permalink": "goldman-sachs"
                },
                "person": null
            },
            {
                "financial_org": {
                    "name": "Greylock Partners",
```

```
                        "permalink": "greylock"
                    }
                },
                {
                    "financial_org": {
                        "name": "Institutional Venture Partners",
                        "permalink": "institutional-venture-partners"
                    }
                },
                {
                    "financial_org": {
                        "name": "Sequoia Capital",
                        "permalink": "sequoia-capital"
                    }
                },
                {
                    "financial_org": {
                        "name": "Accel Partners",
                        "permalink": "accel-partners"
                    }
                },
                {
                    "financial_org": {
                        "name": "Glynn Capital Management",
                        "permalink": "glynn-capital-management"
                    }
                },
                {
                    "financial_org": {
                        "name": "SV Angel",
                        "permalink": "sv-angel"
                    }
                }
            ]
        }
    ]
}
```

Только элементы массива `"rounds"`, для которых полученная сумма превышает 100 000 000 долларов, будут проходить через фильтр. В случае с Dropbox есть лишь один раунд, который соответствует этому критерию. Мы обладаем довольно большой гибкостью относительно настройки выражений фильтра, но это базовая форма, которая предоставляет конкретный пример варианта использования для этого конкретного выражения массива.

Далее, давайте посмотрим на оператор элемента массива. Мы продолжим работу с раундами размещения ценных бумаг, но в этом случае мы просто хотим вывести первый и последний раунды. Нам может быть интересно, например, посмотреть, когда произошли эти раунды, или сравнить их суммы. Это можно сделать с помощью арифметических выражений и дат, как мы увидим в следующем разделе.

Оператор `$arrayElemAt` позволяет выбрать элемент в определенной ячейке массива. Приведенный ниже конвейер дает пример использования этого оператора:

```
db.companies.aggregate([
    { $match: { "founded_year": 2010 } },
    { $project: {
        _id: 0,
        name: 1,
        founded_year: 1,
        first_round: { $arrayElemAt: [ "$funding_rounds", 0 ] },
        last_round: { $arrayElemAt: [ "$funding_rounds", -1 ] }
    } }
]).pretty()
```

Обратите внимание на синтаксис использования `$arrayElemAt` в рамках этапа с оператором `$project`. Мы определяем нужное нам поле и в качестве значения указываем документ с оператором `$arrayElemAt` в качестве имени поля и двухэлементным массивом в качестве значения. Первый элемент должен быть путем к полю, которое определяет поле массива, из которого мы хотим сделать выбор. Второй элемент идентифицирует ячейку в нужном нам массиве. Помните, что элементы массива индексируются начиная с нуля.

Во многих случаях длина массива не всегда доступна. Чтобы выбрать ячейку массива, начиная с конца массива, используйте отрицательные целые числа. Последний элемент в массиве обозначен как –1.

Простой выходной документ для этого конвейера агрегации будет выглядеть следующим образом:

```
{
    "name": "vufind",
    "founded_year": 2010,
    "first_round": {
        "id": 19876,
        "round_code": "angel",
        "source_url": "",
        "source_description": "",
        "raised_amount": 250000,
        "raised_currency_code": "USD",
```

```
            "funded_year": 2010,
            "funded_month": 9,
            "funded_day": 1,
            "investments": []
        },
        "last_round": {
            "id": 57219,
            "round_code": "seed",
            "source_url": "",
            "source_description": "",
            "raised_amount": 500000,
            "raised_currency_code": "USD",
            "funded_year": 2012,
            "funded_month": 7,
            "funded_day": 1,
            "investments": []
        }
    }
```

С оператором `$arrayElemAt` связан оператор `$slice`. Он позволяет возвращать не один, а несколько элементов из массива в последовательности, начиная с определенного индекса:

```
db.companies.aggregate([
    { $match: { "founded_year": 2010 } },
    { $project: {
        _id: 0,
        name: 1,
        founded_year: 1,
        early_rounds: { $slice: [ "$funding_rounds", 1, 3 ] }
    } }
]).pretty()
```

Здесь, опять же, у нас идет массив `funding_rounds`. Мы начинаем с индекса 1 и берем три элемента из массива. Возможно, мы знаем, что в этом наборе данных первый раунд размещения ценных бумаг не так уж интересен, или мы просто хотим получить несколько ранних раундов, но не самый первый.

Фильтрация и выбор отдельных элементов или фрагментов массивов – одни из наиболее распространенных операций, которые мы должны выполнять над массивами. Однако, вероятно, наиболее распространенной является определение размера или длины массива. Для этого можно использовать оператор `$size`:

```
db.companies.aggregate([
    { $match: { "founded_year": 2004 } },
    { $project: {
        _id: 0,
        name: 1,
        founded_year: 1,
        total_rounds: { $size: "$funding_rounds" }
    } }
]).pretty()
```

При использовании в этапе с `$project` оператор `$size` просто предоставляет значение, равное количеству элементов в массиве.

В этом разделе мы рассмотрели некоторые из наиболее распространенных выражений массивов. Их намного больше, и их список растет с каждым выпуском. Пожалуйста, обратитесь к краткому справочнику по конвейерам агрегации из документации по MongoDB (https://oreil.ly/ZtUES), чтобы получить сводку всех доступных выражений.

Аккумуляторы

На данный момент мы рассмотрели несколько различных типов выражений. Далее давайте посмотрим, какие аккумуляторы может предложить фреймворк агрегации. По сути, аккумуляторы – еще один тип выражений, но мы рассматриваем их в их собственном классе, поскольку они вычисляют значения из значений полей, находящихся в нескольких документах.

Аккумуляторы, которые предоставляет фреймворк агрегации, позволяют выполнять такие операции, как суммирование всех значений в определенном поле (`$sum`), вычисление среднего значения (`$avg`) и т. д. Мы также считаем, что `$first` и `$last` являются аккумуляторами, поскольку они учитывают значения во всех документах, которые проходят через этап, в котором они используются. `$max` и `$min` – еще два примера аккумуляторов, которые рассматривают поток документов и сохраняют только одно из значений, которые они видят. `$mergeObjects` можно использовать для объединения нескольких документов в один.

У нас также есть аккумуляторы для массивов. Мы можем использовать оператор `$push` при работе со значениями, когда документы проходят через этап конвейера. `$addToSet` очень похож на `$push`, за исключением того, что гарантирует, что повторяющиеся значения не будут включены в получившийся массив.

Также есть несколько выражений для расчета описательной статистики – например, для расчета стандартного отклонения выборки и совокупности. Оба работают с потоком документов, которые проходят через этап конвейера.

До выхода MongoDB версии 3.2 аккумуляторы были доступны только на этапе `$group`. В MongoDB версии 3.2 появилась возможность доступа к подмножеству аккумуляторов в рамках этапа с оператором `$project`. Основное различие между аккумуляторами на групповом этапе и этапе с оператором `$project` заключается в том, что на этапе с оператором `$project` такие аккумуляторы, как `$sum` и `$avg`, должны работать с массивами в одном документе, в то время как аккумуляторы на групповом этапе, как мы увидим в следующем разделе, предоставляют возможность выполнять вычисления значений в нескольких документах.

Это был краткий обзор аккумуляторов, чтобы предоставить вам некоторый контекст и подготовить почву для подробного изучения примеров.

Использование аккумуляторов в этапах с $project

Начнем с примера использования аккумулятора на этапе `$project`. Обратите внимание, что наш этап с оператором `$match` фильтрует документы, которые содержат поле `"funding_rounds"`, для которого массив `funding_rounds` не пуст:

```
db.companies.aggregate([
    { $match: { "funding_rounds": { $exists: true, $ne: [ ]} } },
    { $project: {
        _id: 0,
        name: 1,
        largest_round: { $max: "$funding_rounds.raised_amount" }
    } }
])
```

Поскольку значение `$funding_rounds` является массивом в каждом документе компании, мы можем использовать аккумулятор. Помните, что на этапах с оператором `$project` аккумуляторы должны работать с полем, имеющим значение массива. В этом случае мы можем сделать здесь что-то очень крутое. Мы легко определяем наибольшее значение в массиве, обращаясь к вложенному документу в этом массиве и проецируя максимальное значение в выходных документах:

```
{ "name" : "Wetpaint", "largest_round" : 25000000 }
{ "name" : "Digg", "largest_round" : 28700000 }
{ "name" : "Facebook", "largest_round" : 1500000000 }
{ "name" : "Omnidrive", "largest_round" : 800000 }
{ "name" : "Geni", "largest_round" : 10000000 }
{ "name" : "Twitter", "largest_round" : 400000000 }
{ "name" : "StumbleUpon", "largest_round" : 17000000 }
{ "name" : "Gizmoz", "largest_round" : 6500000 }
{ "name" : "Scribd", "largest_round" : 13000000 }
```

```
{ "name" : "Slacker", "largest_round" : 40000000 }
{ "name" : "Lala", "largest_round" : 20000000 }
{ "name" : "eBay", "largest_round" : 6700000 }
{ "name" : "MeetMoi", "largest_round" : 2575000 }
{ "name" : "Joost", "largest_round" : 45000000 }
{ "name" : "Babelgum", "largest_round" : 13200000 }
{ "name" : "Plaxo", "largest_round" : 9000000 }
{ "name" : "Cisco", "largest_round" : 2500000 }
{ "name" : "Yahoo!", "largest_round" : 4800000 }
{ "name" : "Powerset", "largest_round" : 12500000 }
{ "name" : "Technorati", "largest_round" : 10520000 }
...
```

В качестве другого примера давайте воспользуемся аккумулятором `$sum` для расчета общего финансирования каждой компании в нашей коллекции:

```
db.companies.aggregate([
    { $match: { "funding_rounds": { $exists: true, $ne: [ ]} } },
    { $project: {
        _id: 0,
        name: 1,
        total_funding: { $sum: "$funding_rounds.raised_amount" }
    } }
])
```

Это только самая малость того, что можно сделать, используя аккумуляторы на этапе с оператором `$project`. Опять же, предлагаем вам ознакомиться с кратким справочником по конвейеру агрегации из документации MongoDB (https://oreil.ly/SZiFx), где представлен полный обзор доступных выражений аккумулятора.

Знакомство с группировкой

Традиционно аккумуляторы были областью этапа с оператором `$group` в фреймворке агрегации MongoDB. Данный этап выполняет функцию, аналогичную SQL-команде `GROUP BY`. На этом этапе мы можем сгруппировать значения из нескольких документов и выполнять с ними операции агрегации определенного типа, например вычислять среднее. Давайте посмотрим на пример:

```
db.companies.aggregate([
    { $group: {
        _id: { founded_year: "$founded_year" },
        average_number_of_employees: { $avg: "$number_of_employees" }
    } },
```

```
    { $sort: { average_number_of_employees: -1 } }
])
```

Здесь мы используем этап с оператором `$group`, чтобы сгруппировать все компании на основе года, в котором они были созданы, а затем рассчитать среднее число сотрудников за каждый год.

Вывод этого конвейера выглядит примерно так:

```
{ "_id" : { "founded_year" : 1847 }, "average_number_of_employees" : 405000 }
{ "_id" : { "founded_year" : 1896 }, "average_number_of_employees" : 388000 }
{ "_id" : { "founded_year" : 1933 }, "average_number_of_employees" : 320000 }
{ "_id" : { "founded_year" : 1915 }, "average_number_of_employees" : 186000 }
{ "_id" : { "founded_year" : 1903 }, "average_number_of_employees" : 171000 }
{ "_id" : { "founded_year" : 1865 }, "average_number_of_employees" : 125000 }
{ "_id" : { "founded_year" : 1921 }, "average_number_of_employees" : 107000 }
{ "_id" : { "founded_year" : 1835 }, "average_number_of_employees" : 100000 }
{ "_id" : { "founded_year" : 1952 }, "average_number_of_employees" : 92900 }
{ "_id" : { "founded_year" : 1946 }, "average_number_of_employees" : 91500 }
{ "_id" : { "founded_year" : 1947 }, "average_number_of_employees" : 88510.5 }
{ "_id" : { "founded_year" : 1898 }, "average_number_of_employees" : 80000 }
{ "_id" : { "founded_year" : 1968 }, "average_number_of_employees" : 73550 }
{ "_id" : { "founded_year" : 1957 }, "average_number_of_employees" : 70055 }
{ "_id" : { "founded_year" : 1969 }, "average_number_of_employees" : 67635.1 }
{ "_id" : { "founded_year" : 1928 }, "average_number_of_employees" : 51000 }
{ "_id" : { "founded_year" : 1963 }, "average_number_of_employees" : 50503 }
{ "_id" : { "founded_year" : 1959 }, "average_number_of_employees" : 47432.5 }
{ "_id" : { "founded_year" : 1902 }, "average_number_of_employees" : 41171.5 }
{ "_id" : { "founded_year" : 1887 }, "average_number_of_employees" : 35000 }
...
```

Вывод включает в себя документы, в которых в качестве значения `"_id"` указан документ, а затем отчет о среднем числе сотрудников. Этот тип анализа мы могли бы сделать в качестве первого шага в оценке корреляции между годом, в котором компания была основана, и ее ростом, возможно, с учетом возраста компании.

Как видно, собранный нами конвейер состоит из двух этапов: `$group` и `$sort`.

Основой этапа `$group` является поле `"_id"`, которое мы указываем как часть документа.

Это значение самого оператора `$group`, использующего очень строгую интерпретацию.

Мы используем это поле, чтобы определить, что использует этап `$group` для организации документов, которые он видит. Поскольку этот этап идет первым, команда `aggregate` пропустит через него все документы в коллек-

ции *companies*. Этап примет каждый документ, у которого одно и то же значение поля `"founded_year"`, и будет рассматривать их как единую группу. При построении значения этого поля этот этап будет использовать оператор `$avg` для расчета среднего числа сотрудников для всех компаний с одинаковым значением `"founded_year"`.

Можно смотреть на это так. Каждый раз, когда этап `$group` встречает документ с определенным годом основания, он добавляет значение `"number_of_employees"` из этого документа к промежуточной сумме числа сотрудников и добавляет единицу к количеству документов, просмотренных за этот год. После того как все документы проходят через этап `$group`, он может рассчитать среднее значение с использованием этой промежуточной суммы и числа для каждой группы документов, которые он определил, основываясь на годе основания компании.

В конце этого конвейера мы сортируем документы в порядке убывания по среднему числу сотрудников (`average_number_of_employees`).

Рассмотрим еще один пример. Одно из полей, которое мы еще не рассматривали в наборе данных *companies*, – это `relationships`. Поле `relationships` отображается в документах в следующей форме:

```
{
    "_id": "52cdef7c4bab8bd675297d8e",
    "name": "Facebook",
    "permalink": "facebook",
    "category_code": "social",
    "founded_year": 2004,
    ...
    "relationships": [
        {
            "is_past": false,
            "title": "Founder and CEO, Board Of Directors",
            "person": {
                "first_name": "Mark",
                "last_name": "Zuckerberg",
                "permalink": "mark-zuckerberg"
            }
        },
        {
            "is_past": true,
            "title": "CFO",
            "person": {
                "first_name": "David",
                "last_name": "Ebersman",
                "permalink": "david-ebersman"
            }
```

```
        },
    ...
    ],
    "funding_rounds": [
    ...
    {
            "id": 4,
            "round_code": "b",
            "source_url": "http://www.facebook.com/press/info.php?factsheet",
            "source_description": "Facebook Funding",
            "raised_amount": 27500000,
            "raised_currency_code": "USD",
            "funded_year": 2006,
            "funded_month": 4,
            "funded_day": 1,
            "investments": [
                {
                    "company": null,
                    "financial_org": {
                        "name": "Greylock Partners",
                        "permalink": "greylock"
                    },
                    "person": null
                },
                {
                    "company": null,
                    "financial_org": {
                        "name": "Meritech Capital Partners",
                        "permalink": "meritech-capital-partners"
                    },
                    "person": null
                },
                {
                    "company": null,
                    "financial_org": {
                        "name": "Founders Fund",
                        "permalink": "founders-fund"
                    },
                    "person": null
                },
                {
                    "company": null,
                    "financial_org": {
                        "name": "SV Angel",
                        "permalink": "sv-angel"
```

```
                    },
                    "person": null
                }
            ]
        },
        ...
        "ipo": {
            "valuation_amount": NumberLong("104000000000"),
            "valuation_currency_code": "USD",
            "pub_year": 2012,
            "pub_month": 5,
            "pub_day": 18,
            "stock_symbol": "NASDAQ:FB"
        },
        ...
    }
```

Поле `relationships` дает нам возможность найти людей, которые так или иначе связаны с относительно большим количеством компаний.

Взглянем на этот код:

```
db.companies.aggregate( [
    { $match: { "relationships.person": { $ne: null } } },
    { $project: { relationships: 1, _id: 0 } },
    { $unwind: "$relationships" },
    { $group: {
        _id: "$relationships.person",
        count: { $sum: 1 }
    } },
    { $sort: { count: -1 } }
]).pretty()
```

Первым идет этап с оператором `$match`. Если мы посмотрим на пример с документом Facebook, то увидим, как структурированы отношения, и поймем, для чего это. Мы используем фильтрацию для всех отношений, где значение `"person"` не равно нулю. Затем идет этап `$project`. В следующий этап конвейера, `$unwind`, мы передадим только отношения. Мы используем оператор `$unwind`, чтобы каждое отношение в массиве доходило до этапа `$group`, который следует далее. Там мы используем путь к полю, чтобы идентифицировать человека в каждом документе `"relationships"`. Все документы с одинаковым значением `"person"` будут сгруппированы. Как мы видели ранее, для документа совершенно нормально быть значением, вокруг которого мы делаем группировку. Таким образом, каждое совпадение с документом по имени, фамилии и постоянной ссылке для человека будет сгруппировано. Мы используем оператор `$sum`, чтобы подсчитать число от-

ношений, в которых участвовал каждый человек. Наконец, мы выполняем сортировку в порядке убывания. Вывод этого конвейера выглядит примерно так:

```
{
    "_id": {
        "first_name": "Tim",
        "last_name": "Hanlon",
        "permalink": "tim-hanlon"
    },
    "count": 28
}
    {
    "_id": {
        "first_name": "Pejman",
        "last_name": "Nozad",
        "permalink": "pejman-nozad"
    },
    "count": 24
}
    {
    "_id": {
        "first_name": "David S.",
        "last_name": "Rose",
        "permalink": "david-s-rose"
    },
    "count": 24
}
    {
    "_id": {
        "first_name": "Saul",
        "last_name": "Klein",
        "permalink": "saul-klein"
    },
    "count": 24
}
...
```

Тим Хэнлон – человек, который участвовал в большинстве отношений с компаниями из этой коллекции. Возможно, г-н Хэнлон имел связи с 28 компаниями, но мы не можем знать это наверняка, потому что также возможно, что у него были отношения с одной или несколькими компаниями, у каждой из которых свое название. Данный пример иллюстрирует один очень важный момент касательно конвейеров агрегации: убедитесь, что вы полностью понимаете, с чем вы работаете, когда делаете вычис-

ления, особенно когда вычисляете значения агрегации с использованием каких-либо выражений-аккумуляторов.

В этом случае можно сказать, что Тим Хэнлон встречается 28 раз в документах `"relationships"` по всем компаниям в нашей коллекции. Нужно было бы покопаться немного поглубже, чтобы точно узнать, со сколькими компаниями он был связан, но мы оставим создание этого конвейера вам в качестве упражнения.

Поле _id в этапах $group

Прежде чем продолжить обсуждение этапа `$group`, поговорим еще немного о поле `_id` и рассмотрим ряд передовых рекомендаций по созданию значений для этого поля на этапах агрегирования `$group`. Мы разберем несколько примеров, иллюстрирующих несколько различных способов группировки документов. В качестве нашего первого примера рассмотрим этот конвейер:

```
db.companies.aggregate([
    { $match: { founded_year: { $gte: 2013 } } },
    { $group: {
        _id: { founded_year: "$founded_year"},
        companies: { $push: "$name" }
    } },
    { $sort: { "_id.founded_year": 1 } }
]).pretty()
```

Вывод данного конвейера выглядит примерно так:

```
{
    "_id": {
        "founded_year": 2013
    },
    "companies": [
        "Fixya",
        "Wamba",
        "Advaliant",
        "Fluc",
        "iBazar",
        "Gimigo",
        "SEOGroup",
        "Clowdy",
        "WhosCall",
        "Pikk",
        "Tongxue",
        "Shopseen",
        "VistaGen Therapeutics"
```

```
    ]
}
...
```

В этом выводе у нас есть документы с двумя полями: `"_id"` и `"companies"`. Каждый из этих документов содержит список компаний, основанных в том или ином году (`"founded_year"`), а `"companies"` – это массив названий компаний.

Обратите внимание, как мы построили поле `"_id"` на этапе `$group`. Почему бы просто не указать год основания, вместо того чтобы помещать его в документ с полем, помеченным как `"founded_year"`. Причина, по которой мы этого не делаем, заключается в том, что если мы не помечаем значение группы, это не означает, что мы делаем группировку по году, в котором была основана компания. Чтобы избежать путаницы, лучше четко обозначить значения, по которым мы группируем.

В некоторых случаях может потребоваться использовать другой подход, при котором значение `_id` является документом, состоящим из нескольких полей. В этом случае мы фактически группируем документы на основе года основания и кода категории:

```
db.companies.aggregate([
    { $match: { founded_year: { $gte: 2010 } } },
    { $group: {
        _id: { founded_year: "$founded_year", category_code: "$category_code" },
        companies: { $push: "$name" }
    } },
    { $sort: { "_id.founded_year": 1 } }
]).pretty() ,
```

Совершенно нормально использовать документы с несколькими полями в качестве значения `_id` в этапах с оператором `$group`. В других случаях может также потребоваться сделать что-то вроде этого:

```
db.companies.aggregate([
    { $group: {
        _id: { ipo_year: "$ipo.pub_year" },
        companies: { $push: "$name" }
    } },
    { $sort: { "_id.ipo_year": 1 } }
).pretty()
```

В этом случае мы группируем документы на основе года, в котором компании провели первоначальное публичное предложение акций, и этот год фактически является полем вложенного документа. Обычной практикой является использование путей к полям, которые обращаются ко вложен-

ным документам, в качестве значения для группировки в этапе с оператором `$group`. В этом случае вывод будет выглядеть следующим образом:

```
{
    "_id": {
        "ipo_year": 1999
    },
    "companies": [
        "Akamai Technologies",
        "TiVo",
        "XO Group",
        "Nvidia",
        "Blackberry",
        "Blue Coat Systems",
        "Red Hat",
        "Brocade Communications Systems",
        "Juniper Networks",
        "F5 Networks",
        "Informatica",
        "Iron Mountain",
        "Perficient",
        "Sitestar",
        "Oxford Instruments"
    ]
}
```

Обратите внимание, что в примерах, приведенных в этом разделе, используется аккумулятор, который мы не видели раньше: `$push`. По мере того как этап `$group` обрабатывает документы в своем входном потоке, выражение `$push` добавит полученное значение в массив, который он создает на протяжении всего цикла. В случае с предыдущим конвейером этап `$group` создает массив, состоящий из названий компаний.

Наш последний пример – тот, что мы уже видели, но мы включили его сюда для полноты картины:

```
db.companies.aggregate( [
    { $match: { "relationships.person": { $ne: null } } },
    { $project: { relationships: 1, _id: 0 } },
    { $unwind: "$relationships" },
    { $group: {
        _id: "$relationships.person",
        count: { $sum: 1 }
    } },
    { $sort: { count: -1 } }
])
```

В предыдущем примере, в котором мы делали группировку по году IPO, мы использовали путь к полю, который разрешался в скалярное значение – год IPO. В этом случае наш путь к полю разрешается в документ, содержащий три поля: `"first_name"`, `"last_name"` и `"permalink"`.

Это показывает, что этап с оператором `$group` поддерживает группировку по значениям документа.

Теперь вы видели несколько способов построения значений `_id` на групповых этапах.

В целом учитывайте, что здесь мы хотим убедиться, что в нашем выводе семантика значения `_id` ясна.

Сравнение $group и $project

Чтобы завершить обсуждение этапа агрегации с оператором `$group`, мы рассмотрим несколько дополнительных аккумуляторов, которые недоступны на этапе `$project`. Это должно заставить вас задуматься о том, что можно сделать на этапе `$project` в отношении аккумуляторов и что можно сделать на этапе `$group`. В качестве примера рассмотрим этот запрос на агрегацию:

```
db.companies.aggregate([
    { $match: { funding_rounds: { $ne: [ ] } } },
    { $unwind: "$funding_rounds" },
    { $sort: { "funding_rounds.funded_year": 1,
        "funding_rounds.funded_month": 1,
        "funding_rounds.funded_day": 1 } },
    { $group: {
        _id: { company: "$name" },
        funding: {
        $push: {
            amount: "$funding_rounds.raised_amount",
            year: "$funding_rounds.funded_year"
        } }
    } },
] ).pretty()
```

Здесь мы начинаем с фильтрации для документов, у которых массив `funding_rounds` не является пустым. Затем мы используем оператор `$unwind`. Поэтому этапы `$sort` и `$group` будут видеть по одному документу для каждого элемента массива `funding_rounds` для каждой компании.

Этап `$sort` в этом конвейере выполняет сортировку сначала по годам, затем по месяцам, а потом по дням в порядке возрастания. Это означает, что на этом этапе сначала будут выводиться самые старые раунды размещения ценных бумаг. Как вы знаете из главы 5, можно поддерживать данный тип сортировки с помощью составного индекса.

На этапе `$group`, который следует за `$sort`, мы выполняем группировку по названию компании и используем оператор `$push` для построения отсортированного массива раундов. Массив `funding_rounds` будет отсортирован для каждой компании, потому что мы отсортировали все раунды в глобальном масштабе на этапе `$sort`.

Вывод документов из этого конвейера будет выглядеть примерно так:

```
{
    "_id": {
        "company": "Green Apple Media"
    },
    "funding": [
        {
            "amount": 30000000,
            "year": 2013
        },
        {
            "amount": 100000000,
            "year": 2013
        },
        {
            "amount": 2000000,
            "year": 2013
        }
    ]
}
```

В этом конвейере с помощью этапа `$push` мы аккумулируем массив. В этом случае мы указали наше выражение `$push`, чтобы оно добавляло документы в конец массива. Поскольку раунды размещения ценных бумаг расположены в хронологическом порядке, размещение документов в конце массива гарантирует, что суммы финансирования для каждой компании будут отсортированы в хронологическом порядке.

Выражения `$push` работают только на этапах `$group`. Это связано с тем, что эти этапы предназначены для приема входного потока документов и накопления значений путем обработки каждого документа по очереди. С другой стороны, этапы `$project` работают с каждым документом в своем входном потоке индивидуально.

Давайте посмотрим еще на один пример. Это немного подлиннее, но основывается на предыдущем примере:

```
db.companies.aggregate([
    { $match: { funding_rounds: { $exists: true, $ne: [ ] } } },
    { $unwind: "$funding_rounds" },
    { $sort: { "funding_rounds.funded_year": 1,
        "funding_rounds.funded_month": 1,
```

```
        "funding_rounds.funded_day": 1 } },
    { $group: {
        _id: { company: "$name" },
        first_round: { $first: "$funding_rounds" },
        last_round: { $last: "$funding_rounds" },
        num_rounds: { $sum: 1 },
        total_raised: { $sum: "$funding_rounds.raised_amount" }
        } },
    { $project: {
        _id: 0,
        company: "$_id.company",
        first_round: {
        amount: "$first_round.raised_amount",
        article: "$first_round.source_url",
        year: "$first_round.funded_year"
    },
        last_round: {
        amount: "$last_round.raised_amount",
        article: "$last_round.source_url",
        year: "$last_round.funded_year"
    },
        num_rounds: 1,
        total_raised: 1,
    } },
    { $sort: { total_raised: -1 } }
]).pretty()
```

И снова мы применяем оператор `$unwind` и выполняем сортировку в хронологическом порядке. Однако в этом случае, вместо того чтобы аккумулировать массив записей, каждая из которых представляет собой одиночный `funding_rounds`, мы используем два аккумулятора, которые мы еще не видели в действии: `$first` и `$last`. Выражение `$first` просто сохраняет первое значение, которое проходит через входной поток этапа. Выражение `$last` просто отслеживает значения, которые проходят через этап `$group`, и привязывается к последнему.

Как и в случае с `$push`, мы не можем использовать аккумуляторы `$first` и `$last` на этапах проекта, потому что, опять же, этапы `$project` не предназначены для аккумулирования значений на основе нескольких документов, проходящих через них потоком. Скорее, они предназначены для изменения формы документов по отдельности.

В дополнение к `$first` и `$last` в этом примере мы также используем оператор `$sum` для вычисления общего числа раундов размещения ценных бумаг. Для этого выражения мы можем просто указать значение, 1. Такое выражение `$sum` служит для подсчета числа документов, которые он видит в каждой группировке.

Наконец, этот конвейер включает в себя довольно сложный этап `$project`. Тем не менее все, что он действительно осуществляет, – это делает вывод более красивым. Вместо того чтобы показывать значения `first_round` или целые документы для первого и последнего раундов, этот этап создает сводку. Обратите внимание, что он поддерживает хорошую семантику, потому что каждое значение четко обозначено. Для `first_round` мы создадим простой вложенный документ, который содержит только необходимые данные о сумме, статье и годе, извлекая эти значения из исходного документа `first_round`, который будет значением `$first_round`. Этап `$project` делает нечто похожее для `$last_round`. В конце данный этап просто передает в выходные документы значения `num_rounds` и `total_raised` для документов, которые он получает в своем входном потоке.

Вывод этого конвейера выглядит примерно так:

```
{
    "first_round": {
        "amount": 7500000,
        "article": "http://www.teslamotors.com/display_data/pressguild.swf",
        "year": 2004
    },
    "last_round": {
        "amount": 10000000,
        "article": "http://www.bizjournals.com/sanfrancisco/news/2012/10/10/
                    tesla-motors-to-get-10-million-from.html",
        "year": 2012
    },
    "num_rounds": 11,
    "total_raised": 823000000,
    "company": "Tesla Motors"
}
```

И на этом мы завершаем обзор этапа `$group`.

Запись результатов конвейера агрегации в коллекцию

Существует два этапа, `$out` и `$merge`, которые могут записывать документы, полученные из конвейера агрегации, в коллекцию. Вы можете использовать только один из этих этапов, и он должен быть последним этапом конвейера. `$merge` появился в MongoDB версии 4.2 и является предпочтительным этапом для выполнения записи в коллекцию, если она доступна.

`$out` имеет некоторые ограничения: он может вести запись только в одну и ту же базу данных, он перезаписывает любую существующую коллекцию, если она есть, и не может вести запись в разделенную коллекцию

(т. е. разнесенную по разным серверам). `$merge` может вести запись в любую базу данных и коллекцию, независимо от того, разделена она или нет, а также может включать результаты (вставка новых документов, слияние с существующими документами, сбой операции, сохранение существующих документов или обработка всех документов с пользовательским обновлением) при работе с существующей коллекцией. Но реальным преимуществом использования `$merge` является тот факт, что он может создавать материализованные представления по требованию, где содержимое выходной коллекции постепенно обновляется при запуске конвейера.

В этой главе мы рассмотрели ряд различных аккумуляторов, некоторые из которых доступны на этапе `$project`, а также рассказали, когда использовать `$group`, а когда `$project` при рассмотрении различных аккумуляторов. Далее мы поговорим о транзакциях в MongoDB.

Глава 8

Транзакции

Транзакции – это логические группы обработки в базе данных, и каждая группа или транзакция может содержать одну или несколько операций, таких как чтение и/или запись в нескольких документах. MongoDB поддерживает ACID-совместимые транзакции для нескольких операций, коллекций, баз данных, документов и шардов. В этой главе мы познакомимся с транзакциями, определим, что значит для базы данных акроним ACID, обратим внимание на то, как использовать их в своих приложениях, и дадим советы по настройке транзакций в MongoDB. Мы расскажем:

- что такое транзакция;
- как использовать транзакции;
- как настроить ограничения транзакций для своего приложения.

Знакомство с транзакциями

Как уже было упомянуто выше, транзакция – это логическая единица обработки в базе данных, которая включает в себя одну или несколько операций базы данных. Это могут быть операции чтения или записи. Существуют ситуации, когда вашему приложению может потребоваться чтение и запись в несколько документов (в одной или нескольких коллекциях) как часть этой логической единицы обработки. Важным аспектом транзакции является то, что она никогда не завершается частично – она либо успешна, либо неудачна.

Чтобы использовать транзакции, нужно выполнять развертывание в MongoDB версии 4.2 или более поздней, а драйверы MongoDB нужно обновить до версии 4.2 или более поздней. MongoDB предоставляет справочную страницу по совместимости драйверов (https://oreil.ly/Oe9NE), которую можно использовать, чтобы убедиться в совместимости версии драйвера MongoDB.

Определение ACID

ACID – принятый набор свойств, которым должна соответствовать транзакция, чтобы быть «настоящей». ACID представляет собой аббревиатуру, образованную из четырех слов: Atomicity (атомарность), Consistency (согласованность), Isolation (изолированность) и Durability (долговечность). ACID-транзакции гарантируют достоверность ваших данных и состояния вашей базы данных даже в случае сбоя питания или других ошибок.

Атомарность гарантирует, что все операции внутри транзакции будут либо выполнены, либо не будет выполнено ни одной. Транзакцию нельзя применить частично; либо она фиксируется, либо идет отмена.

Согласованность гарантирует, что в случае успешного выполнения транзакции база данных перейдет из одного согласованного состояния в следующее согласованное состояние.

Изолированность – свойство, которое позволяет одновременно выполнять несколько транзакций в вашей базе данных. Это гарантирует, что транзакция не будет видеть частичные результаты какой-либо другой транзакции, а это означает, что у нескольких параллельных транзакций будут те же результаты, как и в случае, если бы мы выполняли каждую транзакцию последовательно.

Долговечность гарантирует, что при фиксации транзакции все данные будут сохранены даже в случае сбоя системы.

Считается, что база данных является ACID-совместимой, если она обеспечивает соблюдение всех этих свойств и обработку только успешных транзакций. В ситуациях, когда сбой происходит до завершения транзакции, соответствие ACID гарантирует отсутствие изменения данных.

MongoDB – распределенная СУБД с ACID-совместимыми транзакциями между наборами реплик и/или шарды. Сетевой уровень добавляет дополнительный уровень сложности. Группа инженеров в MongoDB предоставила несколько видеороликов по методу школярного обучения (https://www.mongodb.com/transactions), в которых описывается, как они реализовывали необходимые функции для поддержки ACID-транзакций.

Как использовать транзакции

MongoDB предоставляет два типа API для использования транзакций. Первый – это синтаксис, аналогичный реляционным СУБД (например, `start_transaction` и `commit_transaction`), именуемый базовым API, а второй – API обратного вызова, который является рекомендуемым подходом к использованию транзакций.

Базовый API не предоставляет кода повторного выполнения операции для большинства ошибок и требует от разработчика писать код для операций, функции фиксации транзакции и любой необходимый код повторного выполнения операции и ошибок.

API обратного вызова предоставляет единственную функцию, которая оборачивает бóльшую степень функциональности по сравнению с основным API, включая запуск транзакции, связанной с указанным логическим сеансом, выполнение функции, предоставляемой в качестве функции обратного вызова, и затем фиксацию транзакции (или прерывание при возникновении ошибки). Эта функция также включает в себя код повторного выполнения операции, который обрабатывает ошибки фиксации. API обратного вызова был добавлен в MongoDB версии 4.2, чтобы упростить разработку приложений с транзакциями, а также упростить добавление кода повторного выполнения операции для обработки любых ошибок транзакций.

В обоих API разработчик отвечает за запуск логической сессии, которая будет использоваться транзакцией. Оба API требуют, чтобы операции в транзакции были связаны с определенной логической сессией (т. е. передача сеанса в каждую операцию). Логическая сессия в MongoDB отслеживает время и последовательность операций в контексте всего развертывания MongoDB. Логическая сессия или сеанс на стороне сервера является частью базовой структуры, используемой клиентскими сессиями для поддержки повторяющихся записей и причинной согласованности в MongoDB – обе эти функции были добавлены в MongoDB версии 3.6 как часть основы, необходимой для поддержки транзакций. Конкретная последовательность операций чтения и записи, которые имеют причинно-следственную связь, отраженную в их порядке, определяется как причинно-согласованный клиентский сеанс в MongoDB. Клиентский сеанс запускается приложением и используется для взаимодействия с сеансом на стороне сервера.

В 2019 году шесть старших инженеров из MongoDB опубликовали на конференции SIGMOD 2019 документ, озаглавленный «Реализация логических часов и причинной согласованности в рамках всего кластера в MongoDB» (https://oreil.ly/lFLvm)[1]. В нем предоставлено подробное техническое описание механизма логических сессий и причинной согласованности в MongoDB. В документе разобрана работа над многопрофильным, многолетним инженерным проектом. Она включала в себя изменение аспектов уровня хранения, добавление нового согласованного протокола репликации, изменение архитектуры шардинга, рефакторинг кластерных метаданных шардинга и добавление глобальных логических часов. Эти изменения обеспечивают основу, необходимую для базы данных, прежде чем можно будет добавить ACID-совместимые транзакции.

Сложность и дополнительное кодирование, требуемое в приложениях, являются основными причинами, по которым рекомендуется использовать API обратного вызова вместо основного API. Различия между этими API приведены в табл. 8.1.

[1] Его авторы – Миша Тюленев, штатный инженер-программист по шардингу; Энди Шверин, вице-президент по распределенным системам; Ася Камски, главный продакт-менеджер по распределенным системам; Рэндольф Тэн, старший инженер-программист по шардингу; Элисон Кабрал, продакт-менеджер по распределенным системам; и Джек Малроу, инженер по шардингу.

Таблица 8.1. Сравнение базового API и API обратного вызова

Базовый API	API обратного вызова
Требует явного вызова для запуска транзакции и ее фиксации	Запускает транзакцию, выполняет указанные операции и фиксацию (или прерывает работу при возникновении ошибки)
Не включает логику обработки ошибок TransientTransactionError и UnknownTransactionCommitResult, вместо этого обеспечивая гибкость, позволяющую включить настраиваемую обработку этих ошибок	Автоматически включает логику обработки ошибок TransientTransactionError и UnknownTransactionCommitResult
Требует явного логического сеанса для передачи его в API для конкретной транзакции	Требует явного логического сеанса для передачи его в API для конкретной транзакции

Чтобы понять различия между двумя этими API, можно сравнить их, используя простой пример транзакции для сайта электронной коммерции, где размещается заказ и соответствующие товары удаляются из имеющихся запасов по мере их продажи. Это включает в себя два документа в разных коллекциях в одной транзакции. Вот две операции, которые будут основой нашего примера транзакции:

```
orders.insert_one({"sku": "abc123", "qty": 100}, session=session)
inventory.update_one({"sku": "abc123", "qty": {"$gte": 100}},
                     {"$inc": {"qty": -100}}, session=session)
```

Во-первых, давайте посмотрим, как можно использовать базовый API в Python для нашего примера транзакции. Обе операции транзакции выделены в шаге 1 ниже:

```
# Определяем uriString, используя формат соединения DNS Seedlist для соединения;
uri = 'mongodb+srv://server.example.com/'
client = MongoClient(uriString)

my_wc_majority = WriteConcern('majority', wtimeout=1000)

# Предварительное условие / Шаг 0:Создаем коллекции, если их еще не существует;
# Операции CRUD в транзакциях должны быть для существующих коллекций;

client.get_database( "webshop",
                     write_concern=my_wc_majority).orders.insert_one({"sku":
                     "abc123", "qty":0})
client.get_database( "webshop",
                     write_concern=my_wc_majority).inventory.insert_one(
                     {"sku": "abc123", "qty": 1000})
```

```
# Шаг 1: Определяем операции и их последовательность в рамках транзакции;
def update_orders_and_inventory(my_session):
    orders = session.client.webshop.orders
    inventory = session.client.webshop.inventory
    with session.start_transaction(
            read_concern=ReadConcern("snapshot"),
            write_concern=WriteConcern(w="majority"),
            read_preference=ReadPreference.PRIMARY):
        orders.insert_one({"sku": "abc123", "qty": 100}, session=my_session)
        inventory.update_one({"sku": "abc123", "qty": {"$gte": 100}},
                             {"$inc": {"qty": -100}}, session=my_session)
        commit_with_retry(my_session)

# Шаг 2: Пытаемся запустить и зафиксировать транзакцию с помощью кода
# повторного выполнения операции;
def commit_with_retry(session):
    while True:
        try:
            # Фиксация использует запись, установленную при запуске транзакции;
            session.commit_transaction()
            print("Transaction committed.")
            break
        except (ConnectionFailure, OperationFailure) as exc:
            # Можем попытаться повторно выполнить фиксацию;
            if exc.has_error_label("UnknownTransactionCommitResult"):
                print("UnknownTransactionCommitResult, retrying "
                      "commit operation ...")
                continue
            else:
                print("Error during commit ...")
                raise

# Шаг 3: Пытаемся выполнить функцию транзакции txn_func с помощью кода
# повторного выполнения операции;
def run_transaction_with_retry(txn_func, session):
    while True:
        try:
            txn_func(session) # performs transaction
            break
        except (ConnectionFailure, OperationFailure) as exc:
            # При наличии временной ошибки повторяем всю транзакцию;
            if exc.has_error_label("TransientTransactionError"):
                print("TransientTransactionError, retrying transaction ...")
                continue
            else:
```

```
                raise

# Шаг 4: Запуск сеанса с помощью client.start_session() в качестве
# сеанса my_session:

# Шаг 5: Вызываем функцию 'run_transaction_with_retry', передав ей функцию
# для вызова 'update_orders_and_inventory' и сеанс 'my_session', чтобы
# связать его с этой транзакцией;
    try:
        run_transaction_with_retry(update_orders_and_inventory, my_session)
    except Exception as exc:

    # Делаем что-то с ошибкой. Код обработки ошибок не реализуется
    # с помощью базового API.
    raise
```

Теперь давайте посмотрим, как API обратного вызова можно использовать в Python для этого же примера транзакции. Две операции нашей транзакции выделены на шаге 1 ниже:

```
# Определяем uriString, используя формат соединения DNS Seedlist для соединения;
uriString = 'mongodb+srv://server.example.com/'
client = MongoClient(uriString)
my_wc_majority = WriteConcern('majority', wtimeout=1000)

# Предварительное условие / Шаг 0:Создаем коллекции, если их еще не существует;
# Операции CRUD в транзакциях должны быть для существующих коллекций;

client.get_database( "webshop",
                     write_concern=my_wc_majority).orders.insert_one({"sku":
                     "abc123", "qty":0})
client.get_database( "webshop",
                     write_concern=my_wc_majority).inventory.insert_one(
                     {"sku": "abc123", "qty": 1000})

# Шаг 1: Определяем обратный вызов, указывающий последовательность операций,
# которые должны быть выполнены внутри транзакции;
def callback(my_session):
    orders = my_session.client.webshop.orders
    inventory = my_session.client.webshop.inventory

    # Важно : вы должны передать переменную сеанса my_session операциям;

    orders.insert_one({"sku": "abc123", "qty": 100}, session=my_session)
    inventory.update_one({"sku": "abc123", "qty": {"$gte": 100}},
```

```
                        {"$inc": {"qty": -100}}, session=my_session)

# Шаг 2: запускаем сеанс клиента с помощью client.start_session() в качестве сеанса:

# Шаг 3: используем with_transaction для запуска транзакции, выполняем
# обратный вызов и фиксацию (или отмену при возникновении ошибки);

    session.with_transaction(callback,
                             read_concern=ReadConcern('local'),
                             write_concern=my_write_concern_majority,
                             read_preference=ReadPreference.PRIMARY)
}
```

В многодокументных транзакциях MongoDB можно выполнять операции чтения/записи (CRUD) только в существующих коллекциях или базах данных. Как показано в нашем примере, сначала вы должны создать коллекцию вне транзакции, если хотите вставить ее в транзакцию. Операции создания, удаления или индексации в транзакции запрещены.

Настройка ограничений транзакций для вашего приложения

Есть несколько параметров, которые важно знать при использовании транзакций. Их можно настроить, чтобы обеспечить оптимальное использование транзакций вашим приложением.

Ограничения на размер журнала операций и ограничения по времени

В транзакциях MongoDB есть две основные категории ограничений. Первая относится к временным ограничениям транзакции, контролю продолжительности конкретной транзакции, времени ожидания транзакции для получения блокировок и максимальной продолжительности выполнения всех транзакций. Вторая категория, в частности, относится к записи в журнале операций MongoDB и ограничениям размера отдельной записи.

Ограничения по времени

Максимальное время выполнения транзакции по умолчанию составляет одну минуту или меньше. Его можно увели-

чить путем изменения ограничения, управляемого параметром `transactionLifetimeLimitSeconds` на уровне экземпляра `mongod`. В случае разделенных кластеров параметр должен быть установлен для всех членов набора реплик. По истечении этого времени транзакция будет считаться истекшей и будет прервана процессом очистки, который выполняется периодически. Процесс очистки будет запускаться один раз каждые 60 секунд или каждые `transactionLifetimeLimitSeconds/2`, в зависимости от того, что меньше.

Чтобы явно установить ограничение по времени для транзакции, рекомендуется указать `maxTimeMS` для команды `commitTransaction`. Если `maxTimeMS` не установлен, то будет использоваться параметр `transactionLifetimeLimitSeconds`, или если она установлена, но будет превышать `transactionLifetimeLimitSeconds`, то вместо нее будет использоваться `transactionLifetimeLimitSeconds`.

Максимальное время ожидания транзакции по умолчанию для получения блокировок, необходимых для выполнения операций в транзакции, составляет 5 мс. Его можно увеличить, изменив лимит, контролируемый параметром `maxTransactionLockRequestTimeoutMillis`. Если транзакция не может получить блокировки в течение этого времени, она будет прервана. `maxTransactionLockRequestTimeoutMillis` можно установить на 0, –1 или число больше 0. Установка его значения на 0 означает, что транзакция будет прервана, если она не сможет незамедлительно получить все требуемые блокировки. Установка на –1 будет использовать тайм-аут для конкретной операции, как указано в `maxTimeMS`. Любое число больше 0 настраивает время ожидания на это время в секундах в качестве указанного периода, в течение которого транзакция попытается получить требуемые блокировки.

Ограничения на размер журнала операций

MongoDB создаст столько записей в журнале операций, сколько необходимо для операций записи в транзакции. Однако каждая запись должна находиться в пределах ограничений размера документа в формате BSON 16 МБ.

Транзакции предоставляют полезную функцию в MongoDB для обеспечения согласованности, но их следует использовать с моделью расширенного документа. Гибкость этой модели и использование передовых практик, таких как шаблоны проектирования схем, помогут избежать использования транзакций в большинстве ситуаций. Транзакции – мощное свойство, которое не стоит часто использовать в своих приложениях.

Глава 9

Разработка приложений

Эта глава посвящена разработке приложений для эффективной работы с MongoDB. Здесь обсуждаются:

- аспекты проектирования схем;
- компромиссы при принятии решения относительно того, вставлять данные или ссылаться на них;
- советы по оптимизации;
- принципы согласованности;
- как переносить схемы;
- как управлять схемами;
- когда MongoDB – плохой выбор для хранения данных.

Аспекты проектирования схем

Ключевым аспектом представления данных является разработка схемы, которая представляет собой способ представления ваших данных в документах. Наиболее подходящий подход к этой разработке – представлять данные такими, какими их хочет видеть ваше приложение. Таким образом, в отличие от реляционных баз данных, здесь сначала нужно разобраться со своими запросами и освоить шаблоны доступа к данным, прежде чем моделировать схему.

Вот ключевые аспекты, которые необходимо учитывать при проектировании схемы:

Ограничения

Вы должны понимать любые ограничения базы данных или оборудования. Также необходимо учитывать ряд специфических аспектов MongoDB, таких как максимальный размер документа 16 МБ, то, что полные документы читаются и записываются с диска, что при обновлении документ перезаписывается полностью и что атомарные обновления находятся на уровне документа.

Шаблоны доступа к вашим запросам и вашим записям

Вам нужно будет определить и количественно оценить рабочую нагрузку своего приложения и более широкой системы. Рабочая нагрузка охватывает как чтение, так и запись в вашем приложении. Как только вы узнаете, когда выполняются запросы и как часто, вы сможете определить наиболее распространенные запросы. Это запросы, которые вам необходимо разработать для поддержки своей схемы. После того как вы определили эти запросы, вы должны попытаться свести к минимуму количество запросов и убедиться, что данные, которые запрашиваются вместе, хранятся в одном и том же документе.

Данные, не используемые в этих запросах, должны быть помещены в другую коллекцию. Данные, которые используются редко, также должны быть перемещены в другую коллекцию. Стоит подумать, можете ли вы разделить динамические данные (чтение/запись) и статические (в основном чтение). Наилучшие результаты производительности достигаются, когда вы расставляете приоритеты в своей схеме для самых распространенных запросов.

Типы связей

Вы должны учитывать, какие данные связаны друг с другом с точки зрения потребностей вашего приложения, а также взаимосвязи между документами. Затем вы можете определить наиболее подходящие подходы для встраивания данных или документов или ссылки на них. Вам нужно будет решить, как ссылаться на документы, не выполняя дополнительные запросы, а также сколько документов обновляется при изменении взаимосвязи. Необходимо также учитывать, легко ли запрашивать структуру данных, как, например, в случае со вложенными массивами (массивами в массивах), которые поддерживают моделирование определенных взаимосвязей.

Кардинальность

После того как вы определили, как ваши документы и ваши данные связаны друг с другом, следует рассмотреть кардинальность этих связей. В частности, это тип связи «один к одному», «один ко многим», «многие ко многим», «один к миллионам» или «многие к миллиардам». Очень важно установить кардинальность связей, чтобы убедиться, что вы используете наиболее подходящий формат для их моделирования в вашей схеме MongoDB. Вы также должны изучить, выполняется ли обращение к объекту на стороне «многие к миллионам» по отдельности или только в контексте родительского объекта, а также соотношение обновлений к чтениям для рассматриваемого поля данных. Ответы на эти вопросы помогут вам определить, сле-

дует ли встраивать документы или справочные материалы и нужно ли выполнять денормализацию данных в документах.

Шаблоны проектирования схем

Разработка схемы важна в MongoDB, поскольку она напрямую влияет на производительность приложения. При разработке схем встречается много распространенных проблем, которые можно решить с помощью известных шаблонов или «строительных блоков». При разработке схем рекомендуется использовать один или несколько таких шаблонов вместе.

Шаблоны проектирования схем, которые могут применяться, включают в себя:

Полиморфный шаблон

Данный шаблон подходит, когда все документы в коллекции имеют похожую, но не идентичную структуру. Он включает в себя определение общих полей в документах, которые поддерживают общие запросы, которые будут выполняться приложением. Отслеживание определенных полей в документах или вложенных документах поможет определить различия между данными и различными путями кода или классами/подклассами, которые могут быть закодированы в вашем приложении для управления этими различиями. Это позволяет использовать простые запросы в одной коллекции не совсем идентичных документов для повышения производительности запросов.

Атрибут

Подходит, когда в документе есть подмножество полей, которые имеют общие функции, по которым вы хотите сделать сортировку или выполнить запрос, или когда поля, по которым нужно сортировать, существуют только в подмножестве документов, или когда оба эти условия истинны. Он включает в себя преобразование данных в массив пар типа «ключ/значение» и создание индекса для элементов в этом массиве. В качестве дополнительных полей для этих пар можно добавить квалификаторы. Этот шаблон помогает нацелиться на множество похожих полей для каждого документа, поэтому индексов требуется меньше и писать запросы проще.

Ведро

Подходит для данных временных рядов, где данные собираются в виде потока в течение определенного периода времени. В MongoDB гораздо эффективнее объединить эти данные в набор документов, каждый из которых содержит данные за определенный промежуток

времени, чем создавать документ для каждого момента времени / точки данных. Например, можно использовать одночасовое ведро и поместить все показания за этот час в массив в одном документе. Сам документ будет иметь время начала и окончания, указывающее период, который охватывает «ведро».

Выброс

Этот шаблон относится к тем редким случаям, когда несколько запросов документов выходят за рамки обычного шаблона для приложения. Это расширенный шаблон, разработанный для ситуаций, когда популярность является фактором. Его можно встретить в социальных сетях с главными лидерами мнения, продажах книг, обзорах фильмов и т. д. Он использует флаг, чтобы указать, что документ является выбросом, и сохраняет дополнительное переполнение в одном или нескольких документах, которые ссылаются на первый документ через `"_id"`. Флаг будет использоваться кодом вашего приложения для выполнения дополнительных запросов, чтобы получить документ (документы) переполнения.

Вычислительный шаблон

Используется, когда данные должны вычисляться часто, а также может использоваться, когда шаблон доступа к данным обладает высокой интенсивностью чтения данных. Этот шаблон рекомендует, чтобы расчеты производились в фоновом режиме при периодическом обновлении основного документа. Он обеспечивает правильное приближение вычисляемых полей или документов без необходимости их непрерывной генерации для отдельных запросов. Он может значительно снизить нагрузку на ЦП, чтобы избежать повторения одних и тех же вычислений, особенно в тех случаях, когда чтение инициирует вычисление и у вас высокое соотношение чтение/запись.

Подмножество

Используется, когда у вас есть рабочий набор, превышающий доступную оперативную память машины, что может быть вызвано большими документами, содержащими большое количество информации, которая не используется вашим приложением. Этот шаблон предполагает, что вы разделяете часто используемые данные и редко используемые данные на две отдельные коллекции. Типичным примером может служить приложение электронной коммерции, которое хранит десять самых последних обзоров продукта в «основной» (часто используемой) коллекции и перемещает все старые обзоры во вторую коллекцию, запрашиваемую только в том случае, если приложению требуется больше, чем десять последних отзывов.

Расширенная ссылка

Используется в случаях, когда у вас есть множество разных логических сущностей или «вещей», каждая из которых имеет собственную коллекцию, но вы хотите собрать эти сущности воедино для определенной функции. Типичная схема электронной торговли может иметь отдельные коллекции для заказов, клиентов и товаров. Это может оказать негативное влияние на производительность, если мы хотим собрать всю информацию по одному заказу из этих отдельных коллекций. Решение состоит в том, чтобы идентифицировать часто используемые поля и дублировать их в документе заказа. В случае заказа для онлайн-магазина это будет имя и адрес клиента, которому мы отправляем товар. Этот шаблон устраняет дублирование данных для уменьшения количества запросов, необходимых для сопоставления информации.

Аппроксимация

Полезен в ситуациях, когда требуются ресурсозатратные вычисления (время, память, циклы ЦП), но безошибочная точность не требуется. В качестве примера можно привести счетчик картинок или эмодзи, или счетчик просмотров страниц, где не обязательно знать точное количество (например, 999 535 или 1 000 0000). В таких ситуациях применение этого шаблона может значительно сократить количество операций записи, например путем обновления счетчика только после каждых 100 или более просмотров, а не после каждого.

Дерево

Может применяться, когда у вас много запросов и у вас есть данные, которые в основном имеют иерархическую структуру. Этот шаблон происходит из более ранней концепции хранения данных вместе, которая обычно запрашивается вместе. В MongoDB легко можно хранить иерархию в массиве в пределах одного и того же документа. В примере с сайтом онлайн-магазина, в частности с каталогом товаров, часто встречаются товары, относящиеся к нескольким категориям или категориям, которые являются частью других категорий. В качестве примера можно привести «Жесткий диск», который сам по себе является категорией, но относится к категории «Хранилище», которая сама относится к категории «Комплектующие», являющейся частью категории «Электроника». В этом сценарии у нас будет поле, которое будет отслеживать всю иерархию, и еще одно поле, где будет содержаться непосредственная категория («Жесткий диск»). Все поле иерархии, хранящееся в массиве, предоставляет возможность использовать многоключевой индекс для этих значений. Это гарантирует, что все элементы, связанные с категориями

в иерархии, легко будет найти. Поле непосредственной категории позволяет найти все элементы, напрямую связанные с этой категорией.

Предварительное определение

Первоначально использовался с подсистемой хранения MMAP, тем не менее он еще встречается. Данный шаблон рекомендует создать начальную пустую структуру, которая будет заполнена позже. Примером использования может быть система бронирования, которая ежедневно управляет ресурсом, отслеживая, является ли номер свободным или он уже забронирован/недоступен. Двумерная структура ресурсов (*x*) и дней (*y*) позволяет легко проверить доступность номеров и выполнить расчеты.

Шаблон управления версиями документов

Обеспечивает механизм, позволяющий сохранять старые версии документов. Он требует добавления дополнительного поля к каждому документу для отслеживания версии документа в «основной» коллекции и дополнительной коллекции, которая содержит все редакции документов. Этот шаблон делает несколько предположений: в частности, что каждый документ имеет ограниченное количество версий, что это не большое количество документов, которые должны быть версионированы, и что запросы в основном выполняются для текущей версии каждого документа. В ситуациях, когда эти предположения недействительны, вам может потребоваться изменить шаблон или рассмотреть другой шаблон проектирования.

MongoDB предоставляет несколько полезных онлайн-ресурсов по шаблонам и проектированию схем. Университет MongoDB предлагает бесплатный курс M320 Data Modeling (https://oreil.ly/BYtSr), а также серию блогов «Сборка по шаблонам» (https://oreil.ly/MjSld).

Нормализация и денормализация

Существует много способов представления данных, и одним из наиболее важных вопросов, который следует учитывать, является степень нормализации данных. *Нормализация* обозначает разделение данных на несколько коллекций со ссылками между коллекциями. Каждый фрагмент данных находится в одной коллекции, хотя несколько документов могут ссылаться на него. Таким образом, чтобы изменить данные, нужно обновить только один документ. Фреймворк агрегации MongoDB предлагает соединения с помощью этапа `$lookup`, который выполняет левое внешнее соединение, добавляя документы в «объединенную» коллекцию, в кото-

рой есть совпадающий документ в исходной коллекции, – добавляет новое поле массива к каждому сопоставленному документу в «объединенной» коллекции с подробностями о документе из исходной коллекции. Эти измененные документы затем доступны на следующем этапе для дальнейшей обработки.

Денормализация – процесс, обратный нормализации: встраивание всех данных в один документ. Вместо документов, содержащих ссылки на одну точную копию данных, множество документов могут иметь копии данных. Это означает, что в случае изменения информации необходимо обновить несколько документов, но разрешить, чтобы все связанные данные выбирались одним запросом.

Решение о том, когда прибегать к нормализации, а когда к денормализации, может быть непростым: обычно нормализация ускоряет запись, а денормализация ускоряет чтение. Таким образом, вам нужно решить, какие *компромиссы* имеют смысл для вашего приложения.

Примеры представления данных

Предположим, мы сохраняем информацию о студентах и занятиях, которые они посещают. Одним из способов представить ее будет наличие коллекций *students* (каждый студент – это один документ) и *classes* (каждое занятие – один документ). Далее у нас может быть третья коллекция (*studentClasses*), где содержатся ссылки на студентов и занятия, которые они посещают:

```
> db.studentClasses.findOne({"studentId" : id})
{
    "_id" : ObjectId("512512c1d86041c7dca81915"),
    "studentId" : ObjectId("512512a5d86041c7dca81914"),
    "classes" : [
        ObjectId("512512ced86041c7dca81916"),
        ObjectId("512512dcd86041c7dca81917"),
        ObjectId("512512e6d86041c7dca81918"),
        ObjectId("512512f0d86041c7dca81919")
    ]
}
```

Если вы знакомы с реляционными базами данных, то, возможно, встречали такой тип связующей таблицы раньше (хотя обычно у вас есть один студент и одно занятие на документ вместо списка идентификаторов занятий). Это немного больше в стиле MongoDB помещать занятия в массив, но обычно вы вряд ли захотите хранить данные таким образом, потому что требуется большое количество запросов, чтобы получить актуальную информацию.

Предположим, мы хотели найти занятия, которые посещал студент. Мы выполнили запрос на студента из коллекции *students*, выполнили запрос к коллекции *studentClasses*, чтобы получить `"_id"` курса, а затем обратились к коллекции *classes* для получения информации о занятии. Таким образом, для поиска этой информации потребуется три путешествия к серверу. Как правило, это не тот способ, который вы хотите применять, чтобы структурировать данные в MongoDB, за исключением случаев, когда занятия и ученики постоянно меняются и чтение данных не требует быстрого выполнения.

Мы можем удалить один из запросов разыменования, вставив ссылки на занятия в документ студента:

```
{
    "_id" : ObjectId("512512a5d86041c7dca81914"),
    "name" : "John Doe",
    "classes" : [
        ObjectId("512512ced86041c7dca81916"),
        ObjectId("512512dcd86041c7dca81917"),
        ObjectId("512512e6d86041c7dca81918"),
        ObjectId("512512f0d86041c7dca81919")
    ]
}
```

Поле `"classes"` содержит массив `"_id"` занятий, которые посещает Джон Доу. Когда мы хотим узнать информацию об этих занятиях, мы можем запросить коллекцию *classes* с этими `"_id"`. Нужно всего два запроса. Это довольно популярный способ структурирования данных, который не обязательно должен быть мгновенно доступным и изменяется, но не постоянно.

Если нам нужно дальше оптимизировать операции чтения, мы можем получить всю информацию в одном запросе, полностью денормализовывая данные и сохраняя каждое занятие в виде вложенного документа в поле `"classes"`:

```
{
    "_id" : ObjectId("512512a5d86041c7dca81914"),
    "name" : "John Doe",
    "classes" : [
        {
            "class" : "Trigonometry",
            "credits" : 3,
            "room" : "204"
        },
        {
            "class" : "Physics",
            "credits" : 3,
```

```
            "room" : "159"
        },
        {
            "class" : "Women in Literature",
            "credits" : 3,
            "room" : "14b"
        },
        {
            "class" : "AP European History",
            "credits" : 4,
            "room" : "321"
        }
    ]
}
```

Плюс тут состоит в том, что для получения информации требуется всего один запрос. Недостатком является то, что он занимает больше места и его сложнее синхронизировать. Например, если окажется, что физика должна была стоить четыре кредита (а не три), каждому учащемуся на уроке физики необходимо обновить документ (вместо того чтобы просто обновить центральный документ «Физика»).

Наконец, вы можете использовать шаблон расширенной ссылки, упомянутый ранее, который представляет собой гибрид встраивания и ссылок – вы создаете массив вложенных документов с часто используемой информацией, но со ссылкой на фактический документ для получения дополнительной информации:

```
{
    "_id" : ObjectId("512512a5d86041c7dca81914"),
    "name" : "John Doe",
    "classes" : [
        {
            "_id" : ObjectId("512512ced86041c7dca81916"),
            "class" : "Trigonometry"
        },
        {
            "_id" : ObjectId("512512dcd86041c7dca81917"),
            "class" : "Physics"
        },
        {
            "_id" : ObjectId("512512e6d86041c7dca81918"),
            "class" : "Women in Literature"
        },
        {
            "_id" : ObjectId("512512f0d86041c7dca81919"),
```

```
            "class" : "AP European History"
        }
    ]
}
```

Данный подход также является хорошим вариантом, потому что объем внедренной информации может со временем меняться по мере изменения ваших требований: если вы хотите включить больше или меньше информации на странице, можно в большей или меньшей степени включить ее в документ.

Еще важным аспектом является то, насколько часто эта информация будет меняться по сравнению с тем, насколько часто она читается. Если она будет обновляться регулярно, тогда нормализация – хорошая идея. Тем не менее если она меняется нечасто, то оптимизация процесса обновления дает мало преимуществ за счет каждого чтения, которое выполняет ваше приложение.

Например, сценарий использования нормализации учебника – хранить пользователя и его адрес в отдельных коллекциях. Однако адреса людей редко меняются, поэтому обычно не следует «браковать» каждую операцию чтения, если вдруг кто-то переехал. Ваше приложение должно встроить адрес в пользовательский документ.

Если вы решили использовать вложенные документы и вам необходимо обновить их, вам следует настроить задачу планировщика, чтобы гарантировать, что все ваши обновления успешно распространяются на каждый документ. Например, предположим, что вы пытаетесь выполнить многократное обновление, но сервер вылетает до того, как все документы были обновлены. Вам нужен способ обнаружить это и повторить обновление.

С точки зрения операторов обновления, `"$set"` является идемпотентным, а `"$inc"` – нет. Идемпотентные операции будут иметь один и тот же результат, сделали ли бы они одну попытку или несколько; в случае ошибки сети повторное выполнение операции будет достаточным для обновления. В случае неидемпотентных операторов операция должна быть разбита на две операции, которые идемпотентны по отдельности и безопасны для повторения. Этого можно достичь путем включения уникального ожидающего токена в первую операцию, а вторая операция будет использовать и уникальный ключ, и уникальный ожидающий токен. Этот подход позволяет оператору `"$inc"` быть идемпотентным, потому что каждая отдельная операция `updateOne` идемпотентна.

В какой-то степени чем больше информации вы генерируете, тем меньше ее следует встраивать. Если предполагается, что содержимое вложенных полей или их число будет расти без ограничений, то на них обычно следует ссылаться, а не внедрять. Такие вещи, как деревья комментариев или списки активности, должны храниться как их собственные докумен-

ты, а не как вложенные. Также стоит рассмотреть возможность использования шаблона *Подмножество* (описанного в разделе «Шаблоны проектирования схем»), чтобы сохранять самые последние элементы (или другие подмножества) в документе.

Наконец, включенные поля должны быть неотъемлемой частью данных в документе. Если поле почти всегда исключается из результатов при запросе документа, это хороший признак того, что он может относиться к другой коллекции. Эти рекомендации приведены в табл. 9.1.

Таблица 9.1. Сравнение встраивания и ссылок

Встраивание лучше подходит для ...	Ссылки лучше подходят для ...
Небольших вложенных документов	Больших вложенных документов
Данных, которые не меняются регулярно	Изменчивых данных
Когда приемлема согласованность в конечном счете	Когда необходима мгновенная последовательность
Документов, которые растут на небольшое количество	Документов, которые растут на небольшое количество
Данных, для которых вам часто нужно будет выполнить второй запрос, чтобы получить их	Данных, которые вы часто будете исключать из результатов
Быстрых операций чтения	Быстрых операций записи

Предположим, у нас есть коллекция *users*. Вот несколько примеров полей, которые могут быть у нас в пользовательских документах, и указание того, должны ли они быть встроенными:

Настройки аккаунта

Они относятся только к данному пользовательскому документу и, вероятно, будут представлены другой пользовательской информацией в документе. Настройки аккаунта обычно должны быть встроены.

Недавняя активность

Это зависит от того, насколько в последнее время активность растет и изменяется. Если это поле фиксированного размера (скажем, последние 10 вещей), может быть полезно внедрить эту информацию или реализовать шаблон *Подмножество*.

Друзья

Обычно эта информация не должна быть включена или, по крайней мере, должна быть включена не полностью. *См.* раздел «Друзья, подписчики и другие неудобства».

Весь контент, созданный этим пользователем

Это встраивать не нужно.

Кардинальность

Кардинальность указывает на то, сколько ссылок есть у коллекции на другую коллекцию. Распространенные типы связей – «один к одному», «один ко многим» или «многие ко многим». Например, предположим, у нас было приложение для блога. У каждого *поста* есть *заголовок*, поэтому это связь «один к одному». У каждого *автора* много *постов*, поэтому это связь «один ко многим». И у *постов* множество *тегов*, а *теги* относятся ко множеству *постов*, поэтому это связь «многие ко многим».

При использовании MongoDB с концептуальной точки зрения может быть полезно разделить «многие» на подкатегории: «многие» и «несколько». Например, между авторами и постами может быть связь «один к нескольким»: каждый автор пишет только несколько постов. Между постами в блоге и тегами может быть связь «многие к нескольким»: вероятно, постов в блоге у вас намного больше, чем тегов. Однако между сообщениями в блоге и комментариями может быть связь «один ко многим»: у каждого поста множество комментариев.

Определение связей «несколько против многих» может помочь вам решить, что вставлять, а на что ссылаться. Как правило, связи типа «несколько» будут лучше работать со встраиванием, а связи типа «многие» будут лучше работать как ссылки.

Друзья, подписчики и другие неудобства

Держите своих друзей поблизости, а своих врагов – в них.

В этом разделе рассматриваются вопросы, связанные с данными социальных графов. Многие социальные приложения должны связывать людей, контент, подписчиков, друзей и т. д. Выяснить, как сбалансировать встраивание и ссылки на эту тесно связанную информацию, может быть непросто, но в целом добавление в друзья или добавление в избранное можно упростить до системы публикации/подписки: один пользователь подписывается на уведомления от другого. Таким образом, есть две основные операции, которые должны быть эффективными: хранение подписчиков и уведомление всех заинтересованных сторон о событии.

Есть три способа, с помощью которых обычно реализуется подписка. Первый вариант – поместить производителя в документ подписчика, который выглядит примерно так:

```
{
    "_id" : ObjectId("51250a5cd86041c7dca8190f"),
```

```
    "username" : "batman",
    "email" : "batman@waynetech.com"
    "following" : [
        ObjectId("51250a72d86041c7dca81910"),
        ObjectId("51250a7ed86041c7dca81936")
    ]
}
```

Теперь, учитывая документ пользователя, можно выполнить запрос, подобный следующему, чтобы найти все опубликованные действия, которые могут его заинтересовать:

```
db.activities.find({"user" : {"$in" :
        user["following"]}})
```

Однако если вам нужно найти всех, кто интересуется вновь опубликованным действием, вам нужно будет выполнить запрос для поля `"following"` для всех пользователей.

В качестве альтернативы можно добавить подписчиков в документ производителя, например так:

```
{
    "_id" : ObjectId("51250a7ed86041c7dca81936"),
    "username" : "joker",
    "email" : "joker@mailinator.com"
    "followers" : [
        ObjectId("512510e8d86041c7dca81912"),
        ObjectId("51250a5cd86041c7dca8190f"),
        ObjectId("512510ffd86041c7dca81910")
    ]
}
```

Всякий раз, когда этот пользователь что-то делает, все пользователи, которых вы должны уведомить, находятся тут же. Недостатком является то, что теперь вам нужно запросить всю коллекцию *users*, чтобы найти всех, на кого подписан пользователь (противоположное ограничение, как в предыдущем случае).

Любой из этих вариантов имеет дополнительный недостаток: они делают ваши пользовательские документы большими и более изменчивыми. Поле `"following"` (или `"followers"`) нередко даже не нужно возвращать: как часто вы хотите перечислять каждого подписчика? Таким образом, последний вариант нейтрализует эти недостатки, нормализуя данные еще больше и сохраняя подписки в другой коллекции. Подобная нормализация бывает излишней, но это может быть полезным для чрезвычайно изменчивого поля, которое нередко не возвращается вместе с остальной частью

документа. `"followers"` может быть правильным полем для выполнения нормализации таким способом.

В этом случае вы сохраняете коллекцию, которая сопоставляет издателей и подписчиков, с документами, которые выглядят примерно так:

```
{
    "_id" : ObjectId("51250a7ed86041c7dca81936"), // followee's "_id"
    "followers" : [
        ObjectId("512510e8d86041c7dca81912"),
        ObjectId("51250a5cd86041c7dca8190f"),
        ObjectId("512510ffd86041c7dca81910")
    ]
}
```

Это сохраняет ваши пользовательские документы небольшими, но означает, что для получения подписчиков необходим дополнительный запрос.

Работа с эффектом Уила Уитона

Независимо от того, какую стратегию вы используете, встраивание работает только с ограниченным количеством вложенных документов или ссылок. Если у вас есть знаменитости, они могут переполнить любой документ, в котором хранятся подписчики. Типичный способ компенсировать это – использовать шаблон *Выброс*, описанный в разделе «Шаблоны проектирования схем», и при необходимости иметь «продолжение» документа. Например, у вас может быть такой код:

```
> db.users.find({"username" : "wil"})
{
    "_id" : ObjectId("51252871d86041c7dca8191a"),
    "username" : "wil",
    "email" : "wil@example.com",
    "tbc" : [
        ObjectId("512528ced86041c7dca8191e"),
        ObjectId("5126510dd86041c7dca81924")
    ]
    "followers" : [
        ObjectId("512528a0d86041c7dca8191b"),
        ObjectId("512528a2d86041c7dca8191c"),
        ObjectId("512528a3d86041c7dca8191d"),
        ...
    ]
}
{
    "_id" : ObjectId("512528ced86041c7dca8191e"),
    "followers" : [
```

```
        ObjectId("512528f1d86041c7dca8191f"),
        ObjectId("512528f6d86041c7dca81920"),
        ObjectId("512528f8d86041c7dca81921"),
        ...
]
}
{
    "_id" : ObjectId("5126510dd86041c7dca81924"),
    "followers" : [
        ObjectId("512673e1d86041c7dca81925"),
        ObjectId("512650efd86041c7dca81922"),
        ObjectId("512650fdd86041c7dca81923"),
        ...
    ]
}
```

Затем добавьте логику приложения для поддержки выбора документов в массиве `"tbc"` («to be continued»).

Оптимизация манипулирования данными

Чтобы оптимизировать свое приложение, сначала нужно определить его узкое место, оценив производительность чтения и записи. Оптимизация операций чтения обычно предполагает наличие правильных индексов и возврат максимально возможного количества информации в одном документе. Оптимизация операций записи обычно включает в себя минимизацию количества индексов, которым вы располагаете, и делает обновления максимально эффективными.

Часто существует компромисс между схемами, которые оптимизированы для быстрой записи, и схемами, которые оптимизированы для быстрого чтения, поэтому вам, возможно, придется решить, что важнее для вашего приложения. Принимайте во внимание не только важность чтения по сравнению с записью, но и их пропорции: если записи важнее, но вы выполняете тысячу операций чтения для каждой записи, возможно, у вас все равно появится желание сначала оптимизировать операции чтения.

Удаление старых данных

Некоторые данные важны только на короткое время: через несколько недель или месяцев они просто занимают место в хранилище. Существует три популярных варианта удаления старых данных: использование ограниченных коллекций, применение коллекций TTL и удаление коллекций за каждый период времени.

Самый простой вариант – использовать ограниченную коллекцию: установите ее на большой размер и дайте старым данным «упасть» в ко-

нец. Однако ограниченные коллекции накладывают определенные ограничения на операции, которые можно выполнять, и подвержены резким скачкам трафика, что временно сокращает время их удержания. *См.* раздел «Ограниченные коллекции» для получения дополнительной информации.

Второй вариант – использовать коллекции TTL. Это дает вам более полный контроль над тем, когда документы удаляются, но может быть недостаточно быстрым для коллекций с очень большим объемом записи: документы удаляются путем обхода индекса TTL так же, как и при удалении по запросу пользователя. Однако если коллекция TTL продолжает работать, это, вероятно, самое простое решение для реализации. *См.* раздел «Индексы TTL» для получения дополнительной информации об индексах TTL.

Последний вариант – использовать несколько коллекций: например, по одной в месяц. Каждый раз, когда наступает новый месяц, ваше приложение начинает использовать (пустую) коллекцию этого месяца и выполняет поиск данных в коллекциях текущего и предыдущего месяцев. Когда коллекция старше, скажем, шести месяцев, ее можно удалить. Эта стратегия может выдерживать практически любой объем трафика, но приложение сложнее построить, поскольку вам придется использовать имена динамических коллекций (или баз данных) и, возможно, запрашивать несколько баз данных.

Планирование баз данных и коллекций

После того как вы наметили, как выглядят ваши документы, вы должны решить, в какие коллекции или базы данных их поместить. Часто это довольно интуитивно понятный процесс, но есть некоторые рекомендации, о которых следует помнить.

Как правило, документы с похожей схемой должны храниться в одной коллекции. MongoDB, как правило, не позволяет объединять данные из нескольких коллекций, поэтому, если есть документы, которые необходимо запросить или сгруппировать, это хороший вариант для помещения в одну большую коллекцию. Например, у вас могут быть документы, которые имеют довольно разные «формы», но если вы собираетесь их сгруппировать, все они должны находиться в одной коллекции (или можно использовать этап `$merge`, если они находятся в отдельных коллекциях или базах данных).

Что касается коллекций, то крупные проблемы, которые тут следует учитывать, – это блокировка (вы получаете блокировку чтения/записи для каждого документа) и хранение. Как правило, если у вас высокая рабочая нагрузка при записи, вам может потребоваться использовать несколько физических томов для уменьшения узких мест ввода/вывода. Каждая база данных может находиться в своем собственном каталоге, когда вы исполь-

зуете опцию `--directoryperdb`, давая себе возможность монтировать разные базы данных на разные тома. Таким образом, вы, возможно, захотите, чтобы все элементы в базе данных были одинакового «качества» с одинаковым шаблоном доступа или одинаковыми уровнями трафика.

Например, предположим, что у вас есть приложение с несколькими компонентами: компонентом ведения журнала, который создает огромное количество не очень ценных данных, коллекция пользователей и несколько коллекций для данных, сгенерированных пользователями. Эти коллекции имеют высокую ценность: важно, чтобы пользовательские данные были в безопасности. Существует также коллекция с большим объемом трафика для активности в соцсетях, которая имеет меньшее значение, но более важна, чем логи. Эта коллекция в основном используется для уведомлений пользователей, так что это почти коллекция только для добавления.

Разделив их по важности, вы можете получить три базы данных: *log*, *activities* и *users*. Приятным моментом здесь является то, что вы можете обнаружить, что ваши наиболее важные данные – это данные, которых у вас меньше всего (например, пользователи, вероятно, не генерируют столько данных, в отличие от журналирования). Вы не можете позволить себе твердотельный накопитель для всего набора данных, но можете раздобыть его для своих пользователей или использовать уровень RAID10 для пользователей и уровень RAID0 для журналов и операций.

Помните, что существуют некоторые ограничения при использовании нескольких баз данных до MongoDB версии 4.2 и появлении оператора `$merge` во фреймворке агрегации, который позволяет сохранять результаты агрегации из одной базы данных в другую базу данных и другую коллекцию в этой базе данных. Еще один момент, на который следует обратить внимание: команда `renameCollection` работает медленнее при копировании существующей коллекции из одной базы данных в другую, поскольку она должна копировать все документы в новую базу данных.

Управление согласованностью

Вы должны выяснить, насколько согласованными должны быть операции чтения вашего приложения. MongoDB поддерживает огромное количество уровней согласованности: от возможности всегда читать собственные записи до чтения данных неизвестной давности. Если вы отчитываетесь о деятельности за последний год, вам могут понадобиться только данные, которые соответствуют последним двум дням. И наоборот, если вы торгуете в режиме реального времени, вам может потребоваться немедленно прочитать последние записи.

Чтобы понять, как достичь этих различных уровней согласованности, важно понять, что MongoDB делает под капотом. Сервер хранит очередь запросов для каждого соединения. Когда клиент отправляет запрос, он бу-

дет помещен в конец очереди его соединения. Любые последующие запросы на соединение будут происходить после того, как ранее поставленная в очередь операция будет обработана. Таким образом, одно соединение имеет согласованное представление базы данных и всегда может читать свои собственные записи.

Обратите внимание, что это очередь для каждого соединения: если мы откроем две оболочки, у нас будет два соединения с базой данных. Если мы выполняем вставку в одну оболочку, последующий запрос в другую оболочку может не вернуть вставленный документ. Тем не менее в пределах одной оболочки, если мы запросим документ после его вставки, документ будет возвращен. Такое поведение может быть трудно продублировать вручную, но на занятом сервере могут возникать чередующиеся вставки и запросы. Разработчики часто сталкиваются с этим, когда вставляют данные в один поток, а затем проверяют, что они были успешно вставлены в другой. На мгновение или два создается впечатление, что данные не были вставлены, а затем они внезапно появляются.

Такое поведение особенно стоит учитывать при использовании драйверов Ruby, Python и Java, потому что все они используют объединение соединений в пул. Для эффективности эти драйверы открывают несколько соединений (*пул*) с сервером и распределяют запросы между ними. Однако все они имеют механизмы, гарантирующие, что серия запросов обрабатывается одним соединением. Подробная документация по объединению соединений в пул для различных языков программирования содержится на странице https://github.com/mongodb/specifications/blob/master/source/connection-monitoring-and-pooling/connection-monitoring-and-pooling.rst.

Когда вы отправляете операции чтения вторичному члену набору реплик (*см.* главу 12), это становится еще более серьезной проблемой. Вторичные члены могут отставать от первичных, что приводит к чтению данных за секунды, минуты или даже часы назад. Есть несколько способов справиться с этим, проще всего просто отправить все чтения первичному, если вас волнует устаревание данных.

MongoDB предлагает параметр `readConcern` для управления свойствами согласованности и изолированности читаемых данных. Ее можно сочетать с опцией `writeConcern` для контроля согласованности и гарантий доступности, предоставляемых вашему приложению. Существует пять уровней: `"local"`, `"available"`, `"majority"`, `"linearizable"` и `"snapshot"`. В зависимости от приложения, в тех случаях, когда вы хотите избежать устаревания данных по мере чтения, можно использовать уровень `"majority"`, который возвращает только надежные данные, которые были подтверждены большинством членов набора реплик и не будут откатываться.

`"linearizable"` также можно рассмотреть как вариант: он возвращает данные, которые отражают все успешные записи, подтвержденные большинством, которые были завершены до начала операции чтения.

MongoDB может дождаться завершения одновременного выполнения операций записи, прежде чем возвращать результаты с уровнем `"linearizable"`.

Три старших инженера-разработчика из MongoDB опубликовали документ под названием «Настраиваемая согласованность в MongoDB» (https://oreil.ly/PfcBx) на конференции PVLDB в 2019 году[1]. В нем описываются различные модели согласованности MongoDB, используемые для репликации, и то, как разработчики приложений могут использовать их.

Перенос схем

По мере роста вашего приложения и изменения ваших потребностей вашу схему, вероятно, также придется расширять и менять. Есть несколько способов сделать это, но независимо от выбранного вами метода вы должны тщательно задокументировать каждую схему, которую использовало ваше приложение. В идеале следует рассмотреть возможность применения шаблона *управления версиями документа* (*см.* раздел «Шаблоны проектирования схем»).

Самый простой способ – просто развивать вашу схему в соответствии с требованиями вашего приложения, убедившись, что ваше приложение поддерживает все старые версии схемы (например, принимая существование или отсутствие полей либо корректно обрабатывая несколько возможных типов полей). Но этот метод может вызвать путаницу, особенно если у вас есть конфликтующие версии схемы. Например, для одной версии может потребоваться поле `"mobile"`, для другой версии может потребоваться, чтобы этого поля не было, а вместо этого нужно другое поле, а в третьей версии поле `"mobile"` может быть необязательным. Отслеживание этих меняющихся требований может постепенно превратить ваш код в спагетти.

Чтобы обрабатывать изменяющиеся требования несколько более структурированным образом, вы можете включить поле `"version"` (или просто `"v"`) в каждый документ и использовать его, дабы определить, что ваше приложение примет для структуры документа. Это делает вашу схему более строгой: документ должен быть действительным для некой версии схемы, если это не текущая версия. Тем не менее поддержка старых версий по-прежнему необходима.

Последний вариант – перенести все ваши данные при изменении схемы. Как правило, это не очень хорошая идея: MongoDB позволяет вам иметь динамическую схему, чтобы избежать миграций, потому что они создают большую нагрузку на вашу систему. Однако если вы решите изменить каждый документ, вам необходимо убедиться, что все документы были успеш-

[1] Его авторы: Уильям Шульц, старший инженер-программист по репликации; Тесс Авитабиле, руководитель команды по репликации; и Элисон Кабрал, продакт-менеджер по распределенным системам.

но обновлены. MongoDB *поддерживает* транзакции, которые поддерживают данный тип миграции. Если MongoDB дает сбой посреди транзакции, старая схема будет сохранена.

Управление схемами

MongoDB ввела проверку схем в версии 3.2, которая позволяет выполнять проверку во время обновлений и вставок. В версии 3.6 добавлена проверка схемы JSON с помощью оператора `$jsonSchema`, что теперь является рекомендуемым методом для проверки всей схемы в MongoDB. На момент написания этого раздела MongoDB поддерживает черновой вариант 4 схемы JSON, но, пожалуйста, ознакомьтесь с документацией для получения самой последней информации об этой функции.

Процедура не проверяет существующие документы, пока они не будут изменены, и настраивается для каждой коллекции. Чтобы добавить проверку в существующую коллекцию, используйте команду `collMod` с параметром `validator`. Вы можете добавить проверку в новую коллекцию, указав параметр `validator` при использовании `db.createCollection()`. MongoDB также предоставляет две дополнительные опции: `validationLevel` и `validationAction`. `validationLevel` определяет, насколько строго применяются правила проверки к существующим документам во время обновления, а `validationAction` решает, должна ли произойти ошибка плюс отклонение или предупреждение в связи с возможностью использования недопустимых документов.

Когда не стоит использовать MongoDB

В то время как MongoDB представляет собой СУБД общего назначения, которая хорошо работает с большинством приложений, она не всем подходит. Есть несколько причин, по которым вам может понадобиться избегать использовать ее:

- объединение разных типов данных в разных измерениях – это то, что делает реляционные базы данных фантастическими. MongoDB не должна делать это хорошо и, скорее всего, никогда не будет;
- одна из главных (и надеюсь, временных) причин для использования реляционной базы данных поверх MongoDB заключается в том, что вы используете инструменты, которые ее не поддерживают. Существуют тысячи инструментов, которые просто не созданы для поддержки MongoDB, начиная от SQLAlchemy и заканчивая WordPress. Пул инструментов, которые поддерживают ее, тем не менее растет, но ее экосистема пока не имеет такого размера, какой есть у реляционных СУБД.

Часть III

Репликация

Глава 10

Настройка набора реплик

В этой главе описывается система высокой готовности MongoDB: наборы реплик. В ней рассказывается:

- что такое наборы реплик;
- как настроить набор реплик;
- какие параметры конфигурации доступны для членов набора реплик.

Знакомство с репликацией

Начиная с первой главы мы использовали автономный сервер, сервер *mongod*. Это простой способ для старта, но опасный способ для запуска в промышленную эксплуатацию. Что делать, если ваш сервер выходит из строя или становится недоступным? По крайней мере, в течение небольшого периода времени ваша база данных будет недоступна. Если у вас проблемы с оборудованием, возможно, вам придется перенести данные на другой компьютер. В худшем случае проблемы с диском или сетью могут привести к повреждению данных или их недоступности.

Репликация – это способ хранить идентичные копии ваших данных на нескольких серверах, рекомендуется для всех рабочих развертываний. Репликация поддерживает ваше приложение в рабочем состоянии, а ваши данные находятся в безопасности, даже если с одним или несколькими вашими серверами что-то случится.

В MongoDB репликация настраивается путем создания *набора реплик*. Набор реплик – это группа серверов с одним *первичным узлом*, который получает операции записи, и несколькими *вторичными узлами*, где хранятся копии данных первичного узла. Если первичный узел дает сбой, вторичные узлы могут выбрать новый первичный узел из их числа.

Если вы используете репликацию и сервер выходит из строя, вы по-прежнему можете получить доступ к своим данным с других серверов в наборе. Если данные на сервере повреждены или недоступны, вы можете сделать новую копию данных с одного из других членов набора.

В этой главе описываются наборы реплик и рассказывается, как настроить репликацию в своей системе. Если вас в меньшей степени интересует механика репликации и вы просто хотите создать набор реплик для тестирования/разработки или производственной эксплуатации, используйте облачное решение MongoDB, MongoDB Atlas (https://atlas.mongodb.com). Оно просто в применении и предоставляет бесплатный уровень для экспериментов. Кроме того, для управления кластерами MongoDB в собственной инфраструктуре можно использовать Ops Manager (https://oreil.ly/-X6yp).

Настройка набора реплик, часть 1

В этой главе мы покажем вам, как настроить набор реплик из трех узлов на одной машине, чтобы вы могли начать экспериментировать с механикой набора реплик. Это тип установки, для которого вы можете написать скрипты, чтобы настроить и запустить реплику, а затем поработать с ней, используя административные команды в оболочке *mongo*, или смоделировать сетевые разделы или сбои сервера, чтобы лучше понять, как MongoDB работает с высокой готовностью и аварийным восстановлением. В производственной среде вы всегда должны использовать набор реплик и выделять выделенный хост каждому члену, чтобы избежать конфликта ресурсов и изолировать себя от сбоя сервера. Для обеспечения дальнейшей устойчивости также следует использовать формат подключения DNS Seedlist (https://oreil.ly/cCORE), чтобы указать, как ваши приложения подключаются к вашему набору реплик. Преимущество использования DNS состоит в том, что серверы, на которых размещаются члены набора реплик, можно менять по очереди без необходимости повторной настройки клиентов (в частности, их строки подключения).

Учитывая разнообразие доступных вариантов виртуализации и облачных вычислений, почти так же легко создать тестовый набор реплик с каждым членом на выделенном хосте. Мы предоставили скрипт Vagrant, чтобы вы могли поэкспериментировать с этим вариантом[1].

Чтобы начать работу с нашим тестовым набором реплик, давайте сначала создадим отдельные каталоги данных для каждого узла. В Linux или macOS выполните в терминале приведенную ниже команду, чтобы создать три каталога:

```
$ mkdir -p ~/data/rs{1,2,3}
```

В результате этого будут созданы каталоги `~/data/rs1`, `~/data/rs2` и `~/data/rs3` (символ ~ идентифицирует ваш домашний каталог).

В Windows для создания этих каталогов выполните в командной строке (cmd) или PowerShell следующую команду:

```
> md c:\data\rs1 c:\data\rs2 c:\data\rs3
```

[1] См. https://github.com/mongodb-the-definitive-guide-3e/mongodb-the-definitive-guide-3e.

Затем в Linux или macOS выполните каждую из приведенных ниже команд в отдельном терминале:

```
$ mongod --replSet mdbDefGuide --dbpath ~/data/rs1 --port 27017 \
    --smallfiles --oplogSize 200
$ mongod --replSet mdbDefGuide --dbpath ~/data/rs2 --port 27018 \
    --smallfiles --oplogSize 200
$ mongod --replSet mdbDefGuide --dbpath ~/data/rs3 --port 27019 \
    --smallfiles --oplogSize 200
```

В Windows выполните каждую из следующих команд в собственном окне командной строки или PowerShell:

```
> mongod --replSet mdbDefGuide --dbpath c:\data\rs1 --port 27017 \
    --smallfiles --oplogSize 200
> mongod --replSet mdbDefGuide --dbpath c:\data\rs2 --port 27018 \
    --smallfiles --oplogSize 200
> mongod --replSet mdbDefGuide --dbpath c:\data\rs3 --port 27019 \
    --smallfiles --oplogSize 200
```

После их запуска у вас должно быть запущено три отдельных процесса *mongod*.

В целом принципы, которые мы будем рассматривать в оставшейся части этой главы, применимы к наборам реплик, которые используются в производственных развертываниях, где у каждого экземпляра *mongod* есть выделенный хост. Однако есть дополнительные подробности, касающиеся защиты наборов реплик, которые мы рассмотрим в главе 19; здесь мы кратко остановимся на них в качестве предварительного просмотра.

Пара слов касательно работы в сети

Каждый член набора должен иметь возможность устанавливать соединения с другим членом набора (включая себя). Если вы получаете ошибки, сообщающие о том, что члены не могут связаться с другими членами, которые, как вы знаете, работают, возможно, вам придется изменить конфигурацию сети, чтобы разрешить возможность соединения между ними.

Запущенные вами процессы могут также легко выполняться на отдельных серверах.

Однако с выходом MongoDB версии 3.6 *mongod* связывается с *localhost* (127.0.0.1) только по умолчанию. Чтобы каждый член набора реплик мог

обмениваться данными с другими членами, вы также должны выполнить привязку к IP-адресу, доступному для других членов. Если бы мы запускали экземпляр *mongod* на сервере с сетевым интерфейсом с IP-адресом 198.51.100.1, и хотели бы запустить его в качестве члена набора реплик, и при этом каждый член будет находиться на разном сервере, можно было бы указать параметр командной строки `--bind_ip` или использовать `bind_ip` в файле конфигурации для этого экземпляра:

```
$ mongod --bind_ip localhost,192.51.100.1 --replSet mdbDefGuide \
    --dbpath ~/data/rs1 --port 27017 --smallfiles --oplogSize 200
```

В этом случае мы также сделаем аналогичные модификации для запуска других экземпляров *mongod*, независимо от того, работаем мы на Linux, macOS или Windows.

Вопросы безопасности

Перед привязкой к IP-адресам, отличным от *localhost*, при настройке набора реплик необходимо активировать элементы управления авторизацией и указать механизм аутентификации.

Кроме того, рекомендуется зашифровать данные на диске и обмен данными между членами набора реплик, а также между набором и клиентами. Более подробно о защите наборов реплик мы расскажем в главе 19.

Настройка набора реплик, часть 2

Вернемся к нашему примеру. После той работы, которую мы проделали до сих пор, процессы *mongod* пока еще не знают о существовании друг друга. Чтобы рассказать им друг о друге, нам нужно создать конфигурацию, в которой перечислены все члены, и отправить ее одному из наших процессов *mongod*. Он позаботится о передаче этой конфигурации другим членам.

В четвертом терминале, командной строке Windows или окне PowerShell запустите оболочку *mongo*, которая подключается к одному из работающих экземпляров *mongod*. Это можно сделать, набрав приведенную ниже команду. С ее помощью мы подключимся к *mongod*, работающему на порту 27017:

```
$ mongo --port 27017
```

Затем в оболочке *mongo* создайте документ конфигурации и передайте его вспомогательной функции `rs.initiate()`, чтобы инициировать набор реплик. Будет инициирован набор реплик, содержащий три члена, и конфигурация распространится на остальные процессы *mongod*, чтобы сформировать набор реплик:

```
> rsconf = {
    _id: "mdbDefGuide",
    members: [
        { _id: 0, host: "localhost:27017" },
        { _id: 1, host: "localhost:27018" },
        { _id: 2, host: "localhost:27019" }
        ]
  }
> rs.initiate(rsconf)
{ "ok": 1, "operationTime": Timestamp(1501186502, 1) }
```

Есть несколько важных частей документа конфигурации набора реплик. `"_id"` – это имя набора реплик, которое вы передали в командной строке (в данном примере это `"mdbDefGuide"`). Убедитесь, что это имя точно соответствует.

Следующая часть документа представляет собой массив членов набора. Каждому из них нужны два поля: `"_id"`, которое представляет собой целое число, которое уникально среди членов набора реплик, и имя хоста.

Обратите внимание, что мы используем *localhost* в качестве имени хоста для членов этого набора. Это только для примера. В последующих главах, где мы обсуждаем защиту наборов реплик, мы рассмотрим конфигурации, которые больше подходят для производственных развертываний. MongoDB разрешает тестировать наборы реплик на *localhost* локально, но будет протестовать, если вы попытаетесь смешивать локальный и нелокальный серверы в конфигурации.

Этот документ конфигурации является вашей конфигурацией набора реплик. Член, работающий на *localhost:27017*, проанализирует конфигурацию и отправит сообщения другим членам, предупреждая их о новой конфигурации. Как только все они загрузят ее, они выберут основной сервер и приступят к обработке операций чтения и записи.

К сожалению, нельзя преобразовать автономный сервер в набор реплик без простоя, для того чтобы перезапустить его и осуществить инициализацию набора. Таким образом, даже если у вас есть только один сервер для запуска, возможно, у вас возникнет желание настроить его как набор реплик с одним членом. Таким образом, если вы хотите добавить других членов позже, можно сделать это без простоев.

Если вы запускаете абсолютно новый набор, можно отправлять конфигурацию любому члену набора. Если вы начинаете с данных об одном из членов, то должны отправить конфигурацию члену с данными. Нельзя инициировать набор реплик с данными более чем для одного члена.

После инициализации у вас должен быть полностью функциональный набор реплик. Набор реплик должен выбрать первичный узел. Вы можете просмотреть состояние набора реплик, используя функцию `rs.status()`. Вывод этой функции довольно много говорит нам о наборе реплик, в том числе о тех вещах, которые мы еще не рассматривали, но не волнуйтесь, мы дойдем до этого! А пока взгляните на массив `members`. Обратите внимание, что в этом массиве перечислены все три наших экземпляра *mongod* и что один из них, в данном случае экземпляр, работающий на порту 27017, был выбран первичным. Два других являются вторичными. Если вы опробуете этот вариант, у вас наверняка будут разные значения `"date"` и несколько значений `Timestamp` в этих выходных данных, однако вы также можете обнаружить, что другой экземпляр *mongod* был выбран первичным (это совершенно нормально):

```
> rs.status() {
    "set": "mdbDefGuide",
    "date": ISODate("2017-07-27T20:23:31.457Z"),
    "myState": 1,
    "term": NumberLong(1),
    "heartbeatIntervalMillis": NumberLong(2000),
    "optimes": {
        "lastCommittedOpTime": {
            "ts": Timestamp(1501187006, 1),
            "t": NumberLong(1)
        },
        "appliedOpTime": {
            "ts": Timestamp(1501187006, 1),
            "t": NumberLong(1)
        },
        "durableOpTime": {
            "ts": Timestamp(1501187006, 1),
            "t": NumberLong(1)
        }
    },
    "members": [{
            "_id": 0,
            "name": "localhost:27017",
            "health": 1,
            "state": 1,
            "stateStr": "PRIMARY",
            "uptime": 688,
            "optime": { "ts": Timestamp(1501187006, 1), "t": NumberLong(1) },
            "optimeDate": ISODate("2017-07-27T20:23:26Z"),
```

```
            "electionTime": Timestamp(1501186514, 1),
            "electionDate": ISODate("2017-07-27T20:15:14Z"),
            "configVersion": 1,
            "self": true
        },
        {
            "_id": 1,
            "name": "localhost:27018",
            "health": 1,
            "state": 2,
            "stateStr": "SECONDARY",
            "uptime": 508,
            "optime": { "ts": Timestamp(1501187006, 1), "t": NumberLong(1) },
            "optimeDurable": { "ts": Timestamp(1501187006, 1), "t": NumberLong(1) },
            "optimeDate": ISODate("2017-07-27T20:23:26Z"),
            "optimeDurableDate": ISODate("2017-07-27T20:23:26Z"),
            "lastHeartbeat": ISODate("2017-07-27T20:23:30.818Z"),
            "lastHeartbeatRecv": ISODate("2017-07-27T20:23:30.113Z"),
            "pingMs": NumberLong(0),
            "syncingTo": "localhost:27017",
            "configVersion": 1
        },
        {
            "_id": 2,
            "name": "localhost:27019",
            "health": 1,
            "state": 2,
            "stateStr": "SECONDARY",
            "uptime": 508,
            "optime": { "ts": Timestamp(1501187006, 1), "t": NumberLong(1) },
            "optimeDurable": { "ts": Timestamp(1501187006, 1), "t": NumberLong(1) },
            "optimeDate": ISODate("2017-07-27T20:23:26Z"),
            "optimeDurableDate": ISODate("2017-07-27T20:23:26Z"),
            "lastHeartbeat": ISODate("2017-07-27T20:23:30.818Z"),
            "lastHeartbeatRecv": ISODate("2017-07-27T20:23:30.113Z"),
            "pingMs": NumberLong(0),
            "syncingTo": "localhost:27017",
            "configVersion": 1
        }
    ],
    "ok": 1,
    "operationTime": Timestamp(1501187006, 1)
}
```

Вспомогательные функции rs

`rs` – глобальная переменная, которая содержит вспомогательные функции репликации (запустите функцию `rs.help()`, чтобы увидеть информацию о них). Эти функции почти всегда являются просто обертками для команд базы данных. Например, эта команда базы данных эквивалентна `rs.initiate(config)`:

```
> db.adminCommand({"replSetInitiate" : config})
```

Хорошо иметь представление о вспомогательных функциях, равно как и об основных командах, поскольку возможно проще использовать команду вместо функции.

Наблюдение за репликацией

Если ваш набор реплик выбрал *mongod* на порту 27017 в качестве первичного узла, то оболочка *mongo*, используемая для запуска набора реплик, в настоящее время подключена к первичному серверу. Вы должны увидеть, что строка приглашения к вводу изменилась и теперь выглядит примерно так:

```
mdbDefGuide:PRIMARY>
```

Это указывает на то, что мы подключены к первичному узлу, имеющему с `"_id" "mdbDefGuide"`. Чтобы было проще и понятнее, в примерах репликации мы сократим строку приглашения оболочки *mongo* до знака >.

Если ваш набор реплик выбрал другой первичный узел, выйдите из оболочки и подключитесь к нему, указав в командной строке правильный номер порта, как мы это делали при запуске оболочки *mongo* ранее. Например, если основной сервер вашего набора находится на порту 27018, подключитесь к нему с помощью следующей команды:

```
$ mongo --port 27018
```

Теперь, когда вы подключены к серверу, попробуйте выполнить несколько операций записи и посмотрите, что произойдет.

Сначала вставим 1000 документов:

```
> use test
> for (i=0; i<1000; i++) {db.coll.insert({count: i})}
>
> // Убедитесь, что документы на месте;
> db.coll.count()
1000
```

Теперь проверьте один из вторичных узлов и убедитесь, что у него есть копия всех этих документов. Это можно сделать, выйдя из оболочки и подключившись с использованием номера порта одного из вторичных узлов, но соединение с одним из вторичных узлов легко получить, создав экземпляр объекта подключения с помощью конструктора Mongo в уже запущенной оболочке.

Для начала используйте свое соединение с тестовой базой данных на основном сервере для запуска команды `isMaster`. Это покажет вам состояние набора реплик в гораздо более сжатой форме, по сравнению с функцией `rs.status()`. Это также удобный способ определить, какой член является первичным узлом при написании кода приложения или сценариев:

```
> db.isMaster() {
    "hosts": [
        "localhost:27017",
        "localhost:27018",
        "localhost:27019"
    ],
    "setName": "mdbDefGuide",
    "setVersion": 1,
    "ismaster": true,
    "secondary": false,
    "primary": "localhost:27017",
    "me": "localhost:27017",
    "electionId": ObjectId("7fffffff0000000000000004"),
    "lastWrite": {
        "opTime": {
            "ts": Timestamp(1501198208, 1),
            "t": NumberLong(4)
        },
        "lastWriteDate": ISODate("2017-07-27T23:30:08Z")
    },
    "maxBsonObjectSize": 16777216,
    "maxMessageSizeBytes": 48000000,
    "maxWriteBatchSize": 1000,
    "localTime": ISODate("2017-07-27T23:30:08.722Z"),
    "maxWireVersion": 6,
    "minWireVersion": 0,
    "readOnly": false,
    "compression": ["snappy"],
    "ok": 1,
    "operationTime": Timestamp(1501198208, 1)
}
```

Если в какой-то момент назначаются выборы и экземпляр *mongod*, к которому вы подключены, становится вторичным, можно использовать команду `isMaster`, чтобы определить, какой член стал первичным. Здесь вывод говорит нам, что *localhost:27018* и *localhost:27019* являются вторичными, поэтому мы можем использовать любой из них для наших целей. Давайте создадим соединение с *localhost:27019*:

```
> secondaryConn = new Mongo("localhost:27019")
connection to localhost:27019
>
> secondaryDB = secondaryConn.getDB("test")
Test
```

Теперь если мы попытаемся выполнить операцию чтения для коллекции, которая была реплицирована во вторичный узел, то получим ошибку. Давайте попробуем найти эту коллекцию, а затем рассмотрим эту ошибку и выясним, почему мы ее получаем:

```
> secondaryDB.coll.find()
Error: error: {
    "operationTime": Timestamp(1501200089, 1),
    "ok": 0,
    "errmsg": "not master and slaveOk=false",
    "code": 13435,
    "codeName": "NotMasterNoSlaveOk"
}
```

Вторичные узлы могут отставать от первичных, и у них может не быть самых последних записей, поэтому по умолчанию они будут отклонять запросы на чтение, чтобы приложения не могли случайно прочитать устаревшие данные. Таким образом, если вы попытаетесь выполнить запрос к вторичному узлу, то получите сообщение о том, что это не первичный узел. Это необходимо для того, чтобы защитить ваше приложение от случайного подключения ко вторичному узлу и не дать ему прочитать устаревшие данные. Чтобы разрешить запросы к вторичному узлу, можно установить флаг «Все в порядке с чтением из вторичного», например:

```
> secondaryConn.setSlaveOk()
```

Обратите внимание, что `slaveOk` устанавливается для *соединения* (`secondaryConn`), а не для базы данных (`secondaryDB`).

Теперь все готово для выполнения операции чтения. Делаем запрос, как обычно:

```
> secondaryDB.coll.find()
{ "_id" : ObjectId("597a750696fd35621b4b85db"), "count" : 0 }
```

```
{ "_id" : ObjectId("597a750696fd35621b4b85dc"), "count" : 1 }
{ "_id" : ObjectId("597a750696fd35621b4b85dd"), "count" : 2 }
{ "_id" : ObjectId("597a750696fd35621b4b85de"), "count" : 3 }
{ "_id" : ObjectId("597a750696fd35621b4b85df"), "count" : 4 }
{ "_id" : ObjectId("597a750696fd35621b4b85e0"), "count" : 5 }
{ "_id" : ObjectId("597a750696fd35621b4b85e1"), "count" : 6 }
{ "_id" : ObjectId("597a750696fd35621b4b85e2"), "count" : 7 }
{ "_id" : ObjectId("597a750696fd35621b4b85e3"), "count" : 8 }
{ "_id" : ObjectId("597a750696fd35621b4b85e4"), "count" : 9 }
{ "_id" : ObjectId("597a750696fd35621b4b85e5"), "count" : 10 }
{ "_id" : ObjectId("597a750696fd35621b4b85e6"), "count" : 11 }
{ "_id" : ObjectId("597a750696fd35621b4b85e7"), "count" : 12 }
{ "_id" : ObjectId("597a750696fd35621b4b85e8"), "count" : 13 }
{ "_id" : ObjectId("597a750696fd35621b4b85e9"), "count" : 14 }
{ "_id" : ObjectId("597a750696fd35621b4b85ea"), "count" : 15 }
{ "_id" : ObjectId("597a750696fd35621b4b85eb"), "count" : 16 }
{ "_id" : ObjectId("597a750696fd35621b4b85ec"), "count" : 17 }
{ "_id" : ObjectId("597a750696fd35621b4b85ed"), "count" : 18 }
{ "_id" : ObjectId("597a750696fd35621b4b85ee"), "count" : 19 }
Type "it" for more
```

Видно, что все наши документы здесь.

Теперь попробуем сделать запись во вторичный узел:

```
> secondaryDB.coll.insert({"count" : 1001})
WriteResult({ "writeError" : { "code" : 10107, "errmsg" : "not master" } })
> secondaryDB.coll.count()
1000
```

Видно, что вторичный узел не принимает операции записи. Он будет выполнять только те операции записи, что получены через репликацию, а не от клиентов.

Есть еще одна интересная функция, которую стоит опробовать: автоматическое переключение при сбое. Если первичный узел отключается, один из вторичных автоматически будет выбран первичным.

Чтобы протестировать, как это работает, остановите первичный узел:

```
> db.adminCommand({"shutdown" : 1})
```

Вы увидите сообщения об ошибках, сгенерированные при запуске этой команды, поскольку экземпляр *mongod*, работающий на порту 27017 (член, к которому мы подключены), прекратит работу, а оболочка, которую мы используем, потеряет соединение:

```
2017-07-27T20:10:50.612-0400 E QUERY [thread1] Error: error doing query:
  failed: network error while attempting to run command 'shutdown' on host
```

```
  '127.0.0.1:27017' :
DB.prototype.runCommand@src/mongo/shell/db.js:163:1
DB.prototype.adminCommand@src/mongo/shell/db.js:179:16
@(shell):1:1
2017-07-27T20:10:50.614-0400 I NETWORK [thread1] trying reconnect to
  127.0.0.1:27017 (127.0.0.1) failed
2017-07-27T20:10:50.615-0400 I NETWORK [thread1] reconnect
  127.0.0.1:27017 (127.0.0.1) ok
MongoDB Enterprise mdbDefGuide:SECONDARY>
2017-07-27T20:10:56.051-0400 I NETWORK [thread1] trying reconnect to
  127.0.0.1:27017 (127.0.0.1) failed
2017-07-27T20:10:56.051-0400 W NETWORK [thread1] Failed to connect to
  127.0.0.1:27017, in(checking socket for error after poll), reason:
Connection refused
2017-07-27T20:10:56.051-0400 I NETWORK [thread1] reconnect
  127.0.0.1:27017 (127.0.0.1) failed failed
MongoDB Enterprise >
MongoDB Enterprise > secondaryConn.isMaster()
2017-07-27T20:11:15.422-0400 E QUERY [thread1] TypeError:
  secondaryConn.isMaster is not a function :
@(shell):1:1
```

Это не проблема. К вылету оболочки это не приведет. Выполните команду `isMaster` на вторичном узле, чтобы увидеть, кто стал новым первичным узлом:

```
> secondaryDB.isMaster()
```

Вывод команды `isMaster` должен выглядеть примерно так:

```
{
    "hosts": [
        "localhost:27017", "localhost:27018", "localhost:27019"
    ],
    "setName": "mdbDefGuide",
    "setVersion": 1,
    "ismaster": true,
    "secondary": false,
    "primary": "localhost:27018",
    "me": "localhost:27019",
    "electionId": ObjectId("7fffffff0000000000000005"),
    "lastWrite": {
        "opTime": {
            "ts": Timestamp(1501200681, 1),
            "t": NumberLong(5)
        },
        "lastWriteDate": ISODate("2017-07-28T00:11:21Z")
```

```
    },
    "maxBsonObjectSize": 16777216,
    "maxMessageSizeBytes": 48000000,
    "maxWriteBatchSize": 1000,
    "localTime": ISODate("2017-07-28T00:11:28.115Z"),
    "maxWireVersion": 6,
    "minWireVersion": 0,
    "readOnly": false,
    "compression": ["snappy"],
    "ok": 1,
    "operationTime": Timestamp(1501200681, 1)
}
```

Обратите внимание, что первичный узел переключился на 27018. Ваш узел может быть другим; независимо от того, какой вторичный узел заметил, что первичный узел не работает, будет выбран первый. Теперь вы можете отправлять записи на новый первичный узел.

`isMaster` – очень старая команда, которая использовалась еще до появления наборов реплик, когда MongoDB поддерживала только репликацию по типу *master – slave*. Таким образом, она не использует терминологию набора реплик последовательно и по-прежнему называет первичный узел «ведущим устройством». Обычно можно рассматривать «ведущее устройство» как «первичный узел», а «ведомое устройство» – как «вторичный».

Продолжим и вернем сервер, который у нас был на *localhost:27017*. Вам просто нужно найти интерфейс командной строки, из которого вы его запустили. Вы увидите некие сообщения, указывающие на то, что он прекратил работу. Просто запустите его снова, используя ту же команду, с помощью которой вы запускали его изначально.

Поздравляем! Вы просто настроили, использовали и даже немного «попинали» набор реплик, чтобы принудительно завершить работу и выбрать новый первичный узел.

Необходимо помнить несколько ключевых моментов:

- клиенты могут отправлять первичному узлу все те же операции, что и автономному серверу (чтение, запись, команды, построения индекса и т. д.);
- клиенты не могут выполнять операции записи во вторичные узлы;
- клиенты по умолчанию не могут выполнять операции чтения из вторичных узлов. Можно активировать такую возможность, явно

установив для соединения параметр «Я знаю, что я читаю из вторичного».

Изменение настройки набора реплик

Настройки набора реплик можно изменить в любое время: членов можно добавить, удалить или поменять. Для выполнения некоторых распространенных операций существуют вспомогательные функции оболочки. Например, чтобы добавить нового члена в набор, можно использовать функцию `rs.add`:

```
> rs.add("localhost:27020")
```

Точно так же можно удалить членов:

```
> rs.remove("localhost:27017")
{ "ok" : 1, "operationTime" : Timestamp(1501202441, 2) }
```

Вы можете проверить, что перенастройка прошла успешно, запустив функцию `rs.config()` в оболочке. Будет выведена текущая конфигурация:

```
> rs.config() {
    "_id": "mdbDefGuide",
    "version": 3,
    "protocolVersion": NumberLong(1),
    "members": [{
            "_id": 1,
            "host": "localhost:27018",
            "arbiterOnly": false,
            "buildIndexes": true,
            "hidden": false,
            "priority": 1,
            "tags": {},
            "slaveDelay": NumberLong(0),
            "votes": 1
        },
        {
            "_id": 2,
            "host": "localhost:27019",
            "arbiterOnly": false,
            "buildIndexes": true,
            "hidden": false,
            "priority": 1,
            "tags": {},
            "slaveDelay": NumberLong(0),
            "votes": 1
```

```
        },
        {
            "_id": 3,
            "host": "localhost:27020",
            "arbiterOnly": false,
            "buildIndexes": true,
            "hidden": false,
            "priority": 1,
            "tags": {},
            "slaveDelay": NumberLong(0),
            "votes": 1
        }
    ],
    "settings": {
        "chainingAllowed": true,
        "heartbeatIntervalMillis": 2000,
        "heartbeatTimeoutSecs": 10,
        "electionTimeoutMillis": 10000,
        "catchUpTimeoutMillis": -1,
        "getLastErrorModes": {},
        "getLastErrorDefaults": { "w": 1, "wtimeout": 0 },
        "replicaSetId": ObjectId("597a49c67e297327b1e5b116")
    }
}
```

Каждый раз, когда вы меняете конфигурацию, значение в поле `"version"` будет увеличиваться. Оно начинается с версии 1.

Вы также можете изменять существующих членов, а не просто добавлять и удалять их. Чтобы внести изменения, создайте нужный документ конфигурации в оболочке и вызовите функцию `rs.reconfig()`. Например, предположим, что у нас есть конфигурация, подобная этой:

```
> rs.config() {
    "_id": "testReplSet",
    "version": 2,
    "members": [{
            "_id": 0,
            "host": "198.51.100.1:27017"
        },
        {
            "_id": 1,
            "host": "localhost:27018"
        },
        {
            "_id": 2,
            "host": "localhost:27019"
```

```
        }
    ]
}
```

Кто-то случайно добавил члена 0 по IP-адресу, а не по имени хоста. Чтобы изменить это, сначала мы загружаем текущую конфигурацию в оболочку, а затем меняем соответствующие поля:

```
> var config = rs.config()
> config.members[0].host = "localhost:27017"
```

Теперь, когда документ конфигурации правильный, нам нужно отправить его в базу данных с помощью вспомогательной функции `rs.reconfig()`:

```
> rs.reconfig(config)
```

Часто функция `rs.reconfig()` более полезна, чем `rs.add()` и `rs.remove()` для сложных операций, таких как изменение конфигурации членов или добавление/удаление нескольких членов одновременно. Ее можно использовать для внесения любых допустимых изменений в конфигурацию: просто создайте документ, представляющий желаемую конфигурацию, и передайте его функции `rs.reconfig()`.

Проектирование набора

Чтобы спроектировать свой набор, существуют определенные понятия, с которыми вы должны быть знакомы. В следующей главе об этом рассказывается более подробно, но наиболее важным является то, что все наборы реплик связаны с большинством: необходимо большинство членов, чтобы выбрать первичный узел, который может оставаться таковым, пока может достигать большинства, и операция записи безопасна, когда она реплицируется на большинство. Это большинство определяется как «более половины всех членов в наборе», как показано в табл. 10.1.

Таблица 10.1. Что такое большинство?

Количество членов в наборе	Большинство набора
1	1
2	2
3	2
4	3
5	3
6	4
7	4

Обратите внимание, что не имеет значения, сколько членов отключено или недоступно; большинство основано на конфигурации набора.

Например, предположим, что у нас есть набор из пяти членов, и три из них отключены, как показано на рис. 10.1. Но два члена по-прежнему доступны. Эти два члена не могут достичь большинства в наборе (по крайней мере, трех членов), поэтому не могут выбрать первичный узел. Если бы один из них был первичным, он бы отошел, как только заметил, что не может достичь большинства. Через несколько секунд ваш набор будет состоять из двух вторичных узлов и трех недоступных членов.

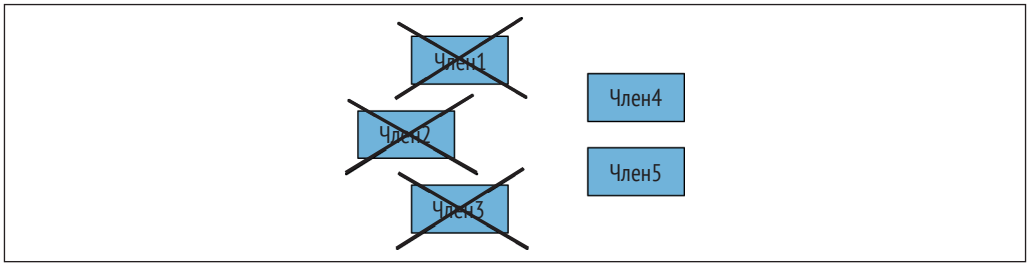

Рис. 10.1. При наличии меньшинства из набора все члены будут вторичными

У многих пользователей это вызывает разочарование: почему два оставшихся члена не могут выбрать первичный узел? Проблема состоит в том, что, возможно, три других члена фактически не отключились и что вместо этого вышла из строя сеть, как показано на рис. 10.2.

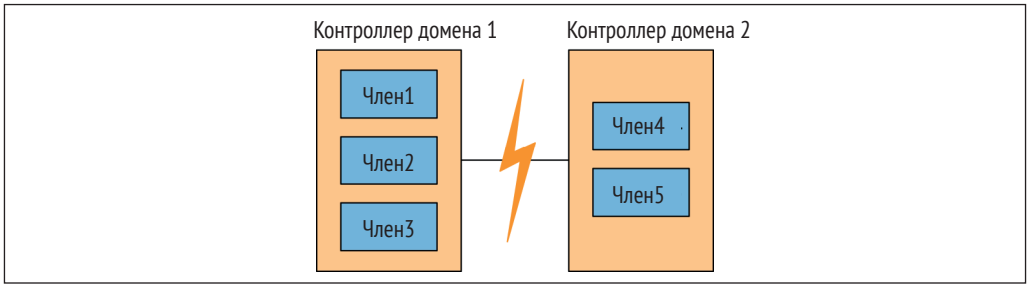

Рис. 10.2. Для членов отказ сети выглядит идентично серверам на другой стороне раздела, которые отключились

В этом случае три члена слева выберут первичный узел, так как они могут достичь большинства в наборе (три члена из пяти). В случае нарушения связности сети мы не хотим, чтобы обе стороны раздела выбирали первичный узел, потому что тогда в наборе их будет два. Оба будут вести запись в базу данных, и наборы данных будут расходиться. Требование, согласно которому нужно сделать выбор или оставить первичный узел, – изящный способ избежать появления других первичных узлов.

Важно настроить свой набор таким образом, чтобы обычно у вас мог быть один первичный узел. Например, в описанном здесь наборе из пяти

членов, если члены 1, 2 и 3 находятся в одном дата-центре, а члены 4 и 5 находятся в другом, почти всегда должно быть большинство, доступное в первом центре (вероятность выхода сети из строя между центрами обработки данных выше, чем внутри них).

Рекомендуется несколько распространенных конфигураций:

- большинство набора в одном дата-центре, как показано на рис. 10.2. Это хороший вариант, если у вас есть основной центр обработки данных, чтобы там, как вы всегда хотите, находился первичный узел вашего набора реплик. Пока ваш основной дата-центр работоспособен, у вас будет первичный узел. Однако если этот центр обработки данных станет недоступен, ваш вторичный дата-центр не сможет выбрать новый первичный узел;
- равное количество серверов в каждом центре обработки данных и еще один сервер в третьем месте. Это хороший вариант, если ваши дата-центры «одинаковы» в предпочтениях, поскольку обычно серверы любого центра обработки данных могут видеть большинство набора. Однако это предполагает наличие трех отдельных местоположений для серверов.

Для более сложных требований могут потребоваться другие конфигурации, но следует помнить, что ваш набор будет получать большинство при неблагоприятных условиях.

Все эти сложности исчезли бы, если бы MongoDB поддерживала наличие более чем одного первичного узла. Тем не менее это привело бы к очередным проблемам. Имея в наличии два первичных узла, вам придется обрабатывать конфликтующие операции записи (например, если кто-то обновляет документ на одном первичном узле, а на другом кто-то удаляет его). Существует два популярных способа обработки конфликтов в системах, которые поддерживают несколько методов записи: ручная сверка или произвольный выбор «победителя» системой. Ни один из этих вариантов не является очень простой моделью для разработчиков, поскольку нельзя быть уверенным в том, что данные, которые вы записали, не изменятся. Таким образом, MongoDB решила поддерживать наличие только одного первичного узла. Это облегчает разработку, но может привести к периодам, когда набор реплик доступен только для чтения.

Как работают выборы

Когда вторичный узел не может связаться с первичным, он свяжется со всеми остальными членами и потребует, чтобы его выбрали первичным. Остальные члены выполняют несколько проверок работоспособности: можно ли связаться с первичным узлом, с которым не может связаться член, желающий быть выбранным? Является ли этот член актуальным?

Есть ли член с более высоким приоритетом, которого можно выбрать взамен?

В версии 3.2 MongoDB представила версию протокола репликации 1. Версия протокола 1 основана на алгоритме RAFT для решения задач консенсуса, разработанном Диего Онгаро и Джоном Оустерхаутом из Стэнфордского университета. Лучше всего описывать его как RAFT-подобный и приспособленный для включения ряда концепций репликации, специфичных для MongoDB, таких как арбитры, приоритет, не участвующие в голосовании члены, гарантии записи и т. д. Протокол версии 1 обеспечил основу для новых функций, таких как более короткое время переключения при отказе, и значительно сокращает время обнаружения ложных ситуаций, а также предотвращает двойное голосование посредством использования идентификаторов терминов.

RAFT – это алгоритм для решения задач консенсуса, разбитый на относительно независимые подзадачи. Консенсус – это процесс, посредством которого несколько серверов или процессов согласовывают значения. RAFT обеспечивает консенсус таким образом, что одна и та же серия команд выдает одну и ту же серию результатов и поступает в одну и ту же серию состояний для всех членов развертывания.

Члены набора реплик отправляют друг другу тактовые сигналы (пинги) каждые две секунды. Если пинг не возвращается в течение 10 секунд, другие члены помечают правонарушителя как недоступного. Алгоритм выборов предпримет попытку «сделать все возможное», чтобы вторичный узел с наивысшим приоритетом объявил выборы. Приоритет членов влияет как на сроки, так и на результаты выборов; вторичные узлы с более высоким приоритетом назначают выборы относительно раньше, чем такие же узлы с более низким приоритетом, и они также с большей вероятностью победят. Однако экземпляр с более низким приоритетом может быть выбран в качестве основного на короткие периоды, даже если доступен вторичный член с более высоким приоритетом. Члены набора реплик продолжают назначать выборы до тех пор, пока доступный элемент с самым высоким приоритетом не станет первичным.

Чтобы быть избранным в качестве первичного узла, член должен быть в курсе репликации, насколько это известно членам, с которыми он может связаться. Все операции репликации строго упорядочены по восходящему идентификатору, поэтому у кандидата должны быть операции, которые позже или равны операциям любого члена, с которым он может связаться.

Параметры конфигурации членов

Наборы реплик, настройку которых мы проводили до сих пор, были достаточно однородными в том смысле, что у каждого члена есть та же конфигурация, что и любого другого члена. Однако существует множество ситуаций, когда вы не хотите, чтобы члены были идентичными: возможно, вы хотите, чтобы один из членов был преимущественно первичным, или хотите сделать члена невидимым для клиентов, чтобы запросы на чтение не могли быть направлены к нему. Эти и многие другие параметры конфигурации могут быть указаны во вложенных документах членов конфигурации набора реплик. В этом разделе описываются параметры членов, которые можно установить.

Приоритет

Приоритет является показателем того, насколько сильно этот член «хочет» стать первичным. Его значение может варьироваться от 0 до 100, а по умолчанию оно равно 1. Установка «приоритета» на 0 имеет особое значение: члены с приоритетом 0 никогда не смогут стать первичными. Они называются *пассивными* членами.

Член с наивысшим приоритетом всегда будет выбираться первичным (при условии что он может достигать большинства из набора и у него самые актуальные данные). Предположим, что вы добавляете в набор члена с приоритетом 1,5, например:

```
> rs.add({"host" : "server-4:27017", "priority" : 1.5})
```

Предполагая, что другие члены набора имеют приоритет 1, как только *server-4* будет синхронизирован с остальной частью набора, текущий первичный узел автоматически отключится, и *server-4* выберет самого себя. Если по какой-то причине *server-4* не смог синхронизироваться, текущий первичный узел останется первичным. Установка приоритетов никогда не приведет к тому, что ваш набор лишится первичных узлов, а также никогда не приведет к тому, что член, стоящий сзади, станет первичным (до тех пор, пока синхронизация не завершится).

Абсолютное значение `"priority"` имеет значение только в отношении того, больше или меньше других приоритетов в наборе: члены с приоритетами 100, 1 и 1 будут вести себя так же, как и члены другого набора с приоритетами 2, 1 и 1.

Скрытые члены

Клиенты не направляют запросы к скрытым членам, и те не являются предпочтительными в качестве источников репликации (хотя они и будут использоваться, если более желательные источники недоступны). Таким образом, многие будут скрывать менее мощные или резервные серверы.

Например, предположим, что у вас был набор, который выглядел так:

```
> rs.isMaster() {
    ...
    "hosts": [
        "server-1:27107",
        "server-2:27017",
        "server-3:27017"
    ],
    ...
}
```

Чтобы скрыть `server-3`, можно добавить поле `hidden: true` в его конфигурацию. Член должен иметь приоритет 0, чтобы быть скрытым (у вас не может быть скрытого первичного узла):

```
> var config = rs.config()
> config.members[2].hidden = true
0
> config.members[2].priority = 0
0
> rs.reconfig(config)
```

Теперь при выполнении команды `isMaster` мы увидим это:

```
> rs.isMaster() {
    ...
    "hosts": [
        "server-1:27107",
        "server-2:27017"
    ],
    ...
}
```

Функции `rs.status()` и `rs.config()` по-прежнему будут показывать член; он исчезает только из `isMaster`. Когда клиенты подключаются к набору реплик, они вызывают команду `isMaster` для определения членов набора. Таким образом, скрытые члены никогда не будут использоваться для запросов на чтение.

Чтобы член перестал быть скрытым, измените параметр `hidden` на `false` или полностью удалите его.

Арбитры

У набора из двух членов есть явные недостатки для требований большинства. Однако те, кто имеет дело с небольшими развертываниями,

не хотят хранить по три копии своих данных, считая, что двух достаточно, а хранение третьей копии не стоит административных, операционных и финансовых затрат.

Для этих развертываний MongoDB поддерживает специальный тип члена, *арбитр*, единственная цель которого – участвовать в выборах. Арбитры не хранят никаких данных и не используются клиентами: они просто предоставляют большинство для групп из двух членов. В целом развертывание без арбитров предпочтительнее.

Поскольку у арбитров нет традиционных обязанностей на сервере *mongod*, можно запускать арбитра в качестве более легковесного процесса на более простом сервере, чем вы обычно используете для MongoDB. Часто хорошей идеей, если это возможно, является запуск арбитра в области отказов, отдельной от других членов, чтобы у него был «взгляд со стороны» на набор, как описано в рекомендациях по развертыванию в разделе «Разработка набора».

Арбитр запускается так же, как и обычный *mongod*, используя опцию `--replSet имя` и пустой каталог данных. Можно добавить его в набор с помощью вспомогательной функции `rs.addArb()`:

```
> rs.addArb("server-5:27017")
```

Также можно указать опцию `"arbiterOnly"` в конфигурации члена:

```
> rs.add({"_id" : 4, "host" : "server-5:27017", "arbiterOnly" : true})
```

Арбитр, однажды добавленный в набор, останется арбитром навсегда: его нельзя перенастроить, чтобы он перестал быть арбитром, или наоборот.

Еще одна вещь, где полезны арбитры, – это разрыв связей в более крупных кластерах. Если у вас четное количество узлов, у вас может быть половина узлов, голосующих за одного члена, и половина за другого. Арбитр может отдать решающий голос. Однако при использовании арбитров следует иметь в виду несколько моментов, которые мы рассмотрим далее.

Используйте не более одного арбитра

Обратите внимание, что в обоих только что описанных случаях использования нужно *не более* одного арбитра. Если у вас нечетное количество узлов, арбитр не нужен. Распространено заблуждение, согласно которому следует добавлять дополнительных арбитров «на всякий случай». Однако это не помогает выборам проходить быстрее и не обеспечивает дополнительную безопасность данных для добавления дополнительных арбитров.

Предположим, у вас есть набор из трех членов. Два члена обязаны выбрать первичный узел.

Если вы добавите арбитра, у вас будет набор из четырех членов, поэтому для выбора первичного узла потребуется три члена. Таким образом, потенциально ваш набор менее стабилен: вместо 67 % теперь вам нужно 75 %.

Наличие дополнительных членов также может затянуть выборы. Если у вас есть четное количество узлов, потому что вы добавили арбитра, использование арбитров может приводить к ничьей, вместо того чтобы предотвращать ее.

Недостаток использования арбитра

Если перед вами стоит выбор между узлом данных и арбитром, выбирайте узел данных. Использование арбитра вместо узла данных в небольшом наборе может усложнить выполнение некоторых рабочих задач. Например, предположим, что вы запускаете набор реплик с двумя «обычными» членами и одним арбитром, и один из членов, хранящих данные, отключается. Если он абсолютно неработоспособен (данные не подлежат восстановлению), вам необходимо будет получить копию данных с текущего первичного узла на новый сервер, который вы будете использовать в качестве вторичного узла. Копирование данных может создать большую нагрузку на сервер и, таким образом, замедлить работу вашего приложения. (Как правило, копирование нескольких гигабайтов на новый сервер – обычное дело, но если их более ста, это становится нецелесообразным.)

И наоборот, если у вас есть три члена, хранящих данные, у вас будет больше «пространства», если сервер полностью выйдет из строя. Вы можете использовать оставшийся вторичный узел для начальной загрузки нового сервера, вместо того чтобы зависеть от вашего первичного узла.

В сценарии «два члена плюс арбитр» первичный узел является последней оставшейся хорошей копией ваших данных *и* тем, кто пытается обработать нагрузку из вашего приложения, в то время как вы пытаетесь получить еще одну копию ваших данных в сети.

Таким образом, если это возможно, используйте нечетное число «обычных» членов вместо арбитра.

В случае с наборами реплик с тремя членами с архитектурой «первичный узел – вторичный узел – арбитр» (PSA) или разделенными кластерами с PSA-шардом из трех членов существует известная проблема, связанная с увеличением давления на кеш хранилища, если один из двух узлов, несущих данные, не работает и гарантии чтения `"majority"` активированы. В идеале для этих развертываний следует заменить арбитра на член, несущий данные. В качестве альтернативы для предотвращения нехватки памяти в кеш-памяти гарантии чтения `"majority"` можно отключить (https://oreil.ly/p6nUm) в каждом из экземпляров mongod при развертывании или в шардах.

Построение индексов

Иногда вторичному узлу не нужно иметь те же (или любые) индексы, которые существуют на первичном узле. Если вы используете вторичный узел только для данных резервного копирования или автономных пакетных заданий, можно указать для `"buildIndexes"` значение `false` в настройках члена. Эта опция не дает вторичному узлу создавать индексы.

Это постоянный параметр: члены, у которых для `"buildIndexes"` установлено значение `false`, никогда нельзя будет повторно настроить как «обычные» члены построения индекса. Если вы хотите изменить член, не создающий индекс, на член, создающий индекс, нужно удалить его из набора, стереть все его данные, снова добавить его в набор и разрешить повторную синхронизацию с нуля.

Как и в случае со скрытыми членами, данный параметр требует, чтобы приоритет члена был равен 0.

Глава 11

Компоненты набора реплик

В этой главе рассказывается, как фрагменты набора реплик согласуются друг с другом, в том числе:

- как члены набора реплик реплицируют новые данные;
- как работает подключение новых членов;
- как работают выборы;
- возможные сценарии сбоя сервера и сети.

Синхронизация

Репликация связана с сохранением идентичной копии данных на нескольких серверах. MongoDB осуществляет ее путем ведения журнала операций, где содержится каждая запись, выполняемая первичным узлом. Это ограниченная коллекция, которая находится в *локальной* базе данных на первичном узле. Вторичные узлы запрашивают эту коллекцию для операций, которые будут реплицированы.

Каждый вторичный узел поддерживает свой журнал операций, записывая каждую операцию, которую он реплицирует с первичного узла. Это позволяет использовать любой член в качестве источника синхронизации для любого другого члена, как показано на рис. 11.1. Вторичные узлы выбирают операции у члена, с которым они синхронизируются, применяют эти операции к своему набору данных, а затем записывают их в свой журнал операций. Если применение операции завершается неудачно (что должно произойти, только если основные данные были повреждены или они каким-либо образом отличаются от данных первичного узла), вторичный узел завершит работу.

Если вторичный узел по какой-либо причине отключается, при перезапуске он начинает синхронизацию с последней операции в своем журнале операций. Поскольку операции применяются к данным, а затем записываются в журнал, вторичный узел может воспроизводить операции, которые он уже применил к своим данным. MongoDB разработана таким образом, чтобы справляться с этим правильно: многократное воспроиз-

ведение операций в журнале дает тот же результат, что и однократное их воспроизведение. Каждая операция в журнале операций идемпотентна. То есть операции из журнала дают одинаковые результаты, примененные один или несколько раз к целевому набору данных.

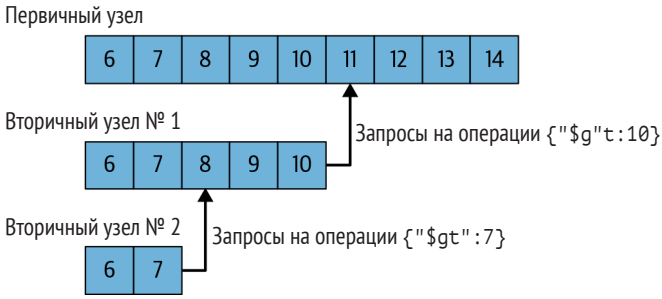

Рис. 11.1. В журналах операций хранится упорядоченный список операций записи, которые имели место быть; у каждого члена есть своя копия такого журнала, которая должна быть идентична копии первичного узла (некоторое отставание по модулю)

Поскольку журнал операций имеет фиксированный размер, он может содержать только определенное количество операций. В целом пространство в нем будет использоваться примерно с той же скоростью, с которой записи поступают в систему: если вы ведете запись со скоростью 1 КБ/мин на первичном узле, ваш журнал, вероятно, будет заполняться со скоростью около 1 КБ/мин. Однако есть несколько исключений: операции, которые влияют на несколько документов, таких как удаление или многократное обновление, будут разбиты на множество записей. Отдельная операция на первичном узле будет разбита на одну операцию для каждого затронутого документа. Таким образом, если вы удалите 1 000 000 документов из коллекции с помощью команды `db.coll.remove()`, получится 1 000 000 записей журнала операций, удаляющих один документ за раз. Если вы выполняете много массовых операций, ваш журнал может заполниться быстрее, чем вы ожидали.

В большинстве случаев размера журнала операций по умолчанию достаточно. Если вы можете предсказать, что рабочая нагрузка вашего набора реплик будет напоминать один из приведенных ниже шаблонов, можно создать журнал, который будет больше, чем тот, что по умолчанию. И наоборот, если ваше приложение преимущественно выполняет операции чтения с минимальным количеством операций записи, может потребоваться менее крупный журнал. Вот виды рабочих нагрузок, для которых может потребоваться журнал операций большего размера:

Обновление нескольких документов одновременно

Журнал операций должен транслировать многократные обновления в отдельные операции, чтобы поддерживать идемпотентность. Тут

может использоваться большое количество пространства журнала без соответствующего увеличения размера данных или использования диска.

Объем удаляемых данных равен объему данных, которые вставляются

Если вы удаляете примерно столько же данных, сколько вставляете, база данных не будет значительно увеличиваться с точки зрения использования диска, но размер журнала операций может быть довольно большим.

Значительное количество обновлений на месте

Если значительную часть рабочей нагрузки составляют обновления, которые не увеличивают размер документов, база данных записывает большое количество операций, но количество данных на диске не изменяется.

Прежде чем сервер *mongod* создаст журнал операций, можно указать его размер с помощью опции `oplogSizeMB`. Однако после того, как вы впервые запустите член набора реплик, изменить размер журнала можно только с помощью процедуры «Изменить размер журнала операций» (https://oreil.ly/mh5SX).

MongoDB использует две формы синхронизации данных: начальную синхронизацию для заполнения новых членов полным набором данных и репликацию для применения текущих изменений ко всему набору данных. Давайте подробнее рассмотрим каждую из них.

Начальная синхронизация

MongoDB выполняет начальную синхронизацию для копирования всех данных из одного члена набора реплик в другой. Когда член набора будет запущен, он проверит, находится ли он в допустимом состоянии, чтобы начать синхронизацию. Если он находится в допустимом состоянии, он попытается сделать полную копию данных от другого члена набора. У этого процесса есть несколько шагов, которым можно следовать в журнале *mongod*.

Выполняйте первоначальную синхронизацию для члена только в том случае, если вы не хотите, чтобы данные находились в вашем каталоге данных, или вы переместили их в другое место, поскольку первое действие, которое осуществляет *mongod*, – удалить их все.

Сперва MongoDB клонирует все базы данных, кроме локальной. *mongod* сканирует каждую коллекцию в каждой исходной базе данных и вставляет все данные в собственные копии этих коллекций на целевом члене. До

начала операций клонирования все существующие данные на нем будут удалены.

В MongoDB версии 3.4 и более поздних версиях при начальной синхронизации создаются все индексы коллекций, поскольку документы копируются для каждой коллекции (в более ранних версиях на этом этапе создавались только индексы `"_id"`). Также извлекаются вновь добавленные записи журналов операций во время копирования данных, поэтому вы должны убедиться, что у целевого члена достаточно дискового пространства в локальной базе данных, чтобы сохранять эти записи на этом этапе копирования данных.

Как только все базы данных будут клонированы, *mongod* использует журнал операций из источника для обновления своего набора данных, чтобы отразить текущее состояние набора реплик, применяя все изменения к набору данных, который появился во время копирования. Эти изменения могут включать в себя любой тип записи (вставки, обновления и удаления), и данный процесс может означать, что *mongod* должен клонировать заново определенные документы, которые были перемещены и, следовательно, пропущены.

Примерно так будут выглядеть журналы, если некоторые документы придется клонировать снова. В зависимости от уровня трафика и типов операций, которые происходят в источнике синхронизации, у вас могут быть или не быть пропущенные объекты:

```
Mon Jan 30 15:38:36 [rsSync] oplog sync 1 of 3
Mon Jan 30 15:38:36 [rsBackgroundSync] replSet syncing to: server-1:27017
Mon Jan 30 15:38:37 [rsSyncNotifier] replset setting oplog notifier to
    server-1:27017
Mon Jan 30 15:38:37 [repl writer worker 2] replication update of non-mod
    failed:
    { ts: Timestamp 1352215827000|17, h: -5618036261007523082, v: 2, op: "u",
      ns: "db1.someColl", o2: { _id: ObjectId('50992a2a7852201e750012b7') },
      o: { $set: { count.0: 2, count.1: 0 } } }
Mon Jan 30 15:38:37 [repl writer worker 2] replication info
    adding missing object
Mon Jan 30 15:38:37 [repl writer worker 2] replication missing object
    not found on source. presumably deleted later in oplog
```

На этом этапе данные должны точно соответствовать набору данных, поскольку они существовали в какой-то момент на первичном узле. Член завершает начальный процесс синхронизации и переходит к нормальной синхронизации, что позволяет ему стать вторичным узлом.

Начальная синхронизация очень проста с точки зрения оператора: просто запустите сервер *mongod* с чистым каталогом данных. Однако вместо этого часто предпочтительнее делать восстановление из резервной копии,

как описано в главе 23. Восстановление из резервной копии часто выполняется быстрее, чем копирование всех ваших данных через *mongod*.

Кроме того, клонирование может испортить рабочий набор источника синхронизации. Часто после развертываний появляются подмножества их данных, к которым нередко обращаются и которые всегда находятся в памяти (потому что ОС постоянно обращается к ним). Выполнение начальной синхронизации вынуждает члена просматривать все свои данные в памяти, высвобождая часто используемые данные. Это может значительно замедлить его работу, поскольку запросы, обрабатываемые данными в ОЗУ, внезапно вынуждены перейти на диск. Однако для небольших наборов данных и серверов с небольшим простором начальная синхронизация является хорошим и простым вариантом.

Одна из самых распространенных проблем, с которыми люди сталкиваются при начальной синхронизации, – на это уходит слишком много времени. В этих случаях может произойти следующее: член может настолько отстать от источника синхронизации, что больше не сможет синхронизироваться, потому что журнал операций источника синхронизации переписал данные, которые член должен был бы использовать, чтобы продолжить репликацию.

Исправить это невозможно, кроме как попытаться выполнить начальную синхронизацию в менее загруженное время или сделать восстановление из резервной копии. Начальную синхронизацию нельзя продолжить, если член выпал из журнала источника синхронизации. В разделе «Работа с устареванием данных» это описано более подробно.

Репликация

Второй тип синхронизации, которую выполняет MongoDB, – это репликация. Вторичные узлы непрерывно реплицируют данные после начальной синхронизации. Они копируют журнал операций из своего источника синхронизации и применяют эти операции в асинхронном процессе. Вторичные узлы могут автоматически изменять свой источник синхронизации из-за необходимости в ответ на изменения времени пинга и состояния репликации других участников. Существует несколько правил, определяющих, с какими членами данный узел может синхронизироваться. Например, члены набора реплик, имеющие один голос, не могут синхронизироваться с участниками, у которых нет голосов, а вторичные узлы избегают синхронизации с отложенными и скрытыми членами. Выборы и различные классы членов набора реплик обсуждаются в следующих разделах.

Работа с устареванием данных

Если вторичный узел слишком сильно отстает от фактических операций, выполняемых с источником синхронизации, он становится уста-

ревшим. Устаревший вторичный узел нельзя синхронизировать, потому что каждая операция в журнале операций источника синхронизации находится далеко впереди: он будет пропускать операции, если продолжит синхронизацию. Это может произойти, если вторичный узел простаивал, у него больше операций записи, чем он может обработать, или он слишком занят обработкой операций чтения.

Когда вторичный узел устареет, он попытается выполнить репликацию от каждого члена набора по очереди, чтобы узнать, есть ли кто-нибудь с более длинным журналом операций, с которого он может загрузиться. Если никого нет, репликация для этого члена будет остановлена, и его необходимо будет полностью синхронизировать повторно (или восстановить из более поздней резервной копии).

Чтобы избежать появления несинхронизированных вторичных узлов, важно иметь большой журнал операций, чтобы первичный узел мог хранить длинную историю операций. Очевидно, что больший журнал будет занимать больше дискового пространства, но в целом это хороший компромисс, поскольку дисковое пространство обычно дешево, и обычно используется небольшая часть журнала, поэтому он не будет занимать много оперативной памяти. Общее эмпирическое правило заключается в том, что журнал операций должен обеспечивать покрытие (окно репликации) на срок от двух до трех дней для нормальной работы. Для получения дополнительной информации о размерах журнала операций *см.* главу 13, раздел «Изменение размера журнала операций» на стр. 333.

Тактовые сигналы

Члены должны знать о состоянии других членов: кто является первичным узлом, с кем они могут синхронизироваться, а кто вышел из строя. Чтобы поддерживать актуальность представления набора, член отправляет тактовый сигнал каждому другому члену набора каждые две секунды. Тактовый сигнал – это короткое сообщение для проверки состояния.

Одна из наиболее важных функций тактовых сигналов – дать понять первичному узлу, сможет ли он достичь большинства из множества. Если первичный узел больше не может достичь большинства серверов, он понизит свой уровень и станет вторичным (*см.* главу 10, раздел «Проектирование набора» на стр. 287).

Состояния членов

С помощью тактовых сигналов члены также сообщают, в каком состоянии они находятся. Мы уже обсуждали два состояния: первичное и вторичное. Есть несколько других обычных состояний, в которых вы часто будете их встречать:

STARTUP

Это состояние, в котором находится член, когда он впервые запускается, когда MongoDB пытается загрузить свою конфигурацию набора реплик. Как только конфигурация будет загружена, он переходит в состояние STARTUP2.

STARTUP2

Это состояние длится в течение всего начального процесса синхронизации, который обычно занимает всего несколько секунд. Член создает пару потоков для обработки репликации и выборов, а затем переходит в следующее состояние: RECOVERING.

RECOVERING

Это состояние указывает на то, что член работает правильно, но недоступен для чтения. Это можно увидеть в разных ситуациях.

При запуске член должен сделать несколько проверок, чтобы убедиться, что он находится в допустимом состоянии, прежде чем принимать операции чтения; следовательно, все участники проходят кратковременное состояние RECOVERING при запуске, прежде чем стать вторичными узлами. Член также может перейти в это состояние во время длительных операций, таких как сжатие, или в ответ на команду `replSetMaintenance` (https://oreil.ly/6mJu-).

Член также перейдет в состояние RECOVERING, если он слишком отстал от других членов, чтобы синхронизироваться. Обычно это состояние сбоя, которое требует повторной синхронизации члена. На этом этапе он не переходит в состояние ошибки, потому что у него есть надежда, что кто-то выйдет в сеть с достаточно длинным журналом операций, чтобы он смог загрузиться сам.

ARBITER

Арбитры (*см.* раздел «Арбитры» в главе 10) имеют особое состояние и должны всегда находиться в этом состоянии во время нормальной работы.

Есть также несколько состояний, которые указывают на проблему с системой. Среди них:

DOWN

Если член был доступен, но затем стал недоступным, он перейдет в это состояние. Обратите внимание, что член, сообщивший о том, что он «недоступен», на самом деле фактически может быть доступен, просто его недоступность вызвана проблемами в работе сети.

UNKNOWN

Если какой-то из членов так и не смог связаться с другим членом, он не будет знать, в каком состоянии тот находится, поэтому он сообщит о нем как о НЕИЗВЕСТНОМ (*UNKNOWN*). Обычно это указывает на то, что неизвестный член недоступен или что между двумя членами возникли проблемы с сетью.

REMOVED

Это состояние члена, который был удален из набора. Если удаленный член добавляется обратно в набор, он возвращается в свое «нормальное» состояние.

ROLLBACK

Это состояние используется, когда член выполняет откат данных, как описано в разделе «Откаты». В конце процесса отката сервер переходит обратно в состояние RECOVERING, а затем станет вторичным узлом.

Выборы

Член будет претендовать на избрание, если не сможет связаться с первичным узлом (и сам по себе имеет право стать первичным узлом). Член, претендующий на избрание, отправит уведомление всем членам, с которыми он может связаться. Они могут знать, почему этот член является неподходящим кандидатом на роль первичного узла: он может отставать в репликации или, возможно, уже существует первичный узел, с которым член, претендующий на избрание, не может связаться. В этих случаях другие члены будут голосовать против этого кандидата.

Предполагая, что нет никаких оснований возражать, другие члены проголосуют за члена, претендующего на избрание. Если он получает голоса от большинства из множества, выборы считаются успешными, и член переходит в состояние PRIMARY. Если он не получил большинства голосов, он останется второстепенным узлом и может снова попытаться стать первичным позже. Первичный узел будет оставаться таковым до тех пор, пока не свяжется с большинством членов, станет недоступным или не будет понижен в ранге, или набор не будет перенастроен.

При условии что сеть исправна и большинство серверов доступны, выборы должны быть быстрыми. Члену потребуется до двух секунд, чтобы заметить, что первичный узел недоступен (из-за упомянутых ранее тактовых импульсов), и он сразу же начнет выборы, что должно занять всего несколько миллисекунд. Тем не менее ситуация часто бывает неоптимальной: выборы могут быть инициированы из-за проблем в работе сети или

перегруженных серверов, которые реагируют слишком медленно. В этих случаях выборы могут занять больше времени – даже до нескольких минут.

Откаты

Процесс выборов, описанный в предыдущем разделе, означает, что если первичный узел выполняет операцию записи и становится недоступен, прежде чем вторичные узлы успевают его реплицировать, следующий избранный первичный узел может не иметь права записи. Например, предположим, у нас есть два центра обработки данных, один с первичным и вторичным узлами, а другой с тремя вторичными узлами, как показано на рис. 11.2.

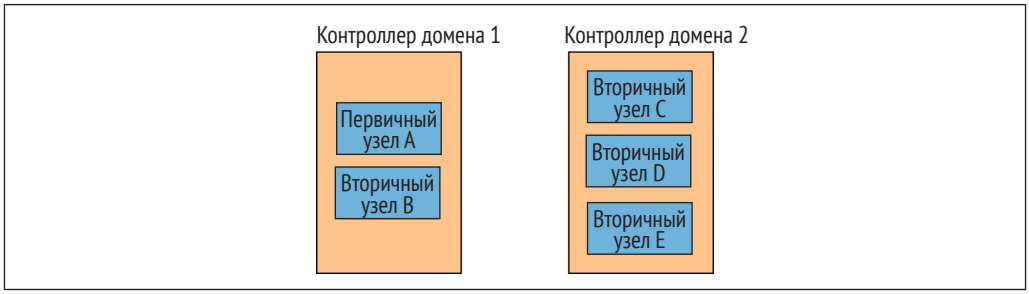

Рис. 11.2. Возможная конфигурация с двумя дата-центрами

Предположим, что существуют проблемы в работе сети между этими двумя центрами обработки данных, как показано на рис. 11.3. Серверы в первом ЦОД доступны до операции 126, но этот центр еще не реплицировал операции на серверы в другом центре обработки данных.

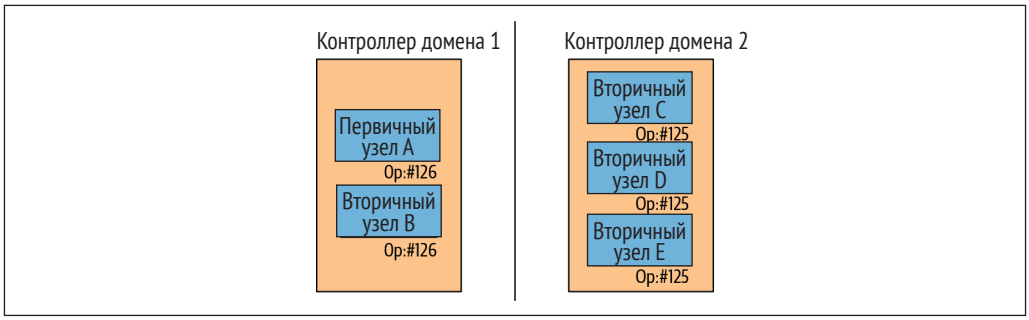

Рис. 11.3. Репликация через центры обработки данных может быть медленнее, чем внутри одного центра обработки данных

Серверы в другом центре обработки данных могут по-прежнему достигать большинства из набора (три из пяти серверов). Таким образом, один из них может быть избран первичным. Новый первичный узел начинает принимать собственные операции записи, как показано на рис. 11.4.

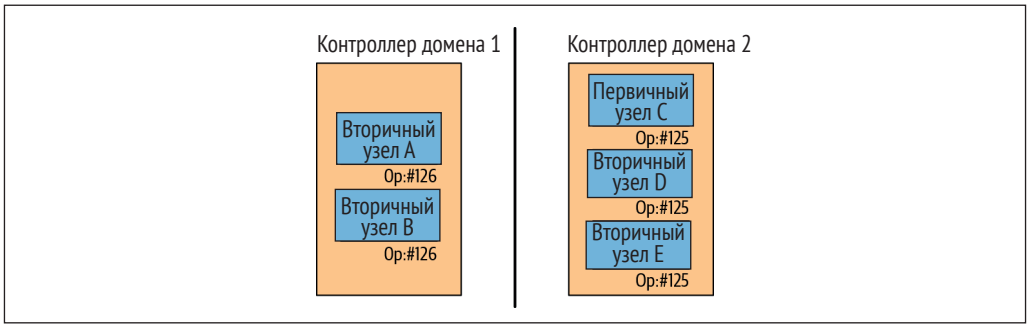

Рис. 11.4. Нереплицированные операции записи не будут совпадать с операциями записи на другой стороне сбоя сети

Когда работа сети восстанавливается, серверы в первом центре обработки данных будут искать операцию 126, чтобы начать синхронизацию с другими серверами, но не смогут ее найти. Когда это произойдет, *A* и *B* начнут процесс, называемый *откатом*. Откат используется для отмены операций, которые не были реплицированы до отработки отказа. Серверы с операцией 126 в своих журналах операций будут вести поиск в журналах серверов в другом центре обработки данных на предмет общей точки. Они обнаружат, что 125 является последней соответствующей операцией. На рис. 11.5 показано, как будут выглядеть журналы операций. *A*, очевидно, дал сбой до репликации операций 126–128, поэтому на *B*, у которого более поздние операции, эти операции отсутствуют. *A* придется откатить эти три операции, прежде чем возобновить синхронизацию.

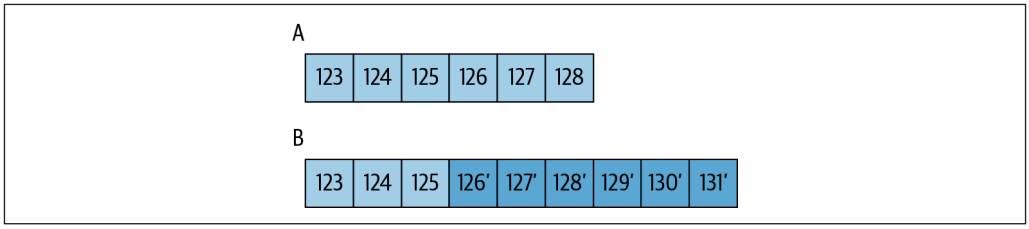

Рис. 11.5. Два члена с конфликтующими журналами операций, последняя общая операция – 125. Таким образом, поскольку у *B* более свежие операции, *A* нужно будет откатить операции 126–128

На этом этапе сервер выполнит операции и запишет свою версию каждого документа, затронутого этими операциями, в файл с расширением *.bson* в каталог *rollback* вашего каталога данных. Таким образом, если (например) операция 126 – это обновление, он запишет документ, обновленный 126, в файл *<Названиеколлекции>.bson*. Затем он скопирует версию этого документа из текущего первичного узла.

Ниже приводится вставка записей журнала, сгенерированных в результате типичного отката:

```
Fri Oct 7 06:30:35 [rsSync] replSet syncing to: server-1
Fri Oct 7 06:30:35 [rsSync] replSet our last op time written: Oct 7
    06:30:05:3
Fri Oct 7 06:30:35 [rsSync] replset source's GTE: Oct 7 06:30:31:1
Fri Oct 7 06:30:35 [rsSync] replSet rollback 0
Fri Oct 7 06:30:35 [rsSync] replSet ROLLBACK
Fri Oct 7 06:30:35 [rsSync] replSet rollback 1
Fri Oct 7 06:30:35 [rsSync] replSet rollback 2 FindCommonPoint
Fri Oct 7 06:30:35 [rsSync] replSet info rollback our last optime: Oct 7
    06:30:05:3
Fri Oct 7 06:30:35 [rsSync] replSet info rollback their last optime: Oct 7
    06:30:31:2
Fri Oct 7 06:30:35 [rsSync] replSet info rollback diff in end of log times:
    -26 seconds
Fri Oct 7 06:30:35 [rsSync] replSet rollback found matching events at Oct 7
    06:30:03:4118
Fri Oct 7 06:30:35 [rsSync] replSet rollback findcommonpoint scanned : 6
Fri Oct 7 06:30:35 [rsSync] replSet replSet rollback 3 fixup
Fri Oct 7 06:30:35 [rsSync] replSet rollback 3.5
Fri Oct 7 06:30:35 [rsSync] replSet rollback 4 n:3
Fri Oct 7 06:30:35 [rsSync] replSet minvalid=Oct 7 06:30:31 4e8ed4c7:2
Fri Oct 7 06:30:35 [rsSync] replSet rollback 4.6
Fri Oct 7 06:30:35 [rsSync] replSet rollback 4.7
Fri Oct 7 06:30:35 [rsSync] replSet rollback 5 d:6 u:0
Fri Oct 7 06:30:35 [rsSync] replSet rollback 6
Fri Oct 7 06:30:35 [rsSync] replSet rollback 7
Fri Oct 7 06:30:35 [rsSync] replSet rollback done
Fri Oct 7 06:30:35 [rsSync] replSet RECOVERING
Fri Oct 7 06:30:36 [rsSync] replSet syncing to: server-1
Fri Oct 7 06:30:36 [rsSync] replSet SECONDARY
```

Сервер начинает синхронизацию с другим членом (в данном случае это *server-1*) и понимает, что не может найти свою последнюю операцию на источнике синхронизации. В этот момент он запускает процесс отката, переходя в состояние ROLLBACK (`replSet ROLLBACK`).

В шаге 2 он находит общую точку между двумя журналами операций, которая была 26 секунд назад. Затем он начинает отменять операции за последние 26 секунд из своего журнала. Когда откат будет завершен, он переходит в состояние RECOVERING и снова начинает обычную синхронизацию.

Чтобы применить операции, откатившиеся к текущему первичному узлу, сначала используйте `mongorestore` для загрузки их во временную коллекцию:

```
$ mongorestore --db stage --collection stuff \
    /data/db/rollback/important.stuff.2018-12-19T18-27-14.0.bson
```

Затем изучите документы (используя оболочку) и сравните их с текущим содержимым коллекции, откуда они пришли. Например, если кто-то создал «обычный» индекс для члена отката и уникальный индекс для текущего первичного узла, вы должны убедиться, что в данных отката не было дубликатов, и разрешить их, если они были.

Когда в вашей промежуточной коллекции у вас появится версия документов, которая вам нравится, загрузите ее в свою основную коллекцию:

```
> staging.stuff.find().forEach(function(doc) {
... prod.stuff.insert(doc);
... })
```

Если у вас есть коллекции только для вставки, вы можете напрямую загрузить документы отката в коллекцию. Однако если вы выполняете операции обновления в коллекции, нужно быть более осторожным относительно того, как вы объединяете данные отката.

Один из часто неправильно используемых параметров конфигурации членов – это количество голосов, которое есть у каждого члена. Манипулирование количеством голосов почти всегда не то, что вам нужно, и приводит к большому количеству откатов (именно поэтому оно не было включено в список параметров конфигурации членов в последней главе). Не меняйте количество голосов, если вы не готовы иметь дело с регулярными откатами.

Для получения дополнительной информации о предотвращении откатов *см.* главу 12.

Когда откаты не работают

В более старых версиях MongoDB могла решить, что откат слишком велик для выполнения. Начиная с версии 4.0 в MongoDB нет ограничений на количество данных, которые можно откатить. Откат в версиях до 4.0 может завершиться неудачей, если для отката требуется более 300 МБ данных, или около 30 минут операций. В этих случаях необходимо выполнить повторную синхронизацию узла, который застрял при откате.

Наиболее частой причиной этого является то, что вторичные узлы отстают, а первичные недоступны. Если один из вторичных узлов станет первичным, он пропустит много операций со старого первичного сервера. Лучший способ убедиться в том, что вы не застряли в откате, – это поддерживать свои вторичные узлы по возможности в обновленном состоянии.

Глава 12

Подключение к набору реплик из своего приложения

В этой главе рассказывается, как приложения взаимодействуют с наборами реплик, в том числе:

- как работают соединения и отработки отказа;
- ожидание репликации при операциях записи;
- направление операций чтения правильному члену.

Как ведет себя соединение типа «клиент к набору реплик»

Клиентские библиотеки MongoDB («драйверы» на языке MongoDB) предназначены для управления обменом данными с серверами MongoDB независимо от того, является сервер автономным экземпляром MongoDB или набором реплик. Для наборов реплик по умолчанию драйверы будут подключаться к первичному узлу и направлять весь трафик на него. Ваше приложение может выполнять операции чтения и записи, как если бы оно разговаривало с автономным сервером, в то время как набор реплик тихо поддерживает горячие резервы в фоновом режиме.

Соединения с набором реплик аналогичны соединениям с отдельным сервером. Используйте класс `MongoClient` (или эквивалентный) в своем драйвере и предоставьте список сидов для драйвера, к которому нужно подключиться. Список сидов – это просто список адресов сервера. *Сиды (seeds)* представляют собой членов набора реплик, из которого ваше приложение будет читать данные и куда будет их записывать. Не нужно перечислять всех членов в начальном списке (хотя это можно сделать). Когда драйвер подключится к сидам, он обнаружит других членов. Строка подключения обычно выглядит примерно так:

```
"mongodb://server-1:27017,server-2:27017,server-3:27017"
```

См. документацию по своему драйверу для получения дополнительной информации.

Чтобы обеспечить дополнительную устойчивость, вы также должны использовать формат подключения DNS Seedlist (https://oreil.ly/Uq4za), чтобы указать, как ваши приложения подключаются к вашему набору реплик. Преимущество использования DNS заключается в том, что серверы, на которых размещаются члены набора реплик MongoDB, можно менять по очереди без необходимости перенастраивать клиентов (в частности, их строки подключения).

Все драйверы MongoDB соответствуют спецификации сервера обнаружения и мониторинга (SDAM) (https://oreil.ly/ZsS8p). Они постоянно следят за топологией вашего набора реплик, чтобы обнаружить любые изменения в способности вашего приложения связаться со всеми членами набора. Кроме того, драйверы следят за набором, чтобы сохранить информацию о том, какой член является первичным.

Цель наборов реплик состоит в том, чтобы ваши данные были доступны большую часть времени при выходе сети из строя или если серверы будут недоступны. При обычных обстоятельствах наборы реплик тонко реагируют на такие проблемы, выбирая новый первичный узел, чтобы приложения могли продолжать считывать и записывать данные. Если первичный узел не работает, драйвер автоматически найдет новый (после того как он будет выбран) и как можно скорее направит запросы к нему. Однако, пока нет доступного первичного узла, ваше приложение не сможет выполнять операции записи.

Первоначальный узел может быть недоступен в течение короткого времени (во время выбора) или в течение длительного периода времени (если доступный член не может стать первичным). По умолчанию драйвер не будет обслуживать какие-либо запросы – чтение или запись – в течение этого периода. При необходимости для вашего приложения вы можете настроить драйвер на использование вторичных узлов для запросов на чтение.

Как правило, хочется, чтобы драйвер скрывал весь процесс выборов (уход старого первичного узла и выбор нового) от пользователя. Тем не менее ни один драйвер не обрабатывает восстановление после отказа таким образом, и на это есть несколько причин. Во-первых, драйвер может только скрывать недостаток первичного узла так долго. Во-вторых, драйвер часто узнает, что первичный узел недоступен, потому что операция завершилась неудачно, а это означает, что драйвер не знает, обработал ли первичный узел операцию, прежде чем отключиться. Это фундаментальная проблема распределенных систем, которую невозможно избежать, поэтому нам нужна стратегия для ее решения, когда она возникнет. Должны ли мы повторить операцию на новом первичном узле, если он будет выбран быстро? Предположить, что это прошло на старом первичном узле? Проверить и посмотреть, работает ли новый первичный узел?

Оказывается, правильная стратегия – повторить попытку не более одного раза. А? Чтобы объяснить, давайте рассмотрим наши варианты. Они сводятся к следующему: не повторять, отказаться после того, как повторная попытка была предпринята фиксированное число раз, или повторять попытку не более одного раза. Мы также должны рассмотреть тип ошибки, которая может быть источником нашей проблемы. Существует три типа ошибок, которые можно увидеть при попытке записи в набор реплик: временная сетевая ошибка, постоянное отключение (сеть или сервер) или ошибка, вызванная командой, которую сервер отклоняет как неправильную (например, неавторизованную). Давайте рассмотрим наши варианты повторных попыток для каждого типа ошибки.

Ради этого обсуждения разберем пример записи с простым инкрементированием счетчика. Если наше приложение пытается увеличить наш счетчик, но не получает ответа от сервера, мы не знаем, получил ли сервер сообщение и выполнил ли обновление. Итак, если мы следуем стратегии не повторять эту операцию записи в случае с временной ошибкой сети, то можем просчитаться. В случае постоянного сбоя или ошибки команды отказ от повторения является правильной стратегией, потому что никакое количество повторных попыток операций записи не даст желаемого эффекта.

Если мы придерживаемся стратегии совершить повторную попытку фиксированное количество раз в случае с временной ошибкой сети, то можем перестараться (в случае успеха нашей первой попытки). В случае постоянного сбоя или ошибки команды неоднократные повторные попытки будут просто бесполезными.

Давайте теперь посмотрим на стратегию выполнения повторной попытки только один раз. В случае с временной ошибкой сети мы можем перестараться. В случае постоянного сбоя или ошибки команды это правильная стратегия. Однако что, если бы мы могли гарантировать, что наши операции идемпотентны? Идемпотентные операции – это операции, которые имеют один и тот же результат, независимо от того, выполняем ли мы их один или несколько раз. При идемпотентных операциях повторная попытка, когда речь идет об ошибках в сети, дает наилучшие шансы на правильное устранение всех трех типов ошибок.

Начиная с MongoDB версии 3.6 сервер и все драйверы MongoDB поддерживают вариант операций записи, которые можно попробовать выполнить повторно. Обратитесь к документации своего драйвера, чтобы получить подробную информацию о том, как использовать эту опцию. В случае с такими операциями записи драйвер автоматически будет следовать стратегии типа «повторная попытка не более одного раза». Ошибки команд будут возвращены приложению для обработки на стороне клиента. Ошибки сети будут повторяться один раз после соответствующей задержки, которая должна соответствовать выбору первичного узла при обычных обсто-

ятельствах. При включенной повторяющейся записи сервер поддерживает уникальный идентификатор для каждой операции записи и поэтому может определить, когда драйвер пытается повторить команду, которая уже выполнена успешно. Вместо того чтобы применить операцию снова, он просто вернет сообщение, указывающее на то, что операция записи успешно завершена, и тем самым преодолеет проблему, вызванную временной проблемой сети.

Ожидание репликации при операциях записи

В зависимости от потребностей вашего приложения вам может потребоваться, чтобы все операции записи реплицировались в большинство набора реплик, до того, как они будут подтверждены сервером. В редких случаях, когда первичный узел набора отключается, а вновь выбранный первичный узел (ранее вторичный) не реплицировал самые последние операции записи в прежний первичный узел, эти операции будут откатываться, когда прежний первичный узел снова станет доступен. Их можно восстановить, но это требует ручного вмешательства. Для многих приложений откат небольшого количества операции записей не является проблемой. Например, в приложении для блога нет реальной опасности откатить один или два комментария одного читателя.

Однако когда речь идет о других приложениях, следует избегать отката любых операций записей. Предположим, что ваше приложение отправляет операцию записи первичному узлу. Оно получает подтверждение, что операция был записана, но первичный узел дает сбой до того, как у вторичных узлов была возможность реплицировать эту операцию. Ваше приложение считает, что оно сможет получить доступ к этой операции, но у текущих членов набора реплик нет ее копии.

В какой-то момент вторичный узел может быть выбран первичным и начать принимать новые операции записи. Когда прежний первичный узел снова становится доступным, он обнаруживает, что у него есть операции записи, которых нет у текущего первичного узла. Чтобы исправить это, он отменит все операции записи, которые не соответствуют последовательности операций на текущем первичном узле. Эти операции не теряются, а записываются в специальные файлы отката, которые необходимо вручную применить к текущему первичному узлу. MongoDB не может применять эти операции записи автоматически, поскольку они могут конфликтовать с другими операциям записи, имевшими место после сбоя. Таким образом, операции записи, по сути, исчезают, пока администратор не получит возможность применить файлы отката к текущему первичному узлу (*см.* главу 11 для получения более подробной информации об откатах).

Требование выполнения записи в большинстве случаев предотвращает эту ситуацию: если приложение получает подтверждение того, что опе-

рация записи была успешной, то у нового первичного узла должна быть копия такой операции для избрания (член должен быть в курсе, чтобы его выбрали первичным узлом). Если приложение не получает подтверждение от сервера или получает сообщение об ошибке, оно будет знать, что нужно повторить попытку, потому что операция записи не была распространена на большую часть набора, перед тем как произошел сбой первичного узла.

Таким образом, для того чтобы гарантировать, что операции записи будут сохраняться независимо от того, что происходит с набором, мы должны гарантировать, что каждая операция распространяется на большинство членов набора. Этого можно добиться, используя `writeConcern`.

Начиная с MongoDB 2.6 `writeConcern` интегрирован с операциями записи. Например, в JavaScript можно использовать `writeConcern` следующим образом:

```
try {
    db.products.insertOne(
        { "_id": 10, "item": "envelopes", "qty": 100, type: "Self-Sealing" },
        { writeConcern: { "w": "majority", "wtimeout": 100 } }
    );
} catch (e) {
    print(e);
}
```

Конкретный синтаксис в вашем драйвере зависит от языка программирования, но семантика остается неизменной. В приведенном здесь примере мы указываем гарантии записи `"majority"`. В случае успеха сервер ответит следующим сообщением:

```
{ "acknowledged" : true, "insertedId" : 10 }
```

Но сервер не будет отвечать, пока эта операция записи не будет реплицирована в большинство членов набора реплик. Только тогда наше приложение получит подтверждение, что эта запись была успешной. Если запись не будет выполнена в течение указанного нами времени, сервер ответит сообщением об ошибке:

```
WriteConcernError({
    "code": 64,
    "errInfo": {
        "wtimeout": true
    },
    "errmsg": "waiting for replication timed out"
})
```

Гарантии записи и протокол выборов с участием наборов реплик гарантируют, что в случае выбора первичного узла первичными могут быть избраны только актуальные вторичные узлы, у которых есть подтвержденные операции записи. Таким образом, мы гарантируем, что откат не произойдет. При наличии опции тайм-аута у нас также есть настраиваемый параметр, который позволяет нам обнаруживать и отмечать любые длительные операции записи на уровне приложений.

Другие параметры для "w"

`"majority"` – не единственный параметр `writeConcern`. MongoDB также позволяет вам указать произвольное количество серверов для репликации, передав `"w"` число, как показано здесь:

```
db.products.insertOne(
    { "_id": 10, "item": "envelopes", "qty": 100, type: "Self-Sealing" },
    { writeConcern: { "w": 2, "wtimeout": 100 }
});
```

Тут мы будем ждать, пока два члена (первичный и один вторичный) не получат операцию записи.

Обратите внимание, что значение `"w"` включает в себя первичный узел. Если вы хотите, чтобы операция записи распространялась на *n* вторичных узлов, нужно установить для `"w"` значение *n* + 1 (чтобы включить первичный узел). `"w": 1` – то же самое, что вообще не передавать параметр `"w"`, поскольку так мы просто проверяем, что операция записи прошла успешно.

Недостатком использования буквального числа является то, что вы должны изменить свое приложение, если конфигурация вашего набора реплик изменяется.

Гарантии специализированной репликации

Запись в большинство набора считается «безопасной». Однако некоторые наборы могут иметь более сложные требования: вы можете захотеть убедиться, что запись выполняется как минимум на один сервер в каждом центре обработки данных или большинство нескрытых узлов. Наборы реплик позволяют вам создавать специальные правила, которые вы можете передать `"getLastError"`, чтобы гарантировать репликацию в любую необходимую вам комбинацию серверов.

По одному серверу на каждый центр обработки данных

Проблемы с сетью между центрами обработки данных встречаются гораздо чаще, чем внутри этих центров, и отключение всего дата-центра более вероятно, чем поверхностная работа серверов в нескольких ЦОД.

Таким образом, вам может понадобиться некая специфическая для центра обработки данных логика для операций записи. Гарантия отправки операции записи в каждый центр обработки данных перед подтверждением успеха означает, что если центр обработки данных переходит в автономный режим, во всех других дата-центрах будет находиться по крайней мере одна локальная копия.

Чтобы настроить это, мы сначала классифицируем членов по дата-центрам. Это можно сделать, если добавить поле `"tags"` к их конфигурации набора реплик:

```
> var config = rs.config()
> config.members[0].tags = {"dc" : "us-east"}
> config.members[1].tags = {"dc" : "us-east"}
> config.members[2].tags = {"dc" : "us-east"}
> config.members[3].tags = {"dc" : "us-east"}
> config.members[4].tags = {"dc" : "us-west"}
> config.members[5].tags = {"dc" : "us-west"}
> config.members[6].tags = {"dc" : "us-west"}
```

Поле `"tags"` является объектом, и у каждого члена может быть несколько тегов. Например, это может быть сервер «высокого качества» в центре обработки данных `"us-east"`, в этом случае нам нужно поле `"tags"`, например, такого вида: `{"dc": "us-east", "quality" : "high"}`.

Второй шаг – добавить правило, создав поле `"getLastErrorModes"` в нашей конфигурации набора реплик. Имя `"getLastErrorModes"` является рудиментарным в том смысле, что до появления MongoDB версии 2.6 приложения использовали метод `"getLastError"` для указания гарантий записи. В настройках реплик каждое правило для `"getLastErrorModes"` имеет форму *¨имя*`": {"`*ключ*`": `*число*`}}`. *"имя"* – это имя правила, которое должно описывать, что правило делает, чтобы клиенты могли его понять, поскольку они будут использовать это имя при вызове метода `getLastError`. В этом примере мы можем назвать это правило `"eachDC"` или как-нибудь более абстрактно, например `"user-level safe"`.

Поле *"ключ"* является ключевым полем из тегов, поэтому в этом примере это будет `"dc"`. *число* – это количество групп, необходимых для выполнения данного правила. В этом случае *число* равно 2 (потому что нам нужен по крайней мере один сервер от `"us-east"` и один от `"us-west"`). *число* всегда означает «хотя бы один сервер от каждой из *число* групп».

Мы добавляем поле `"getLastErrorModes"` в конфигурацию набора реплик, как показано ниже, и выполняем повторную настройку для создания правила:

```
> config.settings = {}
> config.settings.getLastErrorModes = [{"eachDC" : {"dc" : 2}}]
> rs.reconfig(config)
```

Поле `"getLastErrorModes"` находится во вложенном объекте `"settings"` конфигурации набора реплик, который содержит несколько необязательных настроек на уровне набора.

Теперь мы можем использовать это правило для операций записи:

```
db.products.insertOne(
    { "_id": 10, "item": "envelopes", "qty": 100, type: "Self-Sealing" },
    { writeConcern: { "w": "eachDC", wtimeout: 1000 } }
);
```

Обратите внимание, что правила в некоторой степени абстрагированы от разработчика приложения: ему не нужно знать, какие серверы находятся в `"eachDC"`, чтобы использовать правило, а правило может меняться без необходимости изменения приложения. Мы могли бы добавить центр обработки данных или изменить члены набора, и приложению не нужно об этом знать.

Гарантия большинства нескрытых членов

Часто скрытые члены – это словно граждане второго сорта: вам никогда не придется переходить к ним, и они, конечно же, не принимают никаких операций чтения. Таким образом, вы можете заботиться только о том, чтобы нескрытые члены получили операцию записи и позволили скрытым членам разобраться самим.

Предположим, у нас есть пять членов, с *host0* по *host4*. *host4* – скрытый член. Мы хотим убедиться, что большинство нескрытых членов имеют операцию записи, то есть, по крайней мере, три из *host0*, *host1*, *host2* и *host3*. Чтобы создать правило для этого, сначала мы помечаем каждого из нескрытых членов их собственным тегом:

```
> var config = rs.config()
> config.members[0].tags = [{"normal" : "A"}]
> config.members[1].tags = [{"normal" : "B"}]
> config.members[2].tags = [{"normal" : "C"}]
> config.members[3].tags = [{"normal" : "D"}]
```

Скрытому члену *host4* тег не присвоен.

Теперь мы добавим правило для большинства этих серверов:

```
> config.settings.getLastErrorModes = [{"visibleMajority" : {"normal" : 3}}]
> rs.reconfig(config)
```

Наконец, мы можем использовать это правило в нашем приложении:

```
db.products.insertOne(
{ "_id": 10, "item": "envelopes", "qty": 100, type: "Self-Sealing" },
```

```
{ writeConcern: { "w" : "visibleMajority", wtimeout : 1000 } }
);
```

Здесь мы будем ждать, пока по крайней мере три из нескрытых членов получат операцию записи.

Создание других гарантий

Правила, которые вы можете создавать, безграничны. Помните, что есть два шага для создания специализированного правила репликации:

1) пометьте члены, назначив им пары типа «ключ/значение». Ключи описывают классификации; например, у вас могут быть ключи `"data_center"`, или `"region"`, или `"serverQuality"`. Значения определяют, к какой группе принадлежит сервер в рамках классификации. Например, для ключа `"data_center"` у вас могут быть какие-то серверы с меткой `"us-east"`, какие-то с меткой `"us-west"` и какие-то с меткой `"aust"`;
2) напишите правило на основе создаваемых вами классификаций. Правила всегда имеют вид `{"`*имя*`": {"`*ключ*`": `*число*`}}`, где по крайней мере один сервер из *числа* групп должен иметь операцию записи, прежде чем ожидать успешного выполнения. Например, можно создать правило `{"twoDCs": {"data_center": 2}}`, которое будет означать, что хотя бы один сервер в двух дата-центрах с тегами должен подтвердить операцию записи, прежде чем она будет успешной.

Затем вы можете использовать это правило в `getLastErrorModes`.

Правила являются чрезвычайно мощным способом конфигурирования репликации, хотя их сложно понять и настроить. Если вы не задействовали требования репликации, то должны совершенно безопасно придерживаться `"w": "majority"`.

Отправка операций чтения на вторичные узлы

По умолчанию драйверы будут направлять все запросы на первичный узел. Как правило, это то, что вам нужно, но вы можете настроить другие параметры, установив предпочтения чтения в вашем драйвере. Предпочтения чтения позволяют указать типы серверов, на которые следует отправлять запросы.

Отправка запросов на чтение вторичным узлам – как правило, плохая идея. Существуют некоторые специфические ситуации, когда это имеет смысл, но, как правило, вы должны отправлять весь трафик первичному узлу. Если вы планируете отправлять операции чтения на вторичные узлы, тщательно взвесьте все «за» и «против», прежде чем разрешить это. В этом разделе описывается, почему это плохая идея, и конкретные условия, когда имеет смысл это делать.

Соображения по поводу согласованности

Приложения, которые требуют строго согласованных операций чтения, не должны выполнять чтение из вторичных узлов.

Вторичные узлы обычно должны находиться в рамках нескольких миллисекунд от первичного узла. Тем не менее нет никаких гарантий этого. Иногда вторичные узлы могут отставать на минуты, часы или даже дни из-за нагрузки, неправильной конфигурации, сетевых ошибок или других проблем. Клиентские библиотеки не могут сказать, насколько актуален вторичный узел, поэтому клиенты будут с удовольствием отправлять запросы вторичным узлам, которые находятся далеко позади. Можно спрятать вторичный узел от операций чтения клиента, но это ручной процесс. Таким образом, если вашему приложению нужны данные, которые предсказуемо актуальны, их не следует читать из вторичных узлов.

Если вашему приложению необходимо прочитать собственные записи (например, вставить документ, а затем запросить его и найти), не стоит отправлять операцию чтения на вторичный узел (только если операция записи не ожидает репликации во все вторичные узлы, используя «w», как было показано ранее). В противном случае приложение может выполнить успешную операцию записи, попытаться прочитать значение и не сможет найти его (потому что оно отправило операцию чтения на вторичный узел, который еще не реплицировал ее). Клиенты могут выдавать запросы быстрее, чем репликация может копировать операции.

Чтобы всегда отправлять запросы на чтение первичному узлу, установите для вашего предпочтения чтения значение `primary` (или оставьте его в покое, поскольку `primary` – это значение по умолчанию). Если первичного узла нет, запросы будут выдавать ошибку. Это означает, что ваше приложение не может выполнять запросы, если первичный узел не доступен. Тем не менее это, безусловно, приемлемый вариант, если ваше приложение может справляться с простоями во время отработки отказов или сбоев в работе сети или если получение устаревших данных неприемлемо.

Вопросы нагрузки

Многие пользователи отправляют операции чтения вторичным узлам для распределения нагрузки. Например, если ваши серверы могут обрабатывать только 10 000 запросов в секунду, а вам нужно обрабатывать 30 000, можно настроить пару вторичных узлов и заставить их принять на себя часть нагрузки. Однако это опасный способ масштабирования, поскольку можно с легкостью случайно перегрузить систему, а восстановить ее будет трудно.

Например, предположим, что у вас только что описанная ситуация: 30 000 операций чтения в секунду. Вы решаете создать набор реплик с четырьмя членами (один из них будет настроен как не участвующий в

голосовании, чтобы предотвратить ничью на выборах), чтобы справиться с этим: каждый вторичный узел значительно ниже своей максимальной нагрузки, и система работает отлично.

Пока один из вторичных узлов не вылетает.

Теперь каждый из оставшихся членов обрабатывает 100 % своей возможной нагрузки. Если вам нужно восстановить вышедшего из строя члена, возможно, потребуется скопировать данные с одного из других серверов, что приведет к перегрузке оставшихся серверов. Перегрузка сервера часто заставляет его работать медленнее, еще больше снижая емкость набора и заставляя других членов брать на себя большую нагрузку, заставляя их замедляться в смертельной спирали.

Перегрузка может также привести к замедлению репликации, в результате чего оставшиеся вторичные узлы будут отставать. Внезапно член становится недоступен, и появляется отставание, и все слишком перегружено, чтобы иметь какое-либо пространство для маневра.

Если у вас есть четкое представление о том, какую нагрузку может выдержать сервер, вам может показаться, что вы можете лучше спланировать это: используйте пять серверов вместо четырех, и при сбое одного из них набор не будет перегружен. Однако даже если вы все спланируете идеально (и потеряете только то количество серверов, которое ожидали), вам все равно придется исправлять ситуацию с другими серверами, находящимися под большим стрессом, чем они были бы в противном случае.

Более подходящим выбором является использование шардинга для распределения нагрузки. Мы расскажем, как настроить шардинг, в главе 14.

Причины чтения со вторичных узлов

Есть несколько случаев, когда разумно отправлять операции чтения вторичным узлам. Например, вы можете захотеть, чтобы ваше приложение по-прежнему могло выполнять операции чтения, если первичный узел не работает (и вам все равно, что эти операции могут быть несколько устаревшими). Это наиболее распространенный случай для распределения операций чтения вторичным узлам: вам нужен временный режим только для чтения, когда ваш набор лишается первичного узла. Это предпочтение чтения называется `primary Preferred`.

Одним из распространенных аргументов для чтения из вторичных узлов является получение операций чтения с низкой задержкой. В качестве предпочтения чтения можно указать `nearest` для маршрутизации запросов к элементу с наименьшей задержкой на основе среднего времени пинга от драйвера к члену набора реплик. Если вашему приложению необходим доступ к одному и тому же документу с низкой задержкой в нескольких дата-центрах, это единственный способ сделать это. Однако если ваши документы в большей степени основаны на расположении (серверам при-

ложений в этом центре обработки данных требуется доступ к некоторым вашим данным с низкой задержкой, или серверам приложений в другом центре обработки данных требуется доступ к другим данным с низкой задержкой), это следует сделать с помощью шардинга. Обратите внимание, что шардинг нужно использовать, если вашему приложению требуются операции чтения с малой задержкой *и* операции записи с малой задержкой: наборы реплик допускают делать операции записи только в одно местоположение (где находится первичный узел).

Вы должны быть готовы пожертвовать согласованностью, если читаете членов, которые еще не реплицировали все операции записи. В качестве альтернативы можно пожертвовать скоростью записи, если вы хотите подождать, пока операции записи не будут реплицированы на всех членов.

Если ваше приложение действительно может работать приемлемо с произвольно устаревшими данными, то можете использовать предпочтения чтения `secondary` или `secondaryPreferred`. `secondary` всегда будет отправлять запросы на чтение вторичному узлу. Если доступных вторичных узлов нет, это приведет к ошибке, а не к отправке операций чтения на первичный узел. Его можно использовать для приложений, которых не волнуют устаревшие данные и которые хотят использовать первичный узел только для операций записи. Если у вас есть какие-либо опасения по поводу устаревания данных, это не рекомендуется.

`secondaryPreferred` будет отправлять запросы на чтение вторичному узлу, если он доступен. Если вторичные узлы недоступны, запросы будут отправлены первичному.

Иногда нагрузка на чтение кардинально отличается от нагрузки на запись – то есть вы читаете совершенно другие данные, чем те, которые пишете. Возможно, вам понадобятся десятки индексов для автономной обработки, и вы не хотите, чтобы они находились на первичном узле. В этом случае вам может потребоваться настроить вторичный узел с индексами, отличными от первичного. Если вы хотите использовать для этой цели вторичный узел, вы, вероятно, создадите соединение непосредственно с ним из драйвера, вместо того чтобы использовать соединение с набором реплик.

Подумайте, какой из вариантов имеет смысл для вашего приложения. Вы также можете сочетать опции: если какие-то запросы на чтение должны идти от первичного узла, используйте для них `primary`. Если вас устраивает, что у других операций чтения нет самых актуальных данных, используйте для них `primaryPreferred`. А если для определенных запросов требуется низкая задержка по сравнению с согласованностью, используйте для них `nearest`.

Глава 13

Администрирование

В этой главе описывается администрирование набора реплик, в том числе:

- выполнение обслуживания отдельных членов;
- настройка наборов под различные обстоятельства;
- получение информации и изменение размера вашего журнала операций;
- выполнение некоторых более экзотических конфигураций набора;
- преобразование из *главный/подчиненный* в набор реплик.

Запуск членов в автономном режиме

Многие задачи по обслуживанию не могут быть выполнены на вторичных узлах (потому что они включают в себя операции записи), и они не должны выполняться на первичных узлах из-за влияния, которое это может оказать на производительность приложения. Таким образом, в следующих разделах часто упоминается запуск сервера в автономном режиме. Это означает перезапуск члена таким образом, чтобы это был автономный сервер, а не член набора реплик (временно).

Чтобы запустить его в автономном режиме, сначала посмотрите на параметры командной строки, использованные для его запуска. Предположим, они выглядят примерно так:

```
> db.serverCmdLineOpts() {
    "argv": ["mongod", "-f", "/var/lib/mongod.conf"],
    "parsed": {
        "replSet": "mySet",
        "port": "27017",
        "dbpath": "/var/lib/db"
    },
    "ok": 1
}
```

Чтобы выполнить обслуживание на этом сервере, мы можем перезапустить его без параметра `replSet`. Это позволит нам выполнять с ним операции чтения и записи, как в случае с обычным автономным сервером *mongod*. Мы не хотим, чтобы другие серверы в наборе могли связаться с ним, поэтому заставим его прослушивать иной порт (чтобы другие члены не смогли его найти). Наконец, мы хотим сохранить `dbpath` прежним, так как мы, вероятно, запускаем его таким образом, чтобы как-то манипулировать данными сервера.

Сначала мы останавливаем сервер из оболочки *mongo*:

```
> db.shutdownServer()
```

Затем в оболочке операционной системы (например, bash) мы перезапускаем *mongod* на другом порту без параметра `replSet`:

```
$ mongod --port 30000 --dbpath /var/lib/db
```

Теперь он будет работать как автономный сервер, прослушивая порт 30 000 на предмет соединений. Другие члены набора попытаются подключиться к нему через порт 27017 и будут предполагать, что он не работает.

Когда мы закончим выполнение обслуживания на сервере, то можем выключить его и перезапустить с его исходными параметрами. Он будет автоматически синхронизироваться с остальной частью набора, реплицируя любые операции, которые он пропустил, пока «отсутствовал».

Конфигурация набора реплик

Конфигурация набора реплик всегда хранится в документе в коллекции *local.system.replset*. Этот документ одинаков для всех членов набора. Никогда не обновляйте данный документ с помощью `update`. Всегда используйте вспомогательную функцию `rs` или команду `replSetReconfig`.

Создание набора реплик

Набор реплик создается путем запуска экземпляров *mongod*, которые вы хотите видеть в качестве членов, и затем путем передачи одному из них конфигурации с помощью метода `rs.initiate()`:

```
> var config = {
    ... "_id" : <setName>,
    ... "members" : [
    ... {"_id" : 0, "host" : <host1>},
    ... {"_id" : 1, "host" : <host2>},
    ... {"_id" : 2, "host" : <host3>}
    ... ]}
> rs.initiate(config)X
```

Всегда нужно передавать объект конфигурации функции `rs.initiate()`. Если вы этого не сделаете, MongoDB попытается автоматически сгенерировать конфигурацию для набора реплик с одним членом; он может не использовать желаемое имя хоста или правильно настроить набор.

Вы вызываете функцию `rs.initiate()` только для одного члена набора. Тот, кто получает конфигурацию, передает ее другим членам.

Изменение членов набора

Когда вы добавляете нового члена набора, у него либо не должно быть ничего в каталоге данных – в этом случае он выполнит начальную синхронизацию, – либо у него будет копия данных от другого члена (дополнительную информацию о резервном копировании и восстановлении членов набора реплик *см.* в главе 23).

Подключитесь к первичному узлу и добавьте нового члена:

```
> rs.add("spock:27017")
```

Кроме того, можно указать более сложную конфигурацию члена в качестве документа:

```
> rs.add ({"host": "spock: 27017", "priority": 0, "hidden": true})
```

Вы также можете удалять членов по полю `"host"`:

```
> rs.remove("spock:27017")
```

Можно изменить настройки члена путем повторного конфигурирования. Есть несколько ограничений в изменении настроек члена:

- нельзя изменить `"_id"` члена;
- нельзя назначить члена, которому вы отправляете перенастройку, с приоритетом (обычно это первичный узел) 0;
- нельзя превратить арбитра в не арбитра или наоборот;
- нельзя изменить значение поле члена `"buildIndexes"` с `false` на `true`.

В частности, *можно* изменить поле `"host"`. Таким образом, если вы неправильно указали хост (скажем, если вы используете открытый IP-адрес вместо закрытого), вы можете позже вернуться и просто изменить конфигурацию, чтобы использовать правильный IP.

Чтобы изменить имя хоста, можно сделать что-то вроде этого:

```
> var config = rs.config()
> config.members[0].host = "spock:27017"
```

```
spock:27017
> rs.reconfig(config)
```

Эта же стратегия применима к изменению любого другого параметра: извлеките конфигурацию с помощью функции `rs.config()`, измените нужные вам фрагменты и перенастройте набор, передав функции `rs.reconfig()` новую конфигурацию.

Создание более крупных наборов

Наборы реплик ограничены 50 членами в общей сложности и только 7 членами с правом голоса. Это делается для того, чтобы уменьшить объем сетевого трафика, необходимого при отправке таковых импульсов, и ограничить время, затрачиваемое на выборы.

Если вы создаете набор реплик, у которого более семи членов, каждому дополнительному члену должно быть отдано 0 голосов. Это можно сделать, указав его в конфигурации члена:

```
> rs.add({"_id" : 7, "host" : "server-7:27017", "votes" : 0})
```

Это препятствует тому, чтобы данные участники давали положительные голоса на выборах.

Принудительное переконфигурирование

Когда вы навсегда теряете большинство набора, то можете изменить конфигурацию набора, пока у него нет первичного узла. Это немного сложно, так как обычно вы отправляете перенастройку на первичный узел. В этом случае вы можете принудительно перенастроить набор, отправив соответствующую команду вторичному узлу. Подключитесь к вторичному узлу в оболочке и передайте ему перенастройку с помощью параметра `"force"`:

```
> rs.reconfig(config, {"force" : true})
```

Принудительное переконфигурирование следует тем же правилам, что и обычное: вы должны отправить валидную, правильно сформированную конфигурацию с правильными параметрами. Параметр `"force"` не разрешает недопустимые конфигурации; он просто позволяет вторичному узлу принять повторное конфигурирование.

Принудительное переконфигурирование значительно увеличивает номер «версии» набора реплик. Видно, как он перескакивает на десятки или сотни тысяч. Это нормально: это предотвращает конфликты номеров версий (на случай, если по обе стороны сетевого раздела есть реконфигурация).

Когда вторичный узел получает переконфигурирование, он обновляет свою конфигурацию и передает новую конфигурацию другим членам.

Другие члены набора будут воспринимать изменение конфигурации только в том случае, если они признают отправляющий сервер в качестве члена своей текущей конфигурации. Таким образом, если некоторые из ваших членов изменили имена хостов, вы должны принудительно выполнить переконфигурирование члена, который сохранил старое имя хоста. Если у каждого из членов новое имя хоста, нужно отключить всех членов набора, запустить нового в автономном режиме, вручную изменить его документ *local.system.replset*, а затем перезапустить.

Управление состоянием членов

Существует несколько способов вручную изменить состояние члена с целью обслуживания или в ответ на загрузку. Обратите внимание, что нет способа заставить члена стать первичным, кроме соответствующей настройки набора – в этом случае предоставив члену набора реплик приоритет выше, чем у любого другого члена набора.

Превращение первичных узлов во вторичные

Можно понизить первичный узел до вторичного, используя функцию `stepDown`:

```
> rs.stepDown()
```

Так мы понижаем первичный узел до состояния SECONDARY на 60 секунд. Если за этот период времени никаких других первичных узлов выбрано не будет, он сможет предпринять попытку переизбрания. Если вы хотите, чтобы он оставался вторичным в течение более длительного или более короткого промежутка времени, можно указать собственное количество секунд, в течение которого он должен оставаться в состоянии SECONDARY:

```
> rs.stepDown(600) // 10 минут;
```

Предотвращение выборов

Если вам необходимо выполнить обслуживание первичного узла, но вы не хотите, чтобы другие члены становились первичными в промежуточный период, можно заставить их оставаться вторичными, используя функцию `rs.freeze()`:

```
> rs.freeze(10000)
```

Опять же, на то, чтобы член оставался вторичным, уходит несколько секунд.

Если вы закончите обслуживание, которое выполняете на первичном узле, до истечения этого времени и захотите «разморозить» других членов, просто повторите команду для каждого из них, указав тайм-аут в 0 секунд:

```
> rs.freeze(0)
```

После этого такой член сможет провести выборы, если захочет.

Вы также можете «разморозить» первичные узлы, которые были понижены в ранге, выполнив функцию `rs.freeze(0)`.

Мониторинг репликации

Важно иметь возможность отслеживать состояние набора: не только то, что все члены активны, но и то, в каких состояниях они находятся и насколько актуальна репликация. Есть несколько команд, которые можно использовать для просмотра информации о наборе реплик. Такие платформы MongoDB, как Atlas, Cloud Manager и Ops Manager (*см.* главу 22), также предоставляют механизмы для мониторинга репликации и панели мониторинга по ключевым метрикам репликации.

Часто проблемы с репликацией являются временными: сервер не мог связаться с другим сервером, а теперь он может. Самый простой способ увидеть подобные проблемы – посмотреть журналы. Убедитесь, что вы знаете, где они хранятся (и в том, что они хранятся) и что вы можете получить к ним доступ.

Получение статуса

Одна из самых полезных команд, которую вы можете выполнить, – это `replSetGetStatus`. Она получает текущую информацию о каждом члене набора (с точки зрения члена, для которого вы ее выполняете). В оболочке существует вспомогательная функция для этой команды:

```
> rs.status()
    "set": "replset",
    "date": ISODate("2019-11-02T20:02:16.543Z"),
        "myState": 1, "term": NumberLong(1),
        "heartbeatIntervalMillis": NumberLong(2000),
        "optimes": {
            "lastCommittedOpTime": {
                "ts": Timestamp(1478116934, 1),
                "t": NumberLong(1)
            },
            "readConcernMajorityOpTime": {
                "ts": Timestamp(1478116934, 1),
                "t": NumberLong(1)
```

```
            },
            "appliedOpTime": {
                "ts": Timestamp(1478116934, 1),
                "t": NumberLong(1)
            },
            "durableOpTime": {
                "ts": Timestamp(1478116934, 1),
                "t": NumberLong(1)
            }
        },
    "members": [
        {
            "_id": 0,
            "name": "m1.example.net:27017",
            "health": 1,
            "state": 1,
            "stateStr": "PRIMARY",
            "uptime": 269,
            "optime": {
                "ts": Timestamp(1478116934, 1),
                "t": NumberLong(1)
        },
            "optimeDate": ISODate("2019-11-02T20:02:14Z"),
            "infoMessage": "could not find member to sync from",
            "electionTime": Timestamp(1478116933, 1),
            "electionDate": ISODate("2019-11-02T20:02:13Z"),
            "configVersion": 1,
            "self": true
        },
        {
            "_id": 1,
            "name": "m2.example.net:27017",
            "health": 1,
            "state": 2,
            "stateStr": "SECONDARY",
            "uptime": 14,
            "optime": {
                "ts": Timestamp(1478116934, 1),
                "t": NumberLong(1)
            },
            "optimeDurable": {
                "ts": Timestamp(1478116934, 1),
                "t": NumberLong(1)
            },
            "optimeDate": ISODate("2019-11-02T20:02:14Z"),
```

```
            "optimeDurableDate": ISODate("2019-11-02T20:02:14Z"),
            "lastHeartbeat": ISODate("2019-11-02T20:02:15.618Z"),
            "lastHeartbeatRecv": ISODate("2019-11-02T20:02:14.866Z"),
            "pingMs": NumberLong(0),
            "syncingTo": "m3.example.net:27017",
            "configVersion": 1
        },
        {
            "_id": 2,
            "name": "m3.example.net:27017",
            "health": 1,
            "state": 2,
            "stateStr": "SECONDARY",
            "uptime": 14,
            "optime": {
                "ts": Timestamp(1478116934, 1),
                "t": NumberLong(1)
            },
            "optimeDurable": {
                "ts": Timestamp(1478116934, 1),
                "t": NumberLong(1)
            },
            "optimeDate": ISODate("2019-11-02T20:02:14Z"),
            "optimeDurableDate": ISODate("2019-11-02T20:02:14Z"),
            "lastHeartbeat": ISODate("2019-11-02T20:02:15.619Z"),
            "lastHeartbeatRecv": ISODate("2019-11-02T20:02:14.787Z"),
            "pingMs": NumberLong(0),
            "syncingTo": "m1.example.net:27018",
            "configVersion": 1
            }
    ],
    "ok": 1
}
```

Вот некоторые из наиболее полезных полей:

`"self"`
: Это поле присутствует только в члене, для которого была выполнена функция `rs.status()` – в данном случае это *server-2(m1.example.net:27017)*.

`"stateStr"`
: Строка, описывающая состояние сервера. *См.* раздел «Состояния членов» в главе 11, где приводятся описания различных состояний.

`"uptime"`
: Количество секунд, в течение которых член был доступен, или время, прошедшее с момента запуска этого сервера для члена `"self"`. Таким образом, *server-1* был доступен в течение 269 секунд, а *server-2* и *server-3* – в течение 14 секунд.

`"optimeDate"`
: Последнее время операции в журнале операций каждого члена (где синхронизируется этот член). Обратите внимание, что это состояние каждого члена, о котором сообщает тактовый импульс, поэтому указанное здесь время операции может быть отключено на пару секунд.

`"lastHeartbeat"`
: Время, когда этот сервер в последний раз получал тактовый импульс от члена `"self"`. Если были проблемы с сетью или сервер был занят, оно может быть дольше, чем две секунды назад.

`"pingMs"`
: Скользящее среднее от того, насколько долго шли тактовые импульсы к этому серверу. Используется для определения того, с каким членом синхронизироваться.

`"errmsg"`
: Любое сообщение о статусе, которое член решил вернуть в запросе тактового импульса. Часто это просто сообщения информативного характера, а не сообщения об ошибках. Например, поле `"errmsg"` в *server-3* указывает на то, что этот сервер находится в процессе начальной синхронизации. Шестнадцатеричное число 507e9a30: 851 – это временная метка операции, которую должен получить этот член для завершения начальной синхронизации.

Есть несколько полей, которые дают перекрывающуюся информацию. `"state"` – то же самое, что и `"stateStr"`; это просто внутренний идентификатор состояния. `"health"` просто отражает, является ли данный сервер доступным (1) или недоступным (0), что также отображается с помощью полей `"state"` и `"stateStr"` (они будут иметь значение `UNKNOWN` или `DOWN`, если сервер недоступен). Аналогично, `"optime"` и `"optimeDate"` – это одно и то же значение, представленное двумя способами: одно обозначает миллисекунды от начала «эпохи» (`"t": 135 ...`), а другое – более читабельная дата.

Обратите внимание, что это отчет с точки зрения любого члена из набора, на котором вы его запускаете: информация, которая в нем содержится, может быть неверной или устаревшей из-за проблем с сетью.

Визуализация графика репликации

Если вы выполните функцию `rs.status()` на вторичном узле, появится поле верхнего уровня с именем `"syncingTo"`, что дает хост, который этот член реплицирует. Выполнив команду `replSetGetStatus` для каждого члена набора, вы сможете определить график репликации. Например, если предположить, что `server1` – это соединение с *server1*, `server2` – это соединение с *server2* и так далее, получится что-то вроде этого:

```
> server1.adminCommand({replSetGetStatus: 1})['syncingTo']
server0:27017
> server2.adminCommand({replSetGetStatus: 1})['syncingTo']
server1:27017
> server3.adminCommand({replSetGetStatus: 1})['syncingTo']
server1:27017
> server4.adminCommand({replSetGetStatus: 1})['syncingTo']
server2:27017
```

Таким образом, *server0* является источником репликации для *server1*, *server1* выступает источником репликации для *server2* и *server3*, а *server2* – источником репликации для *server4*.

MongoDB определяет, с кем синхронизироваться, основываясь на времени пинга. Когда один член отправляет тактовые импульсы другому, он фиксирует, сколько времени занимает этот запрос. MongoDB сохраняет скользящее среднее данных показателей. Когда член должен выбрать другого участника для синхронизации, он ищет ближайшего к нему и опережающего в репликации (таким образом, в конечном итоге у вас не получится цикл репликации: члены будут реплицировать только первичные или вторичные узлы, которые находятся впереди).

Это означает, что если вы вызовете нового члена во вторичном центре обработки данных, он с большей вероятностью выполнит синхронизацию другого члена в этом ЦОД, нежели члена в вашем первичном дата-центре (таким образом, минимизируя трафик WAN), как показано на рис. 13.1.

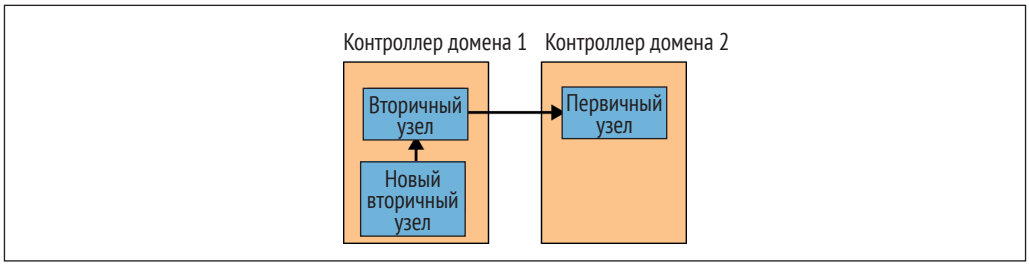

Рис. 13.1. Новые вторичные узлы обычно решают синхронизировать члена в том же центре обработки данных

Однако у автоматической цепочки репликации есть и обратная сторона: большее количество скачков репликации означает, что репликация операций записи на все серверы занимает немного больше времени. Например, допустим, что все находится в одном центре обработки данных, но из-за капризов сетевых скоростей, когда вы добавили членов, получается репликация в ряд, как показано на рис. 13.2.

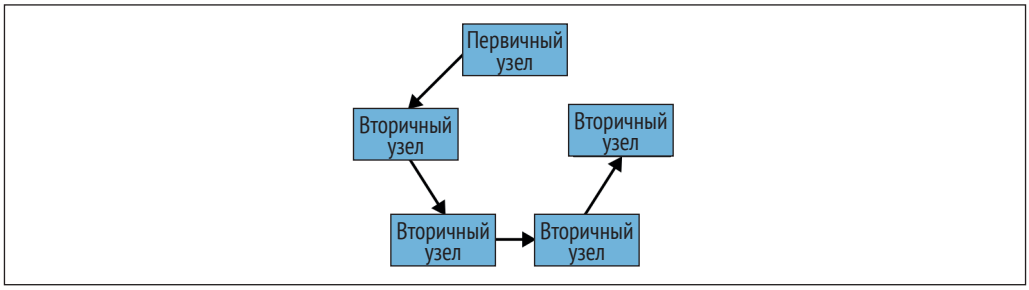

Рис. 13.2. Поскольку цепочки репликации становятся длиннее, всем членам требуется больше времени для получения копии данных

Это очень маловероятно, но не невозможно и, вероятно, нежелательно: каждый вторичный узел в цепочке должен быть немного дальше, чем вторичный узел «перед» ним. Это можно исправить, изменив источник репликации члена с помощью команды `replSetSyncFrom` (или функции `rs.syncFrom()`).

Подключитесь к вторичному узлу, источник репликации которого вы хотите изменить, и выполните эту команду, передав ей сервер, который, как вы хотите, этот элемент должен синхронизировать:

```
> secondary.adminCommand({"replSetSyncFrom" : "server0:27017"})
```

Переключение источников синхронизации может занять несколько секунд, но если вы снова выполните функцию `rs.status()` для этого члена, то должны увидеть, что в поле `"syncingTo"` теперь написано `"server0:27017"`.

Этот член (*server4*) будет продолжать репликацию *server0* до тех пор, пока тот не станет недоступным или, если он окажется вторичным, значительно отстанет от других членов.

Циклы репликации

Цикл репликации – это когда члены заканчивают репликацию друг друга, например *A* синхронизирует *B*, который синхронизирует *C*, синхронизирующий *A*. Поскольку ни один из членов в цикле репликации не может быть первичным, члены не будут получать никаких новых операций для репликации и будут отставать.

Циклы репликации должны быть невозможны, когда члены решают, кого синхронизировать автоматически. Однако вы можете форсиро-

вать циклы репликации, используя команду `replSetSyncFrom`. Внимательно изучите вывод функции `rs.status()`, прежде чем вручную изменять цели синхронизации, и будьте осторожны, чтобы не создать циклы. Команда `replSetSyncFrom` предупредит вас, если вы решите не синхронизовать члена, который находится строго впереди, но позволит сделать это.

Отключение цепочки

Цепочка – когда вторичный член синхронизирует другого вторичного члена (вместо первичного). Как упоминалось ранее, члены могут решить синхронизировать других членов автоматически. Цепочку можно отключить, заставляя всех синхронизировать первичного члена, изменив значение `"chainingAllowed"` на `false` (если оно не указано, по умолчанию это `true`):

```
> var config = rs.config()
> // создаем подобъект настроек, если он еще не существует t
> config.settings = config.settings || {}
> config.settings.chainingAllowed = false
> rs.reconfig(config)
```

Если для параметра `"chainingAllowed"` установлено значение `false`, все члены будут синхронизировать первичного члена. Если тот становится недоступным, они переключаются обратно на синхронизацию вторичных.

Расчет величины отставания

Одним из наиболее важных показателей репликации, которые необходимо отслеживать, является то, насколько хорошо вторичные узлы следуют за первичным. *Отставание* – это то, насколько далеко отстает вторичный узел, что означает разницу между отметкой времени последней операции, выполненной первичным узлом, и временной меткой последней операции, которую выполнил вторичный узел.

Можно использовать функцию `rs.status()`, чтобы увидеть состояние репликации члена, но вы также можете получить краткую сводку, выполнив функцию `rs.printReplicationInfo()` или `rs.printSlaveReplicationInfo()`.

Функция `rs.printReplicationInfo()` дает сводку журнала операций первичного узла, включая его размер и диапазон дат его операций:

```
> rs.printReplicationInfo();
    configured oplog size:    10.48576 MB
    log length start to end: 3590 secs(1.00 hrs)
    oplog first event time:   Tue Apr 10 2018 09: 27: 57 GMT - 0400(EDT)
    oplog last event time:    Tue Apr 10 2018 10: 27: 47 GMT - 0400(EDT)
    now:                      Tue Apr 10 2018 10: 27: 47 GMT - 0400(EDT)
```

В этом примере размер журнала операций составляет примерно 10 МБ (10 мегабайт) и может вместить только около часа операций.

Если бы это было реальное развертывание, он, вероятно, должен бы быть больше (*см.* следующий раздел, где приводятся инструкции по изменению размера журнала операций). Мы хотим, чтобы длина журнала была, *по крайней мере*, такой же, как и время, необходимое для полной повторной синхронизации. Таким образом, это не тот случай, когда вторичный узел доходит до конца журнала до завершения начальной синхронизации.

Длина лога вычисляется, если взять разницу во времени между первой и последней операциями в журнале операций после его заполнения. Если сервер только что запустился и в журнале операций ничего нет, самая ранняя операция будет относительно недавней. В этом случае длина лога будет небольшой, хотя, возможно, в журнале операций еще есть свободное место. Длина – более полезный показатель для серверов, которые работали достаточно долго, чтобы хотя бы раз выполнить операцию записи в свой журнал операций.

Вы также можете использовать функцию `rs.printSlaveReplicationInfo()`, чтобы получить значение `syncedTo` для каждого члена и время, когда была сделана последняя запись журнала операций для каждого вторичного узла, как показано в следующем примере:

```
> rs.printSlaveReplicationInfo();
source: m1.example.net: 27017
    syncedTo: Tue Apr 10 2018 10: 27: 47 GMT - 0400(EDT)
    0 secs(0 hrs) behind the primary
source: m2.example.net: 27017
    syncedTo: Tue Apr 10 2018 10: 27: 43 GMT - 0400(EDT)
    0 secs(0 hrs) behind the primary
source: m3.example.net: 27017
    syncedTo: Tue Apr 10 2018 10: 27: 39 GMT - 0400(EDT)
    0 secs(0 hrs) behind the primary
```

Помните, что задержка члена набора реплик рассчитывается относительно первичного узла, а не по «фактическому времени». Обычно это не имеет значения, но в системах с очень низким уровнем записи это может привести к фантомным «скачкам» отставания при репликации. Например, предположим, что вы выполняете операцию записи один раз в час. Сразу после этой операции, до того, как она будет реплицирована, вторичный

узел будет выглядеть так, как будто он на час позади первичного. Тем не менее он сможет синхронизироваться с этим «часом» операций за несколько миллисекунд. Иногда это может привести к путанице при мониторинге системы с низкой пропускной способностью.

Изменение размера журнала операций

Журнал операций вашего первичного узла следует рассматривать как окно обслуживания. Если у вашего первичного узла есть журнал операций длиной в час, то у вас есть только один час, чтобы исправить все, что пойдет не так, прежде чем ваши вторичные узлы слишком отстанут и их придется синхронизировать повторно с нуля. Таким образом, как правило, вам нужен журнал операций, который может содержать данные за период от пары дней до недели, чтобы у вас было какое-то пространство, если что-то пойдет не так.

К сожалению, нет простого способа определить, насколько длинен будет ваш журнал, прежде чем он заполнится. Подсистема хранения WiredTiger позволяет оперативно изменять размер вашего журнала операций в режиме онлайн во время работы вашего сервера. Сначала вы должны выполнить приведенные ниже шаги для каждого вторичного члена набора реплик; как только они будут изменены, тогда и только тогда вы должны вносить изменения в свой первичный узел. Помните, что каждый сервер, который мог бы стать первичным, должен иметь достаточно большой журнал операций, для того чтобы у вас было полноценное окно обслуживания.

Чтобы увеличить размер журнала операций, выполните следующие действия.

1. Подключитесь к члену набора реплик. Если аутентификация включена, обязательно используйте пользователя с привилегиями, который может изменять базу данных `local`.
2. Проверьте текущий размер журнала:

```
> use local
> db.oplog.rs.stats(1024*1024).maxSize
```

Будет показан размер коллекции в мегабайтах.

3. Измените размер журнала члена набора реплик:

```
> db.adminCommand({replSetResizeOplog: 1, size: 16000})
```

Следующая операция изменяет размер журнала операций члена набора реплик на 16 гигабайт, или 16 000 мегабайт.

4. Наконец, если вы уменьшили размер журнала, вам может потребоваться выполнить команду compact, чтобы освободить место на диске. Ее не стоит использовать с членом, пока он является первичным. Пожалуйста, обратитесь к документации MongoDB (https://oreil.ly/krv0R) для получения более подробной информации об этом случае и процедуре целиком.

Как правило, не следует уменьшать размер своего журнала операций: хотя он может длиться месяцами, для него обычно достаточно места на диске, и он не использует никаких ценных ресурсов, таких как оперативная память или ЦП.

Построение индексов

Если вы отправляете сборку индекса первичному узлу, он будет строить индекс обычным образом, а затем вторичные узлы будут строить индекс, когда будут реплицировать операцию «построение индекса». Хотя это самый простой способ создания индекса, построение индекса – это ресурсоемкие операции, которые могут сделать членов недоступными. Если все ваши вторичные узлы начнут создавать индекс одновременно, почти каждый член вашего набора будет отключен, пока индекс не будет построен. Данный процесс предназначен только для наборов реплик; если речь идет о кластере с разделением на разделы, пожалуйста, ознакомьтесь с документацией MongoDB, где говорится о построении индексов на кластере с разделением (https://oreil.ly/wJNeE).

Нужно остановить все записи в коллекцию при создании «уникального» индекса. Если запись не остановлена, у вас могут получиться несогласованные данные по всем членам набора реплик.

Поэтому вы можете захотеть создать индекс для одного члена за раз, чтобы минимизировать влияние на свое приложение. Для этого сделайте следующее:

1) выключите вторичный узел;
2) перезагрузите его как автономный сервер;

3) постройте индекс на автономном сервере;
4) после завершения построения индекса перезапустите сервер как член набора реплик. При перезапуске необходимо удалить параметр `disableLogicalSessionCacheRefresh`, если он присутствует в параметрах командной строки или в файле конфигурации;
5) повторите шаги с 1 по 4 для каждого вторичного узла в наборе реплик.

Теперь у вас должен быть набор, в котором каждый член, кроме первичного, имеет построенный индекс. Сейчас есть два варианта, и вы должны выбрать тот, который меньше всего повлияет на вашу производственную систему:

1) постройте индекс над первичным узлом. Если вы пребываете в выключенном состоянии, когда у вас меньше трафика, это, вероятно, самое подходящее время для его построения. Возможно, вы захотите изменить предпочтения чтения, чтобы временно переместить больше нагрузки на вторичные узлы во время сборки.

   Первичный узел будет реплицировать сборку индекса во вторичные узлы, но у них уже будет индекс, поэтому для них это будет пустой операцией;
2) понизьте в ранге первичный узел, затем выполните шаги с 2 по 4 описанной выше процедуры. Это требует отработки отказа, но у вас будет нормально работающий первичный узел, пока старый первичный узел строит свой индекс. После завершения построения индекса вы можете снова ввести его в набор.

Обратите внимание на то, что эту технику также можно использовать для создания над вторичным узлом других индексов, отличных от остального набора. Это может быть полезно для автономной обработки, но убедитесь, что член с разными индексами не сможет стать первичным: его приоритет всегда должен быть равен 0.

Если вы создаете уникальный индекс, убедитесь, что первичный узел не вставляет дубликаты или что сначала вы строите индекс над первичным узлом. В противном случае первичный узел может вставить дубликаты, что приведет к ошибкам репликации на вторичных узлах. Если это произойдет, вторичный узел отключится. Вам придется перезапустить его как автономный сервер. Удалите уникальный индекс и перезапустите его.

Бюджетная репликация

Если вы можете достать только один сервер высокого качества, рассмотрите возможность приобретения вторичного сервера, предназначенного исключительно для аварийного восстановления, с меньшим объемом ОЗУ

и ЦП, более медленным дисковым вводом-выводом и т. д. Хороший сервер всегда будет вашим первичным сервером, а более дешевый сервер никогда не будет обрабатывать клиентский трафик (настройте своих клиентов на отправку всех операций чтения на первичный сервер). Ниже перечислены параметры для более дешевого варианта:

`"priority" : 0`
: Вы не хотите, чтобы этот сервер когда-либо становился первичным.

`"hidden" : true`
: Вы не хотите, чтобы клиенты когда-либо отправляли операции чтения на этот вторичный сервер.

`"buildIndexes" : false`
: Необязательный параметр, но он может значительно снизить нагрузку, которую этот сервер должен обрабатывать. Если вам когда-нибудь понадобится восстановить этот сервер, вам нужно будет построить индексы заново.

`"votes" : 0`
: Если у вас только две машины, установите для `"votes"` на этом вторичном узле значение, равное 0, чтобы первичный узел мог оставаться первичным, если эта машина выйдет из строя. Если у вас есть третий сервер (даже просто сервер приложений), вместо этого лучше запустите на нем арбитра.

Это обеспечит безопасность наличия вторичного сервера без необходимости инвестировать в два высокопроизводительных сервера.

Часть IV

Шардинг

Глава 14

Знакомство с шардингом

В этой главе рассказывается, как выполнять масштабирование с помощью MongoDB. Мы рассмотрим:

- что такое шардинг и компоненты кластера;
- как настроить шардинг;
- основы взаимодействия шардинга с вашим приложением.

Что такое шардинг?

Шардинг обозначает процесс распределения данных по разным серверам; для описания этой концепции также иногда используется термин *разделение*. Размещая подмножество данных на каждой машине, становится возможным хранить больше данных и обрабатывать бóльшую нагрузку, без необходимости наличия больших или более мощных машин – достаточно просто бóльшего количества менее мощных компьютеров. Шардинг можно использовать и для других целей, включая размещение данных с более частым доступом для более производительного оборудования или разделение набора данных на основе географии, чтобы найти подмножество документов в коллекции (например, для пользователей, базирующихся в конкретной локали) рядом с серверами приложений, с которых к ним чаще всего обращаются.

Вручную шардинг можно осуществить практически с помощью любой СУБД. При таком подходе приложение поддерживает соединения с несколькими различными серверами баз данных, каждый из которых является полностью независимым. Приложение управляет хранением различных данных на разных серверах и выполняет запросы к соответствующему серверу, чтобы получить данные обратно. Такой вариант может хорошо работать, но его становится трудно поддерживать при добавлении или удалении узлов из кластера или при изменении распределения данных либо шаблонов нагрузки.

MongoDB поддерживает автоматический шардинг, который пытается абстрагировать архитектуру от приложения и упростить администрирование такой системы. MongoDB позволяет вашему приложению игнориро-

вать тот факт, что оно в некоторой степени не взаимодействует с автономным сервером MongoDB. Что касается операций, MongoDB автоматизирует балансировку данных между шардами и упрощает процесс добавления и удаления емкости.

Шардинг является наиболее сложным способом настройки MongoDB как с точки зрения разработки, так и с точки зрения эксплуатации. Тут множество компонентов, которые нужно настраивать и мониторить, и данные перемещаются по кластеру автоматически. Вы должны освоиться с автономными серверами и наборами реплик, прежде чем пытаться развернуть или использовать разделенный кластер. Кроме того, как и в случае с наборами реплик, рекомендуемые средства настройки и развертывания разделенных кластеров – это платформы Ops Manager или Atlas. Рекомендуется использовать Ops Manager, если вам нужно поддерживать управление своей вычислительной инфраструктурой. MongoDB Atlas рекомендуется, если вы можете предоставить управление инфраструктурой MongoDB (у вас есть возможность работать в Amazon AWS, Microsoft Azure или Google Compute Cloud).

Разбираемся с компонентами кластера

Шардинг позволяет вам создавать кластер из множества серверов (шардов) и распределять коллекцию по ним, поместив подмножество данных в каждый шард. Это дает возможность вашему приложению расти за пределами ресурсов автономного сервера или набора реплик.

Многие не понимают разницы между репликацией и шардингом. Помните, что в ходе репликации создается точная копия ваших данных на нескольких серверах, поэтому каждый сервер – это зеркальное отражение любого другого сервера. И наоборот, каждый шард содержит различное подмножество данных.

Одна из целей шардинга – сделать так, чтобы для вашего приложения кластер из 2, 3, 10 или даже сотен шардов выглядел как одна машина. Чтобы скрыть эти детали от приложения, мы запускаем перед шардами один или несколько процессов маршрутизации под названием *mongos*. Процессы *mongos* хранят «оглавление таблиц», которое сообщает им, в каком шарде какие данные содержатся. Приложения могут подключаться к этому маршрутизатору и отправлять запросы в обычном режиме, как показано на рис. 14.1. Маршрутизатор, зная, какие данные на каком шарде находятся, может пересылать запросы соответствующему шарду (шардам). Если есть ответы на запрос, маршрутизатор собирает их и, если необходимо,

объединяет и отправляет обратно в приложение. Насколько известно приложению, оно подключено к автономному серверу *mongod*, как показано на рис. 14.2.

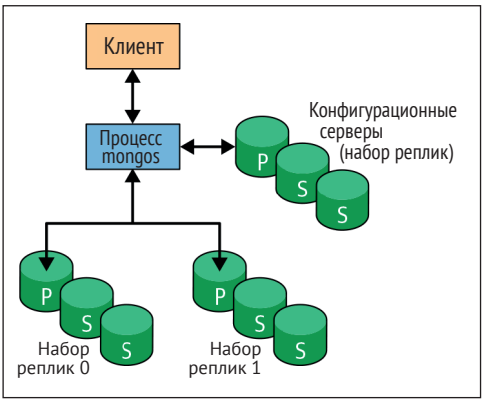

Рис. 14.1. Соединение с разделенным клиентом

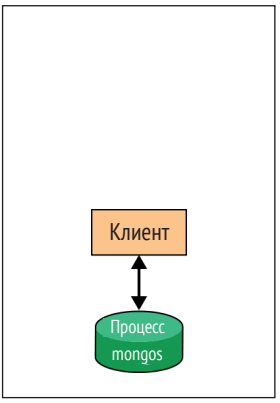

Рис. 14.2. Соединение с обычным клиентом

Настройка кластера на одной машине

Начнем с настройки быстрого кластера на одной машине. Сначала запустите оболочку *mongo* с опциями `--nodb` и `--norc`:

```
$ mongo --nodb --norc
```

Чтобы создать кластер, используйте класс `ShardingTest`. Выполните приведенный ниже код в оболочке *mongo*, которую вы только что запустили:

```
st = ShardingTest({
    name: "one-min-shards",
    chunkSize: 1,
    shards: 2,
    rs: {
        nodes: 3,
        oplogSize: 10
    },
    other: {
        enableBalancer: true
    }
});
```

Параметр `chunksize` описан в главе 17. А пока просто установите для него значение 1. Что касается других параметров, переданных здесь `ShardingTest`, `name` просто предоставляет метку для нашего разделенного кластера, `shards` указывает, что наш кластер будет состоять из двух шардов (мы делаем это,

чтобы поддерживать использование ресурсов на низком уровне для этого примера), а `rs` определяет каждый шард как набор реплик из трех узлов с размером журнала операций (`oplogSize`) 10 Мебибайт (опять же, чтобы сохранить использование ресурсов на низком уровне), Хотя можно запустить по одному автономному серверу *mongod* для каждого шарда, это даст более четкую картину типичной архитектуры разделенного кластера, если мы создадим каждый шард в виде набора реплик. В последнем указанном параметре мы даем указание `ShardingTest` включить балансировщик, как только кластер запустится, что обеспечит равномерное распределение данных по обоим шардам.

`ShardingTest` – это класс, разработанный для внутреннего использования отделом разработки MongoDB. Однако, поскольку он поставляется с сервером MongoDB, он предоставляет наиболее простые средства для экспериментов с разделенным кластером. Изначально `ShardingTest` был разработан для поддержки тестовых наборов серверов и по-прежнему используется для этой цели. По умолчанию он предоставляет ряд удобств, которые помогают поддерживать использование ресурсов на минимально возможном уровне и настраивать относительно сложную архитектуру разделенного кластера. Он предполагает, что на вашем компьютере есть каталог `/data/db`; если `ShardingTest` не запускается, создайте этот каталог и повторите команду.

Когда вы выполните эту команду, `ShardingTest` многое будет делать за вас автоматически. Он создаст новый кластер с двумя шардами, каждый из которых представляет собой набор реплик. Он настроит наборы реплик и запустит каждый узел с необходимыми параметрами для установки протоколов репликации. Запустит процессы *mongos* для управления запросами к шардам, чтобы клиенты могли взаимодействовать с кластером, как если бы они обменивались данными с автономным сервером *mongod*, в некоторой степени. Наконец, он запустит дополнительный набор реплик для серверов конфигурации, которые хранят информацию о таблице маршрутизации, необходимую для обеспечения направления запросов к правильному шарду. Помните, что основные сценарии использования для шардинга – это разделение набора данных для устранения аппаратных и стоимостных ограничений или для обеспечения более высокой производительности приложений (например, географическое разделение). В MongoDB шардинг предоставляет эти возможности способом, который во многих отношениях является безболезненным для приложения.

Как только `ShardingTest` завершит настройку кластера, у вас будет 10 запущенных процессов, к которым вы можете подключиться: два набора реплик по три узла в каждом, один набор реплик сервера конфигурации из трех узлов и один процесс *mongos*. По умолчанию эти процессы должны начинаться на порту 20 000. *mongos* должен работать на порту 20 009. Другие процессы, которые вы выполняете на локальном компьютере, и пре-

дыдущие вызовы `ShardingTest` могут влиять на то, какие порты использует `ShardingTest`, но у вас не должно возникнуть особых трудностей при определении портов, на которых работают процессы вашего кластера.

Затем вы подключитесь к процессу *mongos*, чтобы поэкспериментировать с кластером. Весь ваш кластер будет выгружать свои журналы в вашу текущую оболочку, поэтому откройте второе окно терминала и запустите еще одну оболочку *mongo*:

```
$ mongo -nodb
```

Используйте эту оболочку для подключения к процессу *mongos* своего кластера. Опять же, этот процесс должен работать на порту 20009:

```
> db = (new Mongo("localhost:20009")).getDB("accounts")
```

Обратите внимание, что приглашение оболочки *mongo* должно измениться, чтобы показать, что вы подключены к процессу *mongos*. Теперь вы находитесь в ситуации, показанной ранее, на рис. 14.1: оболочка является клиентом и подключена к процессу *mongos*. Вы можете начать передавать запросы *mongos*, а он направит их в шарды. Вам не нужно ничего знать о шардах, например сколько их или каков их адрес. Пока шарды существуют, вы можете передавать запросы *mongos* и позволять ему пересылать их соответствующим образом.

Начнем со вставки данных:

```
> for (var i=0; i<100000; i++) {
...       db.users.insert({"username" : "user"+i, "created_at" : new Date()});
... }
> db.users.count()
100000
```

Как видите, взаимодействие с *mongos* работает так же, как взаимодействие с автономным сервером.

Вы можете получить общее представление о вашем кластере, выполнив функцию `sh.status()`. Это даст вам краткий обзор ваших шардов, баз данных и коллекций:

```
> sh.status()
    --- Sharding Status ---
    sharding version: {
        "_id": 1,
        "minCompatibleVersion": 5,
        "currentVersion": 6,
        "clusterId": ObjectId("5a4f93d6bcde690005986071")
}
shards: {
```

```
    "_id": "one-min-shards-rs0",
    "host": "one-min-shards-rs0/MBP:20000,MBP:20001,MBP:20002",
    "state": 1
}
{
    "_id": "one-min-shards-rs1",
    "host": "one-min-shards-rs1/MBP:20003,MBP:20004,MBP:20005",
    "state": 1
}
active mongoses:
    "3.6.1": 1
autosplit:
    Currently enabled: no
balancer:
    Currently enabled: no
    Currently running: no
    Failed balancer rounds in last 5 attempts: 0
    Migration Results for the last 24 hours:
        No recent migrations
databases: {
        "_id": "accounts",
        "primary": "one-min-shards-rs1",
        "partitioned": false
    }
    {
        "_id": "config",
        "primary": "config",
        "partitioned": true
    }
    config.system.sessions
shard key: { "_id": 1 }
unique: false
balancing: true
chunks:
    one - min - shards - rs0 1
    { "_id": { "$minKey": 1 } } -->> { "_id": { "$maxKey": 1 }}
    on: one - min - shards - rs0 Timestamp(1, 0)
```

`sh` напоминает `rs`, но предназначена для шардинга: это глобальная переменная, определяющая количество вспомогательных функций шардинга, которые можно увидеть, выполнив функцию `sh.help()`. Как видно из вывода функции `sh.status()`, у вас есть два шарда и две базы данных (конфигурация создана автоматически).

У вашей базы данных *accounts* может быть другой первичный шард, не тот, что показан здесь. Первичный шард – своего рода «основное место дислокации», которое выбирается случайным образом для каждой базы данных. Все ваши данные будут находиться на этом первичном шарде. MongoDB пока еще не может автоматически распределять ваши данные, потому что не знает, как вы хотите, чтобы они были распределены (или хотите ли вообще). Вы должны указать для каждой коллекции, как вы хотите, чтобы она распределяла данные.

Первичный шард отличается от первичного узла набора реплик. Первичный шард относится ко всему набору реплик, образующих шард. Первичным узлом в наборе реплик является единственный сервер в наборе, который может принимать операции записи.

Для разделения определенной коллекции сначала активируйте шардинг в базе данных коллекции. Для этого выполните команду `enableSharding`:

```
> sh.enableSharding("accounts")
```

Теперь в базе данных *accounts* разрешено использование шардинга, что позволяет вам разбивать коллекции в рамках этой базы данных.

Когда вы разбиваете коллекцию, вы выбираете специальный ключ шардинга. Это одно или два поля, которые MongoDB использует для разбивки данных. Например, если вы выбрали `"username"`, MongoDB будет разбивать данные на диапазоны имен пользователей: от `"a1-steak-sauce"` до `"defcon"`, от `"defcon1"` до `"howie1998"` и т. д. Выбор ключа можно рассматривать как упорядочение данных в коллекции. Это похоже на индексирование, и на то есть веская причина: этот ключ становится самым важным индексом в вашей коллекции по мере его увеличения. Чтобы даже создать такой ключ, поле (поля) должно быть проиндексировано.

Итак, перед активацией шардинга необходимо создать индекс для ключа, по которому вы хотите делать разбивку:

```
> db.users.createIndex({"username" : 1})
```

Теперь можно разбить коллекцию по `"username"`:

```
> sh.shardCollection("accounts.users", {"username" : 1})
```

Хотя мы выбираем ключ, не задумываясь, это важное решение, которое должно быть тщательно обдумано, когда речь идет о реальной системе. *См.* главу 16, где приводятся дополнительные советы по выбору ключа шардинга.

Если вы подождете несколько минут и снова выполните функцию `sh.status()`, то увидите, что теперь отображается намного больше информации по сравнению с тем, что было раньше:

```
> sh.status()
--- Sharding Status ---
sharding version: {
    "_id": 1,
    "minCompatibleVersion": 5,
    "currentVersion": 6,
    "clusterId": ObjectId("5a4f93d6bcde690005986071")
    }
shards:
    {
        "_id": "one-min-shards-rs0",
        "host": "one-min-shards-rs0/MBP:20000,MBP:20001,MBP:20002",
        "state": 1
    }
    {
        "_id": "one-min-shards-rs1",
        "host": "one-min-shards-rs1/MBP:20003,MBP:20004,MBP:20005",
        "state": 1
    }
active mongoses:
    "3.6.1": 1
autosplit:
    Currently enabled: no
balancer:
    Currently enabled: yes
    Currently running: no
    Failed balancer rounds in last 5 attempts: 0
    Migration Results for the last 24 hours:
        6: Success
databases:
    {
        "_id": "accounts",
        "primary": "one-min-shards-rs1",
        "partitioned": true
    }
accounts.users
shard key: { "username": 1 }
unique: false
balancing: true
chunks:
    one - min - shards - rs0 6
```

```
        one - min - shards - rs1 7
        { "username": { "$minKey": 1 } } -->>
          { "username": "user17256" } on: one - min - shards - rs0 Timestamp(2, 0)
        { "username": "user17256" } -->>
          { "username": "user24515" } on: one - min - shards - rs0 Timestamp(3, 0)
        { "username": "user24515" } -->>
          { "username": "user31775" } on: one - min - shards - rs0 Timestamp(4, 0)
        { "username": "user31775" } -->>
          { "username": "user39034" } on: one - min - shards - rs0 Timestamp(5, 0)
        { "username": "user39034" } -->>
          { "username": "user46294" } on: one - min - shards - rs0 Timestamp(6, 0)
        { "username": "user46294" } -->>
          { "username": "user53553" } on: one - min - shards - rs0 Timestamp(7, 0)
        { "username": "user53553" } -->>
          { "username": "user60812" } on: one - min - shards - rs1 Timestamp(7, 1)
        { "username": "user60812" } -->>
          { "username": "user68072" } on: one - min - shards - rs1 Timestamp(1, 7)
        { "username": "user68072" } -->>
          { "username": "user75331" } on: one - min - shards - rs1 Timestamp(1, 8)
        { "username": "user75331" } -->>
          { "username": "user82591" } on: one - min - shards - rs1 Timestamp(1, 9)
        { "username": "user82591" } -->>
          { "username": "user89851" } on: one - min - shards - rs1 Timestamp(1, 10)
        { "username": "user89851" } -->>
          { "username": "user9711" } on: one - min - shards - rs1 Timestamp(1, 11)
        { "username": "user9711" } -->>
          { "username": { "$maxKey": 1 } } on: one - min - shards - rs1 Timestamp(1, 12)
        { "_id": "config", "primary": "config", "partitioned": true }
    config.system.sessions
        shard key: { "_id": 1 }
        unique: false
        balancing: true
        chunks:
            one - min - shards - rs0 1
            { "_id": { "$minKey": 1 } } -->>
            { "_id": { "$maxKey": 1 } } on: one - min - shards - rs0 Timestamp(1, 0)
```

Коллекция была разделена на 13 чанков, каждый из которых представляет собой подмножество исходных данных. Они перечислены по диапазону ключа (`{"username" : minValue} -->> {"username" : maxValue}` обозначает диапазон каждого чанка). Глядя на ту часть вывода, где написано `"on" : шард`, видно, что эти чанки были равномерно распределены между шардами.

Этот процесс разбиения коллекции на чанки показан графически на рис. 14.3–14.5. До шардинга коллекция представляет собой единый чанк.

В результате шардинга она разбивается на более мелкие чанки на основе ключа, как показано на рис. 14.4. Эти чанки могут быть распределены по кластеру, как показано на рис. 14.5.

Рис. 14.3. Перед тем как коллекция будет разделена, ее можно рассматривать как единый чанк от наименьшего значения ключа до самого большого

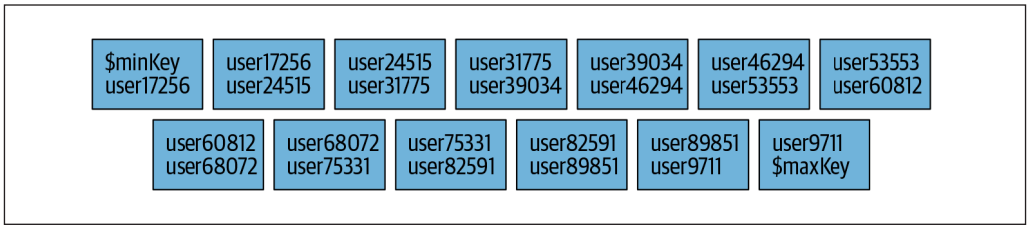

Рис. 14.4. В результате шардинга коллекция разбивается на множество чанков на базе диапазонов ключей

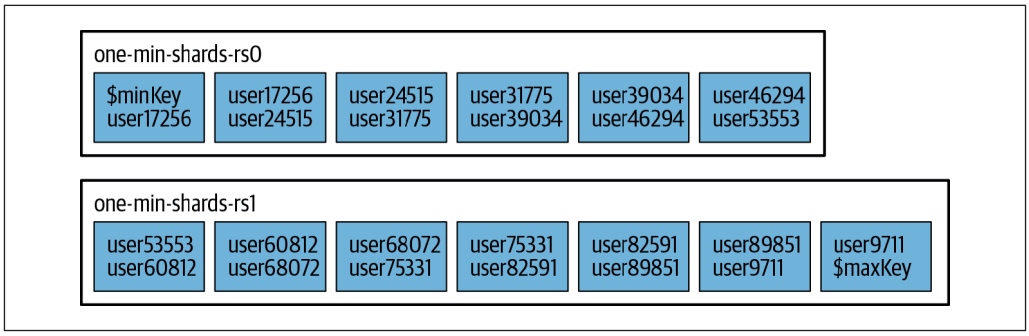

Рис. 14.5. Чанки равномерно распределяются по доступным шардам

Обратите внимание на ключи, расположенные в начале и конце списка чанков: `$minKey` и `$maxKey`. `$minKey` можно рассматривать как «отрицательную бесконечность». Он меньше, чем любое другое значение в MongoDB. Аналогично, `$maxKey` – своего рода «положительная бесконечность». Он больше, чем любое другое значение. Таким образом, вы всегда будете видеть их как «заглавные буквы» на диапазонах своих чанков. Значения вашего ключа всегда будут находиться между `$minKey` и `$maxKey`. На самом деле эти значения – типы BSON, и их не нужно использовать в своем приложении; они предназначены в основном для внутреннего применения. Если вы хотите обращаться к ним в оболочке, используйте константы `MinKey` и `MaxKey`.

Теперь, когда данные распределены по нескольким шардам, попробуем выполнить несколько запросов. Для начала попробуем запросить конкретное имя пользователя:

```
> db.users.find({ username: "user12345" })
{
    "_id": ObjectId("5a4fb11dbb9ce6070f377880"),
    "username": "user12345",
    "created_at": ISODate("2018-01-05T17:08:45.657Z")
}
```

Как видите, выполнение запросов работает нормально. Но давайте выполним команду ~~explain~~, чтобы увидеть, что MongoDB делает за кулисами:

```
> db.users.find({ username: "user12345" }}).explain()
{
    "queryPlanner": {
        "mongosPlannerVersion": 1,
        "winningPlan": {
            "stage": "SINGLE_SHARD",
            "shards": [{
                "shardName": "one-min-shards-rs0",
                "connectionString":
                     "one-min-shards-rs0/MBP:20000,MBP:20001,MBP:20002",
                "serverInfo": {
                    "host": "MBP",
                    "port": 20000,
                    "version": "3.6.1",
                    "gitVersion": "025d4f4fe61efd1fb6f0005be20cb45a004093d1"
                },
                "plannerVersion": 1,
                "namespace": "accounts.users",
                "indexFilterSet": false,
                "parsedQuery": {
                    "username": {
                        "$eq": "user12345"
                    }
                },
                "winningPlan": {
                    "stage": "FETCH",
                    "inputStage": {
                        "stage": "SHARDING_FILTER",
                        "inputStage": {
                            "stage": "IXSCAN",
```

```
                                "keyPattern": {
                                    "username": 1
                                },
                                "indexName": "username_1",
                                "isMultiKey": false,
                                "multiKeyPaths": {
                                    "username": []
                                },
                                "isUnique": false,
                                "isSparse": false,
                                "isPartial": false,
                                "indexVersion": 2,
                                "direction": "forward",
                                "indexBounds": {
                                    "username": [
                                        "[\"user12345\", \"user12345\"]"
                                    ]
                                }
                            }
                        }
                    },
                    "rejectedPlans": []
                }]
            }
        },
        "ok": 1,
        "$clusterTime": {
            "clusterTime": Timestamp(1515174248, 1),
            "signature": {
                "hash": BinData(0, "AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                "keyId": NumberLong(0)
            }
        },
        "operationTime": Timestamp(1515173700, 201)
    }
```

Глядя на поле `"winningPlan"` в выводе команды, видим, что наш кластер удовлетворил этот запрос, используя один шард, *one-min-shards-rs0*. Основываясь на выводе функции `sh.status()`, приведенном ранее, мы видим, что *user12345* попадает в диапазон ключей первого чанка, указанного для этого шарда в нашем кластере.

Поскольку `"username"` – это ключ, процесс *mongos* смог перенаправить запрос непосредственно на правильный шард. Сравните это с результатами выполнения запроса всех пользователей:

```
> db.users.find().explain()
{
    "queryPlanner":{
        "mongosPlannerVersion":1,
        "winningPlan":{
            "stage":"SHARD_MERGE",
            "shards":[
                {
                  "shardName":"one-min-shards-rs0",
                  "connectionString":
                      "one-min-shards-rs0/MBP:20000,MBP:20001,MBP:20002",
                  "serverInfo":{
                      "host":"MBP.fios-router.home",
                      "port":20000,
                      "version":"3.6.1",
                       "gitVersion":"025d4f4fe61efd1fb6f0005be20cb45a004093d1"
                  },
                  "plannerVersion":1,
                  "namespace":"accounts.users",
                  "indexFilterSet":false,
                  "parsedQuery":{
                  },
                  "winningPlan":{
                  "stage":"SHARDING_FILTER",
                  "inputStage":{
                     "stage":"COLLSCAN",
                     "direction":"forward"
                   }
                },
                "rejectedPlans":[
                ]
              },
              {
                  "shardName":"one-min-shards-rs1",
                  "connectionString":
                     "one-min-shards-rs1/MBP:20003,MBP:20004,MBP:20005",
                  "serverInfo":{
                     "host":"MBP.fios-router.home",
                     "port":20003,
                     "version":"3.6.1",
                     "gitVersion":"025d4f4fe61efd1fb6f0005be20cb45a004093d1"
                  },
                  "plannerVersion":1,
                  "namespace":"accounts.users",
                  "indexFilterSet":false,
```

```
                    "parsedQuery":{
                    },
                    "winningPlan":{
                    "stage":"SHARDING_FILTER",
                    "inputStage":{
                    "stage":"COLLSCAN",
                    "direction":"forward"
                     }
                  },
                  "rejectedPlans":[
                  ]
                }
              ]
           }
        },
        "ok":1,
        "$clusterTime":{
            "clusterTime":Timestamp(1515174893, 1),
            "signature":{
                "hash":BinData(0, "AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                "keyId":NumberLong(0)
        }
    },
    "operationTime":Timestamp(1515173709, 514)
}
```

Как видно из этого вывода, запрос должен посетить оба шарда, чтобы найти все данные. В общем, если мы не используем ключ шардинга в запросе, *mongos* должен будет отправить запрос каждому шарду.

Запросы, которые содержат ключ шардинга и могут быть отправлены в один шард или подмножество шардов, называются *направленными запросами*. Запросы, которые должны быть отправлены во все шарды, называются *запросами с разделением и объединением* (широковещательные запросы): *mongos* разбрасывает запрос по всем шардам, а затем собирает результаты.

Как только вы закончите экспериментировать, выключите набор. Вернитесь к исходной оболочке и нажмите клавишу **Enter** несколько раз, чтобы вернуться в командную строку, затем выполните функцию `st.stop()`, чтобы аккуратно завершить работу всех серверов:

```
> st.stop()
```

Если вы не уверены в том, что будет делать операция, может быть полезно использовать класс `ShardingTest`, чтобы запустить локальный кластер и опробовать его.

Глава 15

Конфигурирование шардинга

В предыдущей главе мы настраивали «кластер» на одном компьютере. В этой главе рассказывается, как настроить более реалистичный кластер и как подходит каждый элемент. В частности, вы узнаете:

- как настроить серверы конфигурации, шарды и процессы *mongos;*
- как добавить емкости кластеру;
- как данные хранятся и распределяются.

Когда использовать шардинг

Решение относительно того, когда использовать шардинг, – это компромисс. Как правило, вам не нужно использовать его слишком рано, потому что это увеличивает операционную сложность вашего развертывания и вынуждает вас принимать проектные решения, которые потом сложно изменить. С другой стороны, вы не хотите ждать слишком долго, потому что разбивать перегруженную систему без простоя сложно.

В целом шардинг используется для того, чтобы:

- увеличить доступную оперативную память;
- увеличить доступное дисковое пространство;
- уменьшить нагрузку на сервер;
- читать или записывать данные с большей пропускной способностью, чем та, что может выдержать один сервер *mongod.*

Таким образом, важен хороший мониторинг, чтобы решить, когда шардинг будет необходим. Тщательно оценивайте каждый из этих показателей. Обычно одни сталкиваются с одной из этих задач гораздо быстрее, чем другие, поэтому определите, для какой из них необходимо подготовить развертывание в первую очередь, и заранее составьте планы относительно того, когда и как вы планируете преобразовать свой набор реплик.

Запуск серверов

Первым шагом в создании кластера является запуск всех необходимых процессов. Как упоминалось в предыдущей главе, нужно настроить *mongos* и шарды. Существует также третий компонент, конфигурационные серверы, которые являются важной частью. Конфигурационные серверы – это обычные серверы *mongod*, на которых хранится конфигурация кластера: в каких наборах реплик находятся шарды, какие коллекции разбиты, на каком шарде находится каждый чанк. В MongoDB версии 3.2 появилась возможность использовать наборы реплик в качестве конфигурационных серверов. Наборы реплик заменяют оригинальный механизм синхронизации, используемый конфигурационными серверами; способность использовать этот механизм была удалена в MongoDB версии 3.4.

Конфигурационные серверы

Конфигурационные серверы – мозг вашего кластера: в них находятся все метаданные о том, какие серверы какие данные содержат. Таким образом, их нужно настроить в первую очередь, и данные, которые они хранят, *чрезвычайно* важны: убедитесь, что они работают с включенным журналированием и что диски, на которых хранятся их данные, не виртуальные. При рабочем развертывании набор реплик вашего конфигурационного сервера должен состоять как минимум из трех членов. Каждый сервер должен располагаться на отдельной физической машине, предпочтительно географически распределенной.

Конфигурационные серверы должны запускаться до того, как будут запущены процессы *mongos*, поскольку *mongos* извлекают из них свою конфигурацию. Для начала выполните следующие команды на трех отдельных машинах, чтобы запустить конфигурационные серверы:

```
$ mongod --configsvr --replSet configRS --bind_ip localhost,198.51.100.51 mongod
  --dbpath /var/lib/mongodb

$ mongod --configsvr --replSet configRS --bind_ip localhost,198.51.100.52 mongod
  --dbpath /var/lib/mongodb

$ mongod --configsvr --replSet configRS --bind_ip localhost,198.51.100.53 mongod
  --dbpath /var/lib/mongodb
```

Затем инициируйте серверы как набор реплик. Для этого подключите оболочку *mongo* к одному из членов набора реплик:

```
$ mongo --host <hostname> --port <порт>
```

и используйте вспомогательную функцию `rs.initiate()`:

```
> rs.initiate(
{
    _id: "configRS",
    configsvr: true,
    members: [
        { _id: 0, host: "cfg1.example.net:27019" },
        { _id: 1, host: "cfg2.example.net:27019" },
        { _id: 2, host: "cfg3.example.net:27019" }
    ]
})
```

Здесь мы применяем `configRS` в качестве имени набора реплик. Обратите внимание, что это имя появляется как в командной строке при создании экземпляра сервера, так и при вызове функции `rs.initiate()`.

Опция `--configsvr` указывает *mongod*, что вы планируете использовать его в качестве конфигурационного сервера. На сервере, работающем с этой опцией, клиенты (то есть другие компоненты кластера) не могут записывать данные ни в какую другую базу данных, кроме *config* или *admin*.

База данных *admin* содержит коллекции, относящиеся к аутентификации и авторизации, а также другие коллекции *system.** для внутреннего использования. База данных *config* содержит коллекции, которые включают метаданные разделенного кластера. MongoDB записывает данные в базу данных *config* при изменении метаданных, например после миграции чанка или его разбиения.

При записи на конфигурационные серверы MongoDB использует уровень большинства `writeConcern`. Аналогично, при выполнении операций чтения MongoDB использует уровень `readConcern`. Это гарантирует, что метаданные разделенного кластера не будут зафиксированы в наборе реплик конфигурационного сервера до тех пор, пока их нельзя будет откатить. Это также гарантирует, что будут прочитаны только те метаданные, которые переживут сбой конфигурационных серверов. Это необходимо для обеспечения того, чтобы все маршрутизаторы *mongos* имели согласованное представление о том, как организованы данные в разделенном кластере.

Если все ваши конфигурационные серверы потеряны, вы должны просмотреть данные на своих шардах, чтобы выяснить, где какие данные находятся. Это можно сделать, но медленный и неприятный процесс. Делайте частые резервные копии данных конфигурационных серверов. Всегда делайте резервную копию своих серверов перед выполнением обслуживания кластера.

Конфигурационные серверы должны быть адекватно выделенными с точки зрения сетевых ресурсов и ресурсов ЦП. Они содержат только оглавление данных в кластере, поэтому требуемые ресурсы хранения минимальны. Они должны быть развернуты на отдельном оборудовании, чтобы избежать конкуренции за ресурсы машины.

Процессы mongos

После запуска трех конфигурационных серверов запустите процесс *mongos* для подключения к нему своего приложения. Процессам *mongos* необходимо знать, где находятся эти серверы, поэтому нужно всегда запускать их, используя параметр `--configdb`:

```
$ mongos --configdb \
  configRS/cfg1.example.net:27019, \
    cfg2.example.net:27019,cfg3.example.net:27019 \
  --bind_ip localhost,198.51.100.100 --logpath /var/log/mongos.log
```

По умолчанию процесс *mongos* работает на порту 27017. Обратите внимание, что ему не нужен каталог данных (*mongos* не хранит сами данные; при запуске он загружает конфигурацию кластера с конфигурационных серверов). Убедитесь, что вы установили опцию `--logpath`, чтобы сохранить журнал *mongos* где-нибудь в безопасном месте.

Нужно запустить небольшое количество процессов *mongos* и расположить их как можно ближе ко всем шардам. Это повышает производительность запросов, которым необходим доступ к нескольким шардам или которые выполняют операции с разделением и объединением. Для минимальной настройки нужно по крайней мере два процесса *mongos*, чтобы обеспечить высокую доступность. Можно запустить десятки или сотни процессов *mongos*, но это приведет к конфликту ресурсов на *конфигурационных серверах*. Рекомендуемый подход заключается в предоставлении небольшого пула маршрутизаторов.

Добавление шарда из набора реплик

Наконец, вы готовы добавить шард. Есть две возможности: у вас может быть существующий набор реплик или, возможно, вы начинаете с нуля. Мы начнем с существующего набора. Если вы начинаете с нуля, инициализируйте пустой набор и выполните описанные здесь действия.

Если у вас уже есть набор реплик, обслуживающий ваше приложение, он станет вашим первым шардом. Чтобы преобразовать его в шард, необходимо внести небольшие изменения в конфигурацию членов, а затем сообщить *mongos*, как найти набор реплик, который будет включать шард.

Например, если у вас есть набор реплик с именем *rs0* на *svr1.example.net*, *svr2.example.net* и *svr3.example.net*, сначала подключитесь к одному из членов, используя оболочку *mongo*:

```
$ mongo srv1.example.net
```

Затем используйте функцию `rs.status()`, чтобы определить, какой член является первичным, а какие – вторичными:

```
> rs.status()
"set": "rs0",
"date": ISODate("2018-11-02T20:02:16.543Z"),
    "myState": 1,
    "term": NumberLong(1),
    "heartbeatIntervalMillis": NumberLong(2000),
    "optimes": {
        "lastCommittedOpTime": {
            "ts": Timestamp(1478116934, 1),
            "t": NumberLong(1)
        },
        "readConcernMajorityOpTime": {
            "ts": Timestamp(1478116934, 1),
            "t": NumberLong(1)
        },
        "appliedOpTime": {
            "ts": Timestamp(1478116934, 1),
            "t": NumberLong(1)
        },
        "durableOpTime": {
            "ts": Timestamp(1478116934, 1),
            "t": NumberLong(1)
        }
    },
    "members": [{
            "_id": 0,
            "name": "svr1.example.net:27017",
            "health": 1,
            "state": 1,
            "stateStr": "PRIMARY",
            "uptime": 269,
            "optime": {
                "ts": Timestamp(1478116934, 1),
                "t": NumberLong(1)
            },
            "optimeDate": ISODate("2018-11-02T20:02:14Z"),
```

```
        "infoMessage": "could not find member to sync from",
        "electionTime": Timestamp(1478116933, 1),
        "electionDate": ISODate("2018-11-02T20:02:13Z"),
        "configVersion": 1,
        "self": true
    },
    {
        "_id": 1,
        "name": "svr2.example.net:27017",
        "health": 1,
        "state": 2,
        "stateStr": "SECONDARY",
        "uptime": 14,
        "optime": {
            "ts": Timestamp(1478116934, 1),
            "t": NumberLong(1)
        },
        "optimeDurable": {
            "ts": Timestamp(1478116934, 1),
            "t": NumberLong(1)
        },
        "optimeDate": ISODate("2018-11-02T20:02:14Z"),
        "optimeDurableDate": ISODate("2018-11-02T20:02:14Z"),
        "lastHeartbeat": ISODate("2018-11-02T20:02:15.618Z"),
        "lastHeartbeatRecv": ISODate("2018-11-02T20:02:14.866Z"),
        "pingMs": NumberLong(0),
        "syncingTo": "m1.example.net:27017",
        "configVersion": 1
    },
    {
        "_id": 2,
        "name": "svr3.example.net:27017",
        "health": 1,
        "state": 2,
        "stateStr": "SECONDARY",
        "uptime": 14,
        "optime": {
            "ts": Timestamp(1478116934, 1),
            "t": NumberLong(1)
        },
        "optimeDurable": {
            "ts": Timestamp(1478116934, 1),
            "t": NumberLong(1)
        },
        "optimeDate": ISODate("2018-11-02T20:02:14Z"),
```

```
            "optimeDurableDate": ISODate("2018-11-02T20:02:14Z"),
            "lastHeartbeat": ISODate("2018-11-02T20:02:15.619Z"),
            "lastHeartbeatRecv": ISODate("2018-11-02T20:02:14.787Z"),
            "pingMs": NumberLong(0),
            "syncingTo": "m1.example.net:27017",
            "configVersion": 1
        }
    ],
    "ok": 1
}
```

Начиная с MongoDB версии 3.4, когда речь идет о разделенных кластерах, экземпляры сервера *mongod* для шардов должны конфигурироваться с использованием параметра `--shardsvr`, либо через параметр конфигурационного файла `sharding.clusterRole`, либо через параметр командной строки `--shardsvr`.

Вам нужно будет сделать это для каждого члена набора реплик, который вы конвертируете в шард. Для этого сначала нужно перезапустить каждый вторичный узел по очереди, используя параметр `--shardsvr`, затем отключить первичный узел и перезапустить его, используя параметр `--shardsvr`.

После выключения вторичного узла перезапустите его:

```
$ mongod --replSet "rs0" --shardsvr --port 27017
    --bind_ip localhost,<ip-адрес члена>
```

Обратите внимание на то, что вам понадобится применять правильный IP-адрес для каждого вторичного узла для параметра `--bind_ip`.

Теперь подключите оболочку *mongo* к первичному узлу:

```
$ mongo m1.example.net
```

и воспользуйтесь функцией `rs.stepDown()`:

```
> rs.stepDown ()
```

Затем перезапустите прежний первичный узел, используя параметр `--shardsvr`:

```
$ mongod --replSet "rs0" --shardsvr --port 27017
    --bind_ip localhost,<IP-адрес бывшего первичного узла>
```

Теперь вы готовы добавить свой набор реплик в качестве шарда. Подключите оболочку *mongo* к базе данных *admin* процесса *mongos*:

```
$ mongo mongos1.example.net:27017/admin
```

И добавьте шард в кластер, используя метод `sh.addShard()`:

```
> sh.addShard(
    "rs0 / svr1.example.net: 27017, svr2.example.net: 27017, svr3.example.
net: 27017")
```

Вы можете указать всех членов набора, но вам это не нужно. *mongos* автоматически обнаружит всех членов, которые не были включены в список сидов. Если вы выполните функцию `sh.status()`, то увидите, что MongoDB вскоре перечислит шард:

```
rs0/svr1.example.net:27017,svr2.example.net:27017,svr3.example.net:27017
```

Имя набора, *rs0*, берется в качестве идентификатора этого шарда. Если вы когда-нибудь захотите удалить шард или перенести в него данные, можете использовать идентификатор *rs0* для его описания. Это лучше, чем использовать определенный сервер (например, *svr1.example.net*), поскольку состав членов и статус набора реплик могут со временем меняться.

После того как вы добавили набор реплик в качестве шарда, вы можете преобразовать свое приложение, чтобы вместо соединения с набором реплик оно соединялось с *mongos*. Когда вы добавляете шард, *mongos* регистрирует, что все базы данных в наборе реплик «принадлежат» этому шарду, поэтому он будет проходить все запросы к вашему новому шарду. *mongos* также автоматически обработает отказ для вашего приложения, как это сделала бы ваша клиентская библиотека: он передаст вам ошибки.

Проведите тестирование обработки отказа с первичным узлом шарда в среде разработки, чтобы убедиться, что ваше приложение правильно обрабатывает ошибки, полученные от процесса *mongos* (они должны быть идентичны ошибкам, которые вы получаете при непосредственном общении с первичным узлом).

После того как вы добавили шард, вы должны настроить всех клиентов на отправку запросов в *mongos*, вместо того чтобы связываться с набором реплик. Шардинг не будет работать правильно, если некоторые клиенты по-прежнему выполняют запросы к набору реплик напрямую (а не через *mongos*). Перенастройте всех клиентов, чтобы они связывались с *mongos* сразу же после добавления шарда, и установите правило брандмауэра, чтобы гарантировать, что они не смогут подключиться к шарду напрямую.

До выхода MongoDB версии 3.6 можно было создавать автономный сервер *mongod* в качестве шарда. В более поздних версиях MongoDB такой возможности больше нет. Все шарды должны быть наборами реплик.

Добавляем емкости

Если вы хотите добавить больше емкости, вам нужно будет добавить больше шардов. Чтобы добавить новый пустой шард, создайте набор реплик. Убедитесь, что у него есть имя, отличное от имен других шардов. Как только он будет инициализирован и у него появится первичный узел, добавьте его в свой кластер, выполнив команду `addShard` через *mongos*, указав имя нового набора реплик и его хоста как сиды.

Если у вас есть несколько существующих наборов реплик, которые не являются шардами, вы можете добавить их все в качестве новых шардов в свой кластер, если у них нет общих имен баз данных.

Например, если у вас был один набор реплик с базой данных *blog*, один набор с базой данных *calendar* и один с базами данных *mail, tel* и *music*, вы можете добавить каждый набор реплик в виде шарда и получить кластер с тремя шардами и пятью базами данных. Однако если у вас есть четвертый набор реплик, в котором также есть база данных с именем *tel*, *mongos* откажется добавить его в кластер.

Шардинг данных

MongoDB не будет распространять ваши данные автоматически, пока вы не скажете, как это сделать. Вы должны явно указать базе данных и коллекции, что вы хотите, чтобы они распространялись. Например, предположим, что вы хотите разделить коллекцию *artists* в базе данных *music* по ключу `"name"`. Во-первых, вы должны активировать шардинг для базы данных:

```
> db.enableSharding("music")
```

Разделение базы данных всегда является обязательным условием для разделения одной из ее коллекций.

После того как вы активировали шардинг на уровне базы данных, вы можете разделить коллекцию, выполнив функцию `sh.shardCollection()`:

```
> sh.shardCollection("music.artists", {"name": 1})
```

Теперь коллекция *artists* будет разделена по ключу `"name"`. Если вы разделяете существующую коллекцию, в поле `"name"` должен быть индекс; в противном случае вызов `shardCollection` вернет ошибку. Если вы получили сообщение об ошибке, создайте индекс (*mongos* вернет предложенный индекс как часть сообщения об ошибке) и повторно выполните команду `shardCollection`.

Если коллекция, которую вы разбиваете, еще не существует, *mongos* автоматически создаст для вас индекс ключа шардинга.

Команда `shardCollection` разбивает коллекцию на чанки, являющиеся единицами, которые MongoDB использует для перемещения данных. Как

только команда вернется успешно, MongoDB начнет разносить коллекцию по шардам в вашем кластере. Этот процесс не мгновенный. Если коллекции большие, на то, чтобы завершить его, могут уйти часы. Это время можно уменьшить с помощью предварительного разделения, когда чанки создаются на шардах перед загрузкой данных. Данные, загруженные после этого момента, будут вставлены непосредственно в текущий шард без дополнительного распределения.

Каждый процесс *mongos* всегда должен знать, где найти документ, учитывая его ключ шардинга. Теоретически MongoDB может отслеживать, где находился каждый документ, но в случае с коллекциями с миллионами или миллиардами документов это становится неудобно. Таким образом, MongoDB группирует документы в чанки, которые являются документами в заданном диапазоне ключа шардинга. Чанки всегда находятся на одном шарде, поэтому MongoDB может хранить небольшую таблицу чанков, отображенных в шарды.

Например, если ключом коллекции пользователей является `{"age": 1}`, один чанк может быть всеми документами с полем `"age"` от 3 до 17. Если процесс *mongos* получает запрос на `{"age": 5}`, он может направить его в шард, где находится этот чанк.

По мере записи количество и размер документов в чанки могут меняться. Операции вставки могут привести к тому, что чанк будет содержать больше документов и меньше документов удаляется. Например, если бы мы делали игру для маленьких детей и детей младшего и среднего школьного возраста, наш чанк с диапазоном возрастов от 3 до 17 лет становился бы все больше и больше (можно надеяться). Там были бы почти все наши пользователи, и все это на одном шарде, что несколько противоречит идее распределения данных. Таким образом, когда размер чанка увеличивается до определенного размера, MongoDB автоматически разделяет его на два фрагмента меньшего размера. В этом примере исходный чанк можно разбить на один чанк, содержащий документы с возрастами в диапазоне от 3 до 11, и еще один чанк, содержащий документы с возрастами в диапазоне в возрасте от 12 до 17 лет. Обратите внимание, что эти два чанка по-прежнему охватывают весь тот возрастной диапазон, который охватывал исходный чанк: 3–17. По мере того как эти новые чанки будут расти, их можно разбить на еще более мелкие фрагменты, пока не появятся чанки для каждого возраста.

У вас не могут быть чанки с перекрывающимися диапазонами, например 3–15 и 12–17. Если бы это было возможно, MongoDB должна была бы проверить оба чанка, пытаясь найти дублирующийся возраст, например 14. Более эффективно смотреть только в одном месте, особенно после того, как чанки начинают двигаться вокруг кластера.

Документ всегда принадлежит одному и только одному чанку. Одним из следствий этого правила является то, что вы не можете использовать поле

массива в качестве ключа шардинга, поскольку MongoDB создает несколько индексных записей для массивов. Например, если у документа стоит [5, 26, 83] в поле `"age"`, он будет принадлежать трем чанкам.

Существует распространенное заблуждение, согласно которому данные в чанках физически сгруппированы на диске. Это неверно: чанки не влияют на то, как сервер mongod хранит данные коллекции.

Диапазоны чанков

Каждый чанк описывается диапазоном, который он содержит. Вновь разделенная коллекция начинается с одного чанка, и каждый документ находится в этом чанке. Ее границы простираются от отрицательной бесконечности до бесконечности, что обозначено `$minKey` и `$maxKey` в оболочке.

По мере роста этих чанков MongoDB автоматически разбивает ее на два фрагмента с диапазоном от отрицательной бесконечности до <*некоторого значения*> и от <*некоторого значения*> до бесконечности. <*Некоторое значение*> одинаково для обоих чанков: нижний чанк содержит все до (но не включая) <*некоторого значения*>, а верхний чанк содержит <*некоторое значение*> и все, что выше.

Возможно, будет более понятно, если привести пример. Предположим, что мы осуществляем шардинг по возрасту, как было описано ранее. Все документы с возрастом в диапазоне от 3 до 17 содержатся в одном чанке: `3 ⩽ "age" < 17`. Когда он разделяется, мы получаем два диапазона: `3 ⩽ "age" < 12` в одном чанке и `12 ⩽ "age" < 17` в другом. 12 называется *точкой расщепления*.

Информация о чанках хранится в коллекции *config.chunks*. Если вы посмотрите на содержимое этой коллекции, то увидите документы, которые выглядят примерно так (некоторые поля были опущены для ясности):

```
> db.chunks.find(criteria, { "min": 1, "max": 1 }) {
    "_id": "test.users-age_-100.0",
    "min": { "age": -100 },
    "max": { "age": 23 }
}
{
    "_id": "test.users-age_23.0",
    "min": { "age": 23 },
    "max": { "age": 100 }
}
```

```
{
    "_id": "test.users-age_100.0",
    "min": { "age": 100 },
    "max": { "age": 1000 }
}
```

Основываясь на показанных документах коллекции *config.chunks*, ниже приводится несколько примеров того, где будут находиться различные документы:

`{"_id": 123, "age": 50}`
: Этот документ будет находиться во втором чанке, поскольку в нем содержатся все документы с возрастом в диапазоне от 23 до 100.

`{"_id": 456, "age": 100}`
: Этот документ будет находиться в третьем чанке, поскольку нижние границы включены. Второй чанк содержит все документы до возраста, равного 100, но не документы, где возраст равен 100.

`{"_id": 789, "age": -101}`
: Этот документ не будет находиться ни в одном из первых двух чанков. Он будет в неком чанке с диапазоном ниже, чем у первого.

С составным ключом шардинга диапазоны работают так же, как и сортировка по двум ключам. Например, предположим, что у нас есть ключ по `{"username" : 1, "age" : 1}`. Тогда у нас могут быть следующие диапазоны чанков:

```
{
    "_id": "test.users-username_MinKeyage_MinKey",
    "min": {
        "username": { "$minKey": 1 },
        "age": { "$minKey": 1 }
    },
    "max": {
        "username": "user107487",
        "age": 73
    }
}
{
    "_id": "test.users-username_\"user107487\"age_73.0",
    "min": {
        "username": "user107487",
        "age": 73
    },
    "max": {
```

```
            "username": "user114978",
            "age": 119
        }
    }
    {
        "_id": "test.users-username_\"user114978\"age_119.0",
        "min": {
            "username": "user114978",
            "age": 119
        },
        "max": {
            "username": "user122468",
            "age": 68
        }
    }
```

Таким образом, процесс *mongos* может легко найти, в каком чанке находится человек с определенным именем пользователя (или с заданным именем пользователя и возрастом). Однако, учитывая только возраст, *mongos* придется проверять все или почти все чанки. Если бы мы хотели иметь возможность направлять запросы по возрасту в нужный чанк, нам пришлось бы использовать «противоположный» ключ шардинга: `{"age": 1, "username ": 1}`. Часто это приводит к путанице: диапазон, превышающий вторую половину ключа, будет разделен по нескольким чанкам.

Расщепление чанков

Каждый первичный сервер *mongod* в шардинге отслеживает свои текущие чанки и, как только они достигают определенного порога, проверяет, нужно ли расщеплять чанки, как показано на рис. 15.1 и 15.2. Если чанки нужно расщепить, *mongod* запросит конфигурационное значение глобального размера чанка у конфигурационных серверов. Затем он выполнит разбиение на части и обновит метаданные на этих серверах. Новые документы чанка создаются на конфигурационных серверах, и диапазон старого чанка («max») модифицируется. Если чанк является верхним чанком шарда, то сервер *mongod* попросит балансировщика переместить этот чанк в другой шард. Идея состоит в том, чтобы не допустить, чтобы шард стал «горячим» там, где ключ шардинга использует монотонно увеличивающийся ключ.

Однако шард может не найти точек расщепления, даже для большого чанка, поскольку существует ограниченное количество способов допустимого разбиения чанков. Любые два документа с одним и тем же ключом шардинга должны находиться в одном и том же чанке, поэтому чанки могут быть разделены только между документами, в которых значение ключа

меняется. Например, если ключ шардинга – это `"age"`, приведенный ниже чанк можно разбить в точках, где ключ изменился, как указано:

```
{"age" : 13, "username" : "ian"}
{"age" : 13, "username" : "randolph"}
------------ // точка разделения
{"age" : 14, "username" : "randolph"}
{"age" : 14, "username" : "eric"}
{"age" : 14, "username" : "hari"}
{"age" : 14, "username" : "mathias"}
------------ // точка разделения
{"age" : 15, "username" : "greg"}
{"age" : 15, "username" : "andrew"}
```

Первичный сервер *mongod* требует, чтобы только верхний чанк шарда при разбиении был перенесен в балансировщик. Другие чанки останутся на шарде, только если не будут перемещены вручную.

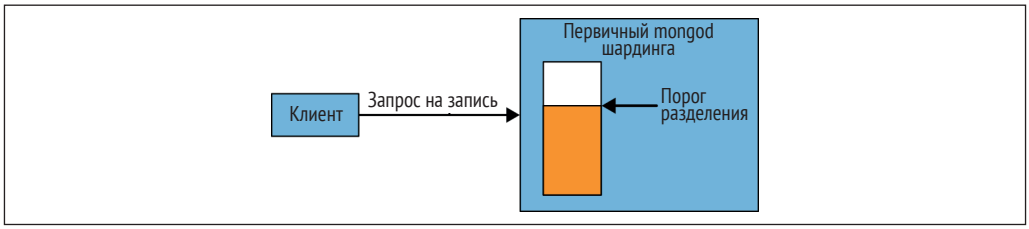

Рис. 15.1. Когда клиент будет выполнять запись в чанк, *mongod* проверит порог расщепления

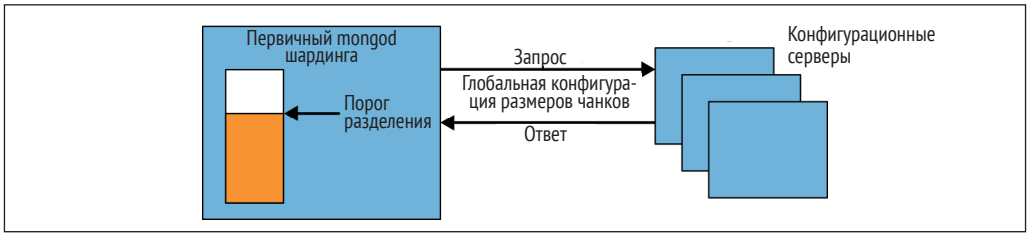

Рис. 15.2. Если порог расщепления достигнут, *mongod* отправит запрос балансировщику для переноса верхнего чанка; в противном случае она останется на шарде

Однако если бы в чанке содержались приведенные ниже документы, его нельзя было бы разделить (только если приложение не начало вставлять дробные возрасты):

```
{"age" : 12, "username" : "kevin"}
{"age" : 12, "username" : "spencer"}
{"age" : 12, "username" : "alberto"}
{"age" : 12, "username" : "tad"}
```

Таким образом, важно иметь различные значения для своего ключа шардинга. Другие важные свойства будут рассмотрены в следующей главе.

Если один из конфигурационных серверов не работает, когда *mongod* пытается выполнить разбиение, последний не сможет обновить метаданные (как показано на рис. 15.3).

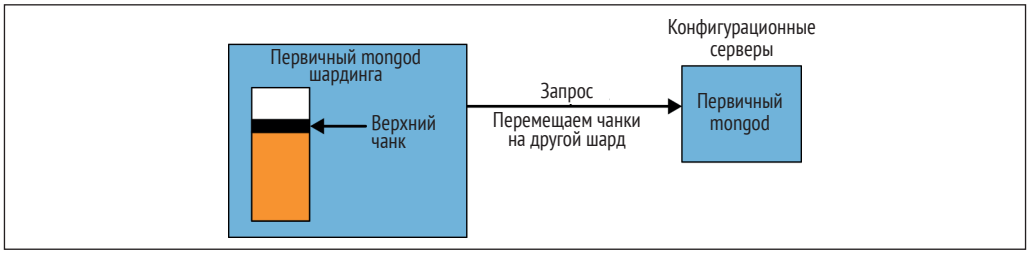

Рис. 15.3. Сервер *mongod* выбирает точку расщепления и пытается проинформировать конфигурационный сервер, но не может связаться с ним; таким образом, он по-прежнему превышает пороговое значение расщепления, и любые последующие операции записи будут запускать этот процесс снова

Все конфигурационные серверы должны работать и быть доступны для разбиения. Если *mongod* продолжит получать запросы на запись для чанка, он будет продолжать пытаться разделить чанк и потерпит неудачу. Пока конфигурационные серверы не работоспособны, разделение по-прежнему не будет работать, и все попытки разбиения могут замедлить работу *mongod* и шарда (что повторяет процесс, показанный на рис. 15.1–15.3 для каждой входящей записи). Этот процесс *mongod*, когда он несколько раз пытается разделить чанк и у него не получается, называется *шквалом расщеплений*. Единственный способ предотвратить его – обеспечить, чтобы ваши конфигурационные серверы были доступны и находились в работоспособном состоянии как можно дольше.

Балансировщик

Балансировщик отвечает за перенос данных. Он регулярно проверяет дисбаланс между шардами и, если обнаруживает его, начинает переносить чанк. В MongoDB версии 3.4 и выше балансировщик расположен на первичном узле набора реплик конфигурационного сервера; до появления этой версии каждый процесс *mongos* играл роль «балансировщика» время от времени.

Балансировщик – фоновый процесс на первичном узле набора реплик конфигурационного сервера, который отслеживает число чанков на каждом шарде. Он становится активным только тогда, когда количество чанков достигает определенного порога миграции.

В MongoDB версии 3.4 и выше число согласованных миграций увеличилось до одной миграции на шард, при этом максимальное число согласованных миграций составляет половину от общего числа шардов. В более ранних версиях поддерживалась только одна согласованная миграция.

Предполагая, что некоторые коллекции достигли порогового значения, балансировщик начнет переносить чанки. Он выбирает чанк из перегруженного шарда и спрашивает последнего, должен ли он разбить этот чанк перед миграцией. Как только он делает все необходимые разбиения, он переносит этот чанк (чанки) на машину с меньшим количеством чанков.

Приложение, использующее кластер, не должно знать, что данные перемещаются: все операции чтения и записи направляются в старый чанк до тех пор, пока перемещение не будет завершено. После обновления метаданных любой процесс *mongos*, пытающийся получить доступ к данным в старом местоположении, получит ошибку. Эти ошибки не должны быть видны клиенту: *mongos* автоматически обработает ошибку и попытается снова выполнить операцию с новым шардом.

Это частая причина ошибок, которые можно увидеть в журналах *mongos*. Они связаны с неспособностью выполнить команду `setShardVersion`. Когда процесс *mongos* получает такой тип ошибки, он ищет новое расположение данных на конфигурационных серверах, обновляет свою таблицу чанков и снова пытается сделать запрос. Если он успешно извлекает данные из нового местоположения, он вернет их клиенту, как будто все в порядке (но он напечатает в журнале сообщение о том, что произошла ошибка).

Если *mongos* не может получить новое местоположение чанка из-за недоступности конфигурационных серверов, клиент вернет ошибку. Это еще одна причина, по которой важно всегда поддерживать работоспособность конфигурационных серверов.

Сличения

Сличения в MongoDB позволяют задавать языковые правила для сравнения строк. Примеры этих правил включают в себя сравнение букв и символов ударения. Можно разбить коллекцию, которая является сличением по умолчанию. Два требования: у коллекции должен быть индекс, префикс которого является ключом шардинга, а индекс также должен иметь сличение `{locale: "simple"}`.

Потоки изменений

Потоки изменений позволяют приложениям отслеживать изменения данных в базе данных в режиме реального времени. До выхода MongoDB версии 3.6 это было возможно только путем подгонки журнала операций и было сложной подверженной ошибкам операции. Потоки изменений предоставляют механизм подписки для всех изменений данных в коллекции, наборе коллекций, базе данных или во всем развертывании. Эта функция использует фреймворк агрегации. Она позволяет приложениям фильтровать определенные изменения или преобразовывать полученные уведомления об изменениях. В разделенном кластере все операции потока изменений должны выполняться с *mongos*.

Изменения в разделенном кластере упорядочены благодаря использованию глобальных логических часов. Это гарантирует порядок изменений, а потоковые уведомления можно смело интерпретировать в порядке их получения. Процессу *mongos* нужно проверить каждый шард после получения уведомления об изменении, чтобы убедиться, что ни один шард не видел более недавние изменения. Уровень активности кластера и географическое распределение шардов могут влиять на время ответа в течение этой проверки. Использование фильтров уведомлений может улучшить время отклика в этих ситуациях.

Несколько замечаний и предостережений касательно использования потоков изменений с разделенным кластером. Вы открываете поток изменений, выполняя операцию открытого потока изменений. В случае разделенных развертываний это должно делаться с *mongos*. Если операция обновления с `multi: true` выполняется для разделенной коллекции с открытым потоком изменений, то можно отправлять уведомления для потерянных документов. Если шард удален, это может привести к закрытию курсора открытого потока изменений. Более того, этот курсор может быть не полностью возобновляемым.

Глава 16

Выбор ключа шардинга

Наиболее важной задачей при использовании шардинга является выбор способа распределения ваших данных. Чтобы сделать разумный выбор в этом отношении, вы должны понимать, как MongoDB распространяет данные. Эта глава поможет вам правильно выбрать ключ шардинга. Темы, о которых пойдет речь:

- как выбрать один из нескольких возможных ключей шардинга;
- ключи шардинга для нескольких случаев использования;
- что нельзя использовать в качестве ключа шардинга;
- альтернативные стратегии, если вы хотите настроить распределение данных;
- как вручную разделить свои данные.

Предполагается, что вы усвоили основные компоненты шардинга, описанные в предыдущих двух главах.

Подводя итоги использования

Когда вы разделяете коллекцию, то выбираете одно или два поля, чтобы использовать их для разбиения данных. Этот ключ (или ключи) называется ключом шардинга. После того как вы разделили коллекцию, ключ шардинга нельзя изменить, поэтому важно сделать правильный выбор.

Чтобы выбрать правильный ключ, вы должны понимать свою рабочую нагрузку и то, как ваш ключ будет распределять запросы вашего приложения. Это может быть сложно изобразить, поэтому попробуйте поработать над примерами или, что еще лучше, использовать резервный набор данных с образцом трафика. В этом разделе есть множество диаграмм и объяснений, но лучше всего, если вы поэкспериментируете с собственными данными.

Кода речь идет о коллекции, которую вы планируете разбить, начните с ответов на следующие вопросы:

- сколько шардов вы планируете добавить? Кластер из трех шардов обладает гораздо большей гибкостью, чем кластер, состоящий из

тысячи. По мере роста кластера вы не должны планировать запуск запросов, которые могут затронуть все сегменты, поэтому почти все запросы должны включать ключ шардинга;

- вы используете шардинг, чтобы уменьшить задержку чтения или записи? (Под задержкой подразумевается количество времени, необходимое для какого-либо действия, например запись занимает 20 мс, но вам нужно, чтобы она занимала 10 мс.) Уменьшение задержки записи обычно включает в себя отправку запросов на географически более близкие или более мощные машины;
- вы используете шардинг, чтобы увеличить пропускную способность по чтению или записи? (Под пропускной способностью подразумевается число запросов, которые кластер может обрабатывать одновременно, например кластер может выполнить 1000 записей за 20 мс, но вам необходимо выполнить 5000 записей за 20 мс.) Повышение пропускной способности обычно включает в себя добавление параллелизации, и необходимо убедиться, что запросы распределены по кластеру равномерно;
- вы используете шардинг, чтобы увеличить системные ресурсы (например, дать MongoDB больше оперативной памяти на ГБ данных)? Если это так, вам нужно, чтобы размер рабочего множества был как можно меньше.

Используйте эти ответы для оценки приведенных ниже описаний ключа шардинга и решите, будет ли ключ, который вы рассматриваете, хорошо работать в вашей ситуации. Это дает вам необходимые целевые запросы? Меняет ли это пропускную способность или задержку вашей системы нужными вам способами? Если вам необходимо компактное рабочее множество, получаете ли вы его?

Иллюстрация распределений

Наиболее распространенные способы разделения данных – это монотонно возрастающие ключи, случайные ключи и ключи с привязкой к местоположению пользователя. Существуют и другие типы ключей, которые можно использовать, но большинство вариантов использования подпадают под одну из этих категорий. Различные типы распределений обсуждаются в последующих разделах.

Монотонно возрастающие ключи

Монотонно возрастающие ключи шардинга, как правило, представляют собой нечто вроде поля `"date"` или `ObjectId` – что-то, что постоянно увеличивается со временем. Первичный ключ с автоприращением – еще один

пример монотонно возрастающего поля, хотя и не очень хорошо отображаемого в MongoDB (только если вы не импортируете из другой базы данных).

Предположим, что мы используем шардинг в монотонно возрастающем поле, например `"_id"` в коллекции с использованием `ObjectId`. В случае с `"_id"` данные будут разбиты на чанки диапазонов `"_id"`, как показано на рис. 16.1. Эти чанки будут распределены по нашему кластеру, скажем, из трех шардов, как показано на рис. 16.2.

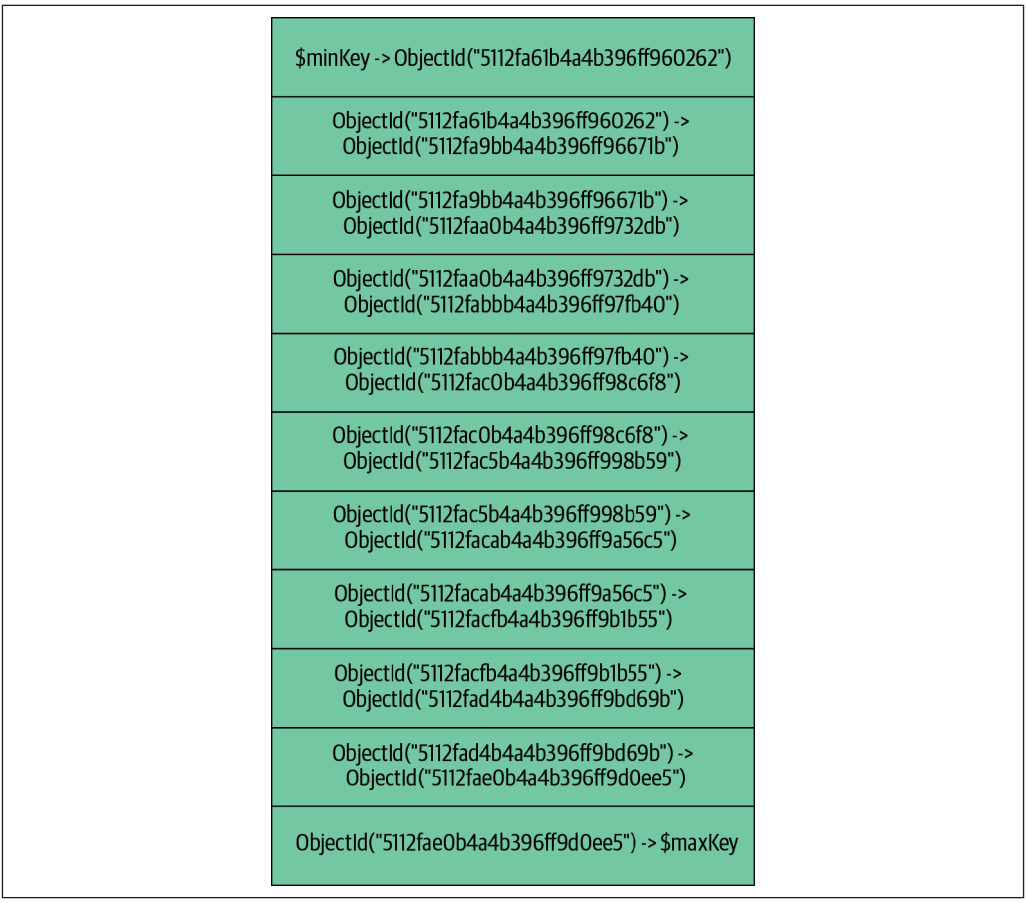

Рис. 16.1. Коллекция разделена на диапазоны ObjectIds; каждый диапазон – это чанк

Предположим, мы создаем новый документ. В каком чанке он будет находиться? Ответ: в чанке с диапазоном `ObjectId("5112fae0b4a4b396ff9d0ee5")` через `$maxKey`.

Она называется максимальный чанк, поскольку содержит `$maxKey`.

Если мы вставим еще один документ, он также будет находиться в этом чанке. Фактически каждый последующий документ будет вставлен в мак-

симальный чанк! Поле `"_id"` каждой операции вставки будет ближе к бесконечности, чем предыдущее (потому что `ObjectId` всегда возрастают), поэтому все они попадут в максимальный чанк.

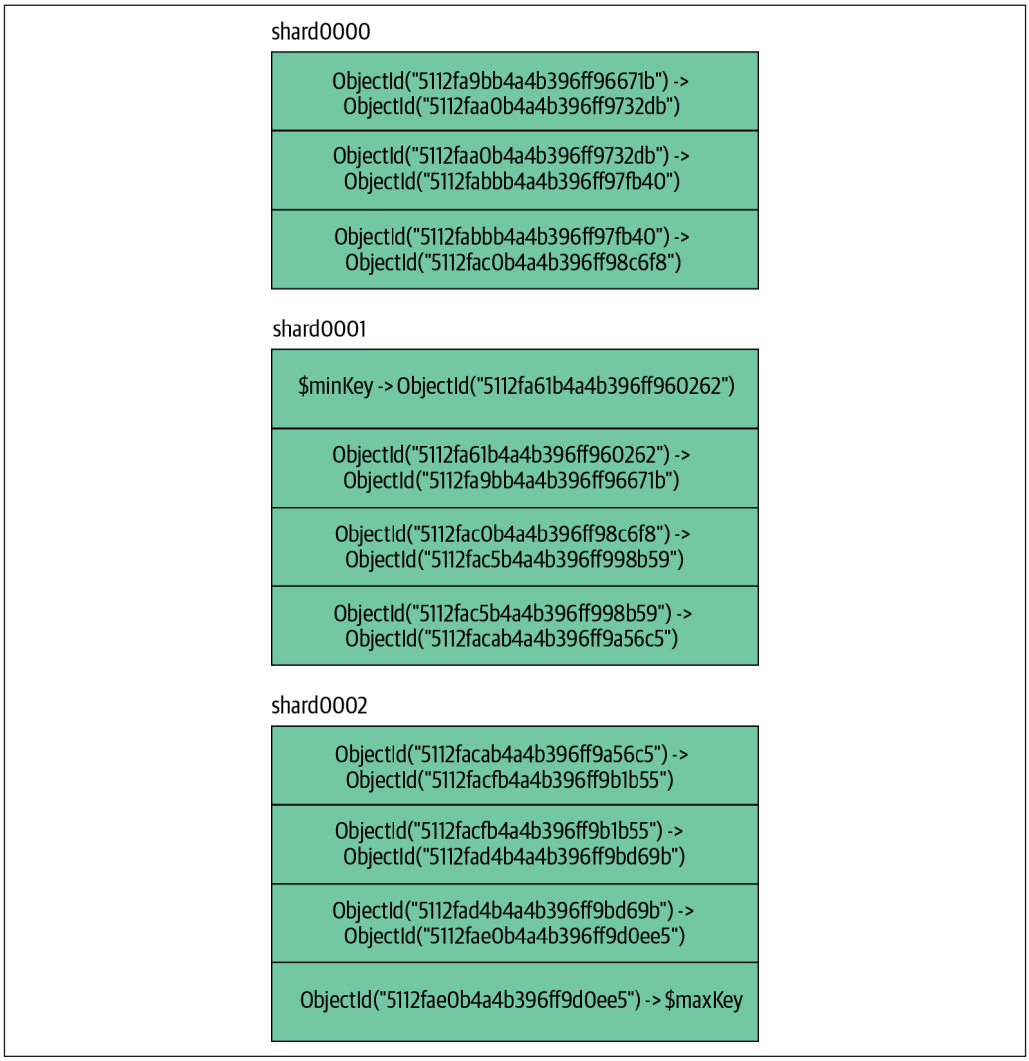

Рис. 16.2. Чанки распределяются по шардам в случайном порядке

У этого процесса есть пара интересных (и часто нежелательных) особенностей. Во-первых, все ваши операции записи будут перенаправлены на один шард (в данном случае *shard0002*). Этот чанк будет единственным, который увеличивается и разделяется, поскольку он единственный, кто получает вставки. Когда вы вставляете данные, новые чанки будут «отваливаться» от него, как показано на рис. 16.3.

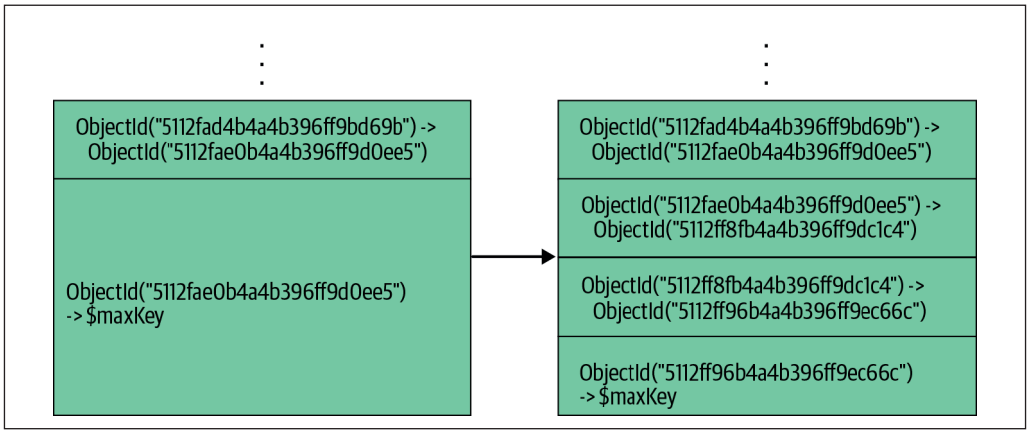

Рис. 16.3. Максимальный чанк продолжает расти и делится на несколько чанков

Данный шаблон часто мешает MongoDB поддерживать равномерный баланс чанков, поскольку все чанки создаются одним шардом. Поэтому MongoDB должна постоянно перемещать чанки в другие шарды, вместо того чтобы исправлять небольшие дисбалансы, которые могут возникнуть в более равномерно распределенных системах.

В MongoDB версии 4.2 передача функции автоматического разделения первичному серверу *mongod* добавила оптимизацию верхнему чанку, чтобы соответствовать шаблону монотонно возрастающего ключа шардинга. Балансировщик решит, в каком шарде разместить верхний чанк. Это помогает избежать ситуации, когда все новые чанки создаются только на одном шарде.

Случайно распределенные ключи

На другом конце спектра расположены случайно распределенные ключи шардинга. Случайно распределенные ключи могут быть именами пользователей, адресами электронной почты, универсальными уникальными идентификаторами, MD5-хешами или любым другим ключом, который не имеет идентифицируемого шаблона в вашем наборе данных.

Предположим, что ключ шардинга – случайное число от 0 до 1. В результате мы получим случайное распределение чанка по разным шардам, как показано на рис. 16.4.

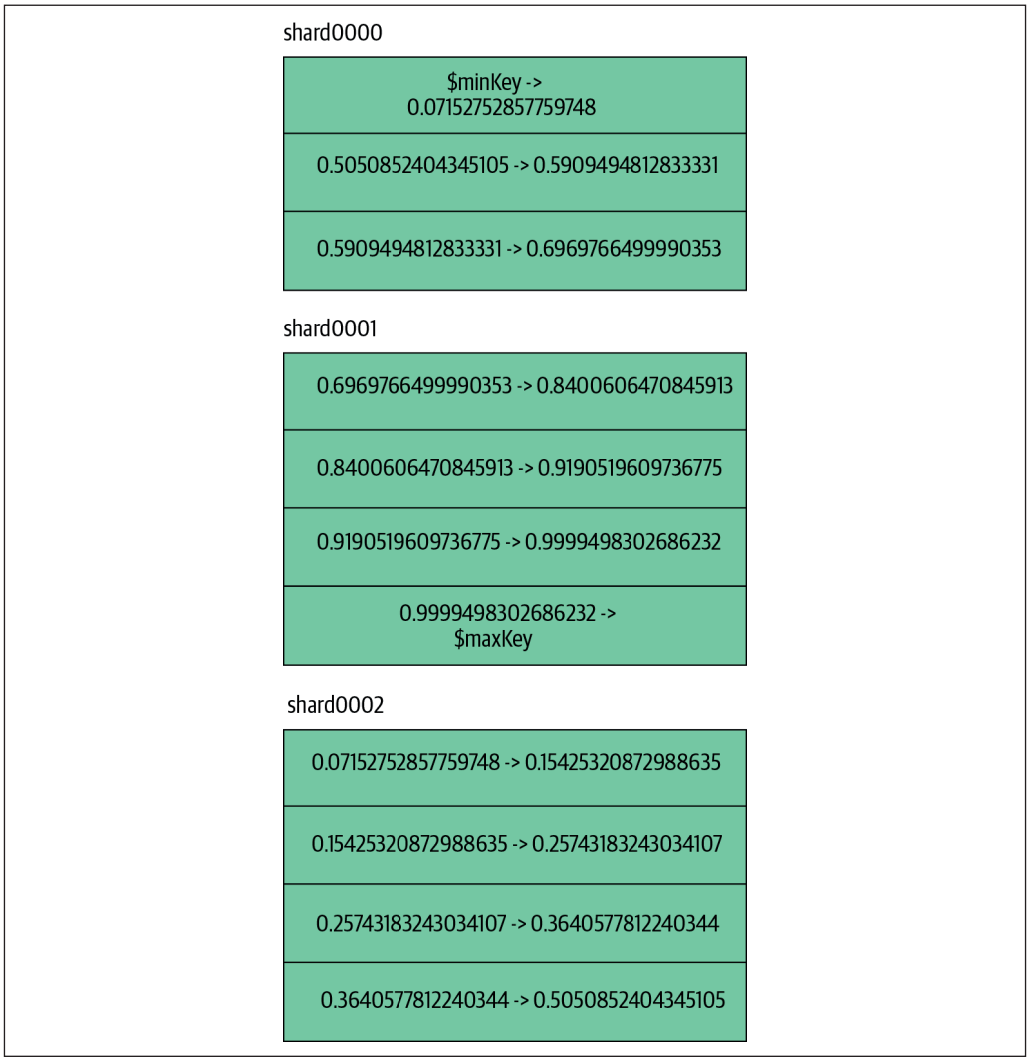

Рис. 16.4. Как и в предыдущем разделе, чанки распределены случайным образом по кластеру

По мере добавления большего количества данных случайный характер данных означает, что вставки должны попадать в каждый чанк довольно равномерно. Вы можете сами убедиться в этом, вставив 10 000 документов и посмотрев, где они в конечном итоге окажутся:

```
> var servers = {}
> var findShard = function(id) {
...     var explain = db.random.find({ _id: id }).explain();
...     for (var i in explain.shards) {
```

```
...         var server = explain.shards[i][0];
...             if (server.n == 1) {
...                 if (server.server in servers) {
...                     servers[server.server]++;
...                 } else {
...                     servers[server.server] = 1;
...                 }
...             }
...         }
...     }
> for (var i = 0; i < 10000; i++) {
...     var id = ObjectId();
...     db.random.insert({ "_id": id, "x": Math.random() });
...     findShard(id);
...     }
> servers {
    "spock:30001": 2942,
    "spock:30002": 4332,
    "spock:30000": 2726
}
```

Поскольку записи распределяются случайным образом, шарды должны расти примерно с той же скоростью, что ограничивает число миграций, которые должны произойти.

Единственный недостаток таких ключей – это то, что MongoDB не эффективна, когда речь идет об обращении случайным образом к данным, превышающим объем оперативной памяти. Однако если у вас есть возможность или вы не возражаете против снижения производительности, с помощью случайных ключей можно прекрасно распределить нагрузку по своему кластеру.

Ключи с привязкой к местоположению пользователя

Ключами с привязкой к местоположению могут быть IP-адрес пользователя, широта и долгота или обычный адрес. Они не обязательно связаны с полем физического местоположения: «местоположение» может быть более абстрактным способом, которым данные должны быть сгруппированы. В любом случае ключ с привязкой к местоположению – это ключ, в котором документы с некоторым сходством попадают в диапазон на базе этого поля. Это может быть удобно как для размещения данных рядом с пользователями, так и для хранения связанных данных вместе на диске. Также необходимо соблюдать Генеральный регламент о защите персональных данных или другие подобные постановления. MongoDB использует для этой цели зональный шардинг.

В MongoDB версии 4.0.3 и выше можно определять зоны и диапазоны зон перед разделением коллекции, в результате чего заполняются чанки как для диапазонов зон, так и для значений ключей шардинга, а также выполняется их первоначальное распределение. Это значительно снижает сложность настройки разделенных зон.

Например, предположим, у нас есть коллекция документов, которые разделены по IP-адресу. Документы будут организованы в чанки на основе их IP-адресов и случайным образом распределены по кластеру, как показано на рис. 16.5.

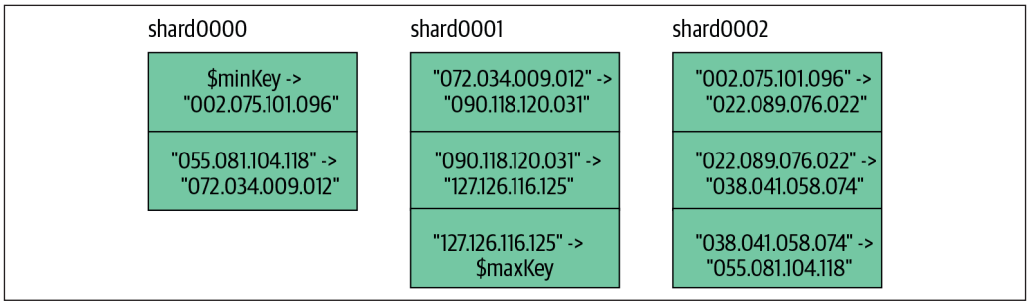

Рис. 16.5. Пример распределения чанков в коллекции IP-адресов

Если бы мы хотели, чтобы определенные диапазоны чанков были присоединены к определенным шардам, мы могли бы разбить эти чанки на зоны, а затем назначить диапазоны фрагментов для каждой зоны. В этом примере предположим, что мы хотим сохранить определенные чанки IP на определенных шардах: скажем, 56.*.*.* (блок IP-адресов почтовой службы США) на *shard0000* и 17.*.*.* (блок IP-адресов Apple) на *shard0000* или *shard0002*. Нам все равно, где находятся другие IP-адреса. Мы могли бы попросить балансировщика сделать это, настроив зоны:

```
> sh.addShardToZone ("shard0000", "USPS")
> sh.addShardToZone ("shard0000", "Apple")
> sh.addShardToZone ("shard0002", "Apple")
```

Далее мы создаем правила:

```
> sh.updateZoneKeyRange ("test.ips", {"ip": "056.000.000.000"},
... {"ip": "057.000.000.000"}, "USPS")
```

Все IP-адреса, большие или равные 56.0.0.0 и меньшие, чем 57.0.0.0, будут присоединены к шарду, обозначенному как `"USPS"`. Далее добавим правило для Apple:

```
> sh.updateZoneKeyRange ("test.ips", {"ip": "017.000.000.000"},
... {"ip": "018.000.000.000"}, "Apple")
```

Когда балансировщик будет перемещать чанки, он попытается переместить чанки с этими диапазонами в эти шарды. Обратите внимание, что этот процесс не происходит сиюминутно. Чанки, которые не были охвачены диапазоном ключей зоны, будут перемещаться как обычно. Балансировщик будет продолжать пытаться равномерно распределить чанки по шардам.

Стратегии

В этом разделе представлен ряд параметров ключей шардинга для различных типов приложений.

Хешированные ключи шардинга

Для максимально быстрой загрузки данных лучше всего использовать хешированные ключи. Хешированный ключ шардинга может сделать любое поле случайным образом распределенным, поэтому это хороший выбор, если вы собираетесь использовать монотонно возрастающий ключ в большом количестве запросов, но хотите, чтобы операции записи распределялись случайным образом.

Компромисс состоит в том, что вы никогда не сможете выполнить запрос по целевому диапазону с помощью хешированного ключа. Если вы не будете выполнять запросы по диапазону, лучше использовать хешированные ключи.

Чтобы создать хешированный ключ шардинга, для начала создадим хешированный индекс:

```
> db.users.createIndex({"username" : "hashed"})
```

Затем разбиваем коллекцию:

```
> sh.shardCollection("app.users", {"username" : "hashed"})
    { "collectionsharded" : "app.users", "ok" : 1 }
```

Если вы создаете хешированный ключ в несуществующей коллекции, команда `shardCollection` ведет себя интересно: она предполагает, что вам нужны равномерно распределенные чанки, поэтому сразу же создает кучу пустых чанков и распределяет их по кластеру.

Например, предположим, что наш кластер выглядел так, перед тем как был создан хешированный ключ:

```
> sh.status()
    ---Sharding Status---
```

```
        sharding version: { "_id": 1, "version": 3 }
    shards:
        {
            "_id": "shard0000",
            "host": "localhost:30000"
        }
        {
            "_id": "shard0001",
            "host": "localhost:30001"
        }
        {
            "_id": "shard0002",
            "host": "localhost:30002"
        }
    databases:
        {
            "_id": "admin",
            "partitioned": false,
            "primary": "config"
        }
        {
            "_id": "test",
            "partitioned": true,
            "primary": "shard0001"
        }
```

Сразу после возврата `shardCollection` в каждом шарде есть два чанка, равномерно распределяющих ключевое пространство по кластеру:

```
> sh.status()
    ---Sharding Status---
    sharding version: { "_id": 1, "version": 3 }
    shards:
        { "_id": "shard0000", "host": "localhost:30000" }
        { "_id": "shard0001", "host": "localhost:30001" }
        { "_id": "shard0002", "host": "localhost:30002" }
    databases:
        { "_id": "admin", "partitioned": false, "primary": "config" }
        { "_id": "test", "partitioned": true, "primary": "shard0001" }
            test.foo
                shard key: { "username": "hashed" }
                chunks:
                    shard0000       2
                    shard0001       2
                    shard0002       2
```

```
{ "username": { "$MinKey": true } }
    -->> { "username": NumberLong("-6148914691236517204") }
    on: shard0000 { "t": 3000, "i": 2 }
{ "username": NumberLong("-6148914691236517204") }
    -->> { "username": NumberLong("-3074457345618258602") }
    on: shard0000 { "t": 3000, "i": 3 }
{ "username": NumberLong("-3074457345618258602") }
    -->> { "username": NumberLong(0) }
    on: shard0001 { "t": 3000, "i": 4 }
{ "username": NumberLong(0) }
    -->> { "username": NumberLong("3074457345618258602") }
    on: shard0001 { "t": 3000, "i": 5 }
{ "username": NumberLong("3074457345618258602") }
    -->> { "username": NumberLong("6148914691236517204") }
    on: shard0002 { "t": 3000, "i": 6 }
{ "username": NumberLong("6148914691236517204") }
    -->> { "username": { "$MaxKey": true } }
    on: shard0002 { "t": 3000, "i": 7 }
```

Обратите внимание, что в коллекции пока еще нет документов, но, когда вы начинаете вставлять их, записи должны быть равномерно распределены по шардам с самого начала. Обычно вам придется ждать, когда чанки вырастут, разделятся и переместятся, чтобы начать запись в другие шарды. При таком автоматическом заполнении у вас сразу же будут диапазоны чанков на всех шардах.

Существуют некоторые ограничения на то, каким может быть ваш ключ шардинга, если вы используете хешированный ключ. Во-первых, нельзя использовать параметр `unique`. Как и в случае с другими ключами шардинга, нельзя использовать поля массива. Наконец, имейте в виду, что значения с плавающей точкой будут округлены до целых чисел перед хешированием, поэтому и 1, и 1.999999 будут хешироваться до одного и того же значения.

Хешированные ключи шардинга для GridFS

Прежде чем пытаться разбить коллекции GridFS, убедитесь, что вы понимаете, как GridFS хранит данные (*см.* главу 6).

В приведенном ниже объяснении термин «чанки» перегружен, поскольку GridFS разбивает файлы на чанки, а в результате шардинга коллекции разбиваются на чанки. Таким образом, два типа чанков называются «чанки GridFS» и «чанки шардинга».

Коллекции GridFS, как правило, являются отличными кандидатами для шардинга, поскольку содержат огромное количество файловых данных. Однако ни один из индексов, которые автоматически создаются для коллекции *fs.chunks*, не являются особо подходящими ключами: `{"_id": 1}` является монотонно возрастающим ключом, а `{"files_id": 1, "n": 1}` выбирает поле *fs.files* `"_id"`, таким образом, это тоже монотонно возрастающий ключ.

Однако если вы создадите хешированный индекс в поле `"files_id"`, каждый файл будет случайным образом распределен по кластеру, и файл всегда будет содержаться в одном чанке. Таким образом, мы берем лучшее из обоих миров: записи будут равномерно распределяться по всем шардам, а при чтении данных файла потребуется только одно обращение к шарду.

Чтобы настроить все это, нужно создать новый индекс `{"files_id": "hashed"}` (на момент написания этих строк процесс *mongos* не может использовать подмножество составного индекса в качестве ключа шардинга). Затем разбейте коллекцию:

```
> db.fs.chunks.ensureIndex({"files_id" : "hashed"})
> sh.shardCollection("test.fs.chunks", {"files_id" : "hashed"})
    { "collectionsharded" : "test.fs.chunks", "ok" : 1 }
```

Напомним, что коллекция *fs.files* может или не может нуждаться в шардинге, поскольку она будет намного меньше, чем коллекция *fs.chunks*. Можете разбить ее, если хотите, но вряд ли это будет необходимо.

Стратегия «пожарного шланга»

Если у вас есть несколько серверов, которые являются более мощными по сравнению с другими, можно позволить им обрабатывать пропорционально большую нагрузку, чем обрабатывают ваши менее мощные серверы. Например, предположим, у вас есть шард, который может в 10 раз увеличить нагрузку на другие ваши машины. К счастью, у вас есть 10 других шардов. Вы можете принудительно сделать так, чтобы все вставки отправлялись на более мощный шард, а затем позволить балансировщику переместить более старые чанки в другие шарды, что дало бы операции записи с меньшей задержкой.

Чтобы использовать эту стратегию, нужно прикрепить самый высокий чанк к более мощному шарду.

Для начала свяжем этот шард с зоной:

```
> sh.addShardToZone("<shard-name>", "10x")
```

Затем мы привязываем текущее значение монотонно возрастающего ключа через бесконечность к этому шарду, поэтому все новые записи идут на него:

```
> sh.updateZoneKeyRange("<dbName.collName>", {"_id" : ObjectId()},
... {"_id" : MaxKey}, "10x")
```

Теперь все вставки будут перенаправлены в этот последний чанк, который всегда будет находиться в шарде с зоной `"10x"`.

Однако диапазоны с данного момента до бесконечности будут захвачены этим шардом, если мы не изменим диапазон ключей зоны. Для решения этой проблемы мы могли бы создать задание cron, чтобы обновлять диапазон ключей один раз в день, например:

```
> use config
> var zone = db.tags.findOne({"ns" : "<dbName.collName>",
... "max" : {"<shardKey>" : MaxKey}})
> zone.min.<shardKey> = ObjectId()
> db.tags.save(zone)
```

Тогда все чанки предыдущего дня смогут перейти к другим шардам.

Еще одним недостатком этой стратегии является то, что она требует масштабирования изменений. Если ваш самый мощный сервер больше не может обрабатывать количество входящих записей, нельзя просто разделить нагрузку между этим сервером и другим.

Если у вас нет высокопроизводительного сервера, или вы не используете зональный шардинг, не применяйте монотонно возрастающий ключ в качестве ключа шардинга. Если вы это сделаете, все записи будут отправлены в один шард.

Несколько хот-спотов

Автономные серверы *mongod* наиболее эффективны при выполнении монотонно возрастающих записей, что противоречит шардингу, поскольку последний наиболее эффективен, когда записи распространяются по кластеру. С помощью описанного здесь метода в основном создается несколько хот-спотов – оптимально по нескольку на каждый шард, – так чтобы записи были равномерно распределены по кластеру, но в пределах шарда были монотонно возрастающими.

Для этого мы используем составной ключ шардинга. Первое значение в составном ключе – грубое, случайное значение с низковатой кардинальностью. Можно изобразить каждое значение в первой части ключа в виде чанков, как показано на рис. 16.6. Это в конечном итоге сработает, когда вы добавите больше данных, хотя, вероятно, они никогда не будут разделены аккуратно (прямо на линиях `$minKey`). Однако если вы вставите достаточно данных, у вас в конечном итоге будет примерно по одному чанку на каждое случайное значение. Продолжая вставлять данные, вы получите несколько чанков с одинаковым случайным значением, что подводит нас ко второй части ключа шардинга.

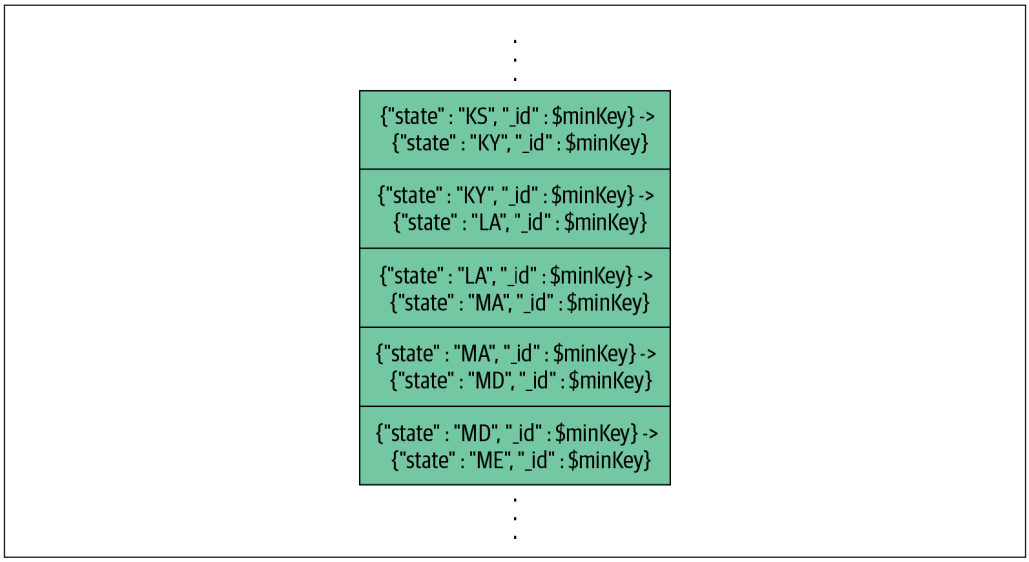

Рис. 16.6. Подмножество чанков: каждый чанк содержит одно состояние и диапазон значений `"_id"`

Вторая часть ключа – это монотонно возрастающий ключ. Это означает, что в пределах чанка значения всегда увеличиваются, как показано в документах на рис. 16.7. Таким образом, если бы у вас было по одному чанку на шард, у вас была бы идеальная настройка: монотонно возрастающие записи на каждом шарде, как показано на рис. 16.8. Конечно, при наличии *n* чанков с *n* хот-спотами распределение по *n* шардам не дает много возможностей для расширения: добавьте новый шард, и он не получит никаких записей, потому что на нем нет хот-спот-чанков. Таким образом, вам нужно несколько таких чанков на шард (чтобы у вас было место для роста), но их не должно быть слишком много. Их наличие сохранит эффективность монотонно возрастающих записей, но наличие, скажем, тысячи хот-спотов на шарде в конечном итоге будет эквивалентно случайной записи.

Можно представить себе эту настройку, когда каждый чанк – это стек монотонно возрастающих документов. На каждом шарде есть несколько стеков, каждый из которых монотонно возрастает до тех пор, пока чанк не будет разбит. После разделения чанков только один новый чанк будет хот-спотом: другой чанк, по сути, будет «мертвым» и уже больше не будет расти. Если стеки равномерно распределены по шардам, записи будут распределены равномерно.

Правила и рекомендации

Есть несколько практических ограничений, о которых нужно знать, прежде чем выбирать ключ шардинга.

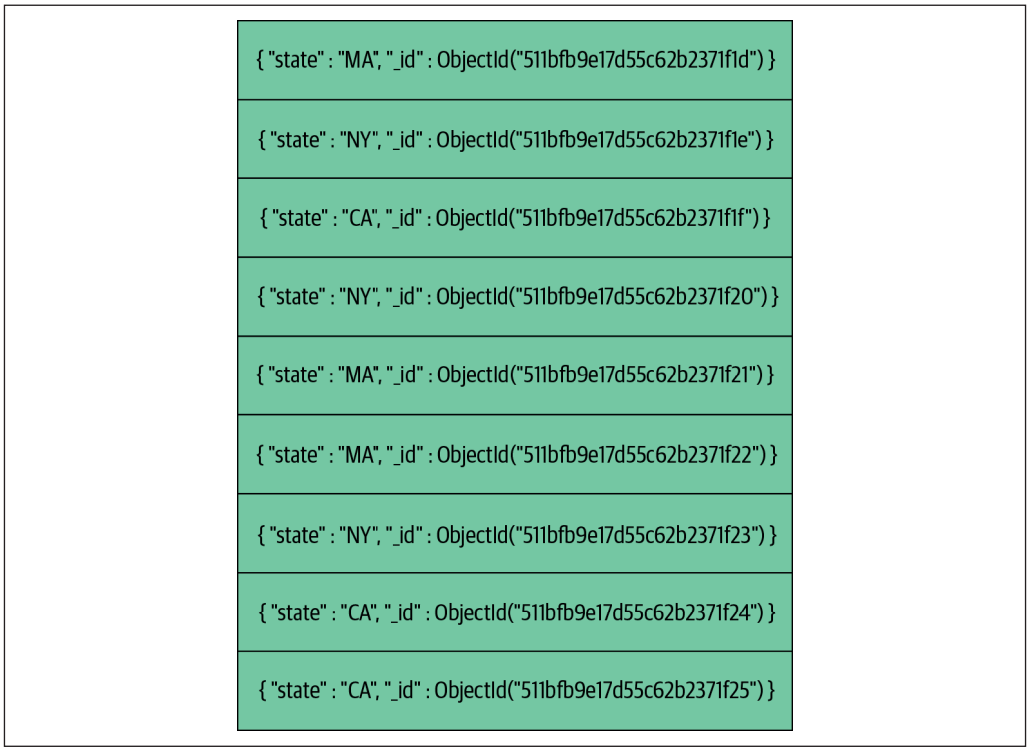

Рис. 16.7. Пример списка вставленных документов (обратите внимание, что все значения `"_id"` возрастают)

Определение того, какой выбрать, и создание ключей шардинга должны напоминать индексирование, поскольку эти две концепции похожи. Фактически часто ваш ключ может быть индексом, который вы используете чаще всего (или какой-то его вариант).

Ограничения

Ключи шардинга не могут быть массивами. Функция `sh.shardCollection()` завершится ошибкой, если какой-либо ключ имеет значение массива, и вставлять массив в это поле недопустимо.

После вставки значение ключа документа может быть изменено, если только поле ключа не является неизменяемым полем `_id`. В более старых версиях MongoDB до выхода версии 4.2 изменить значение ключа шардинга документа было невозможно.

Большинство специальных типов индексов нельзя использовать для ключей шардинга. В частности, нельзя использовать геопространственный индекс. Использование хешированного индекса разрешено, как было описано ранее.

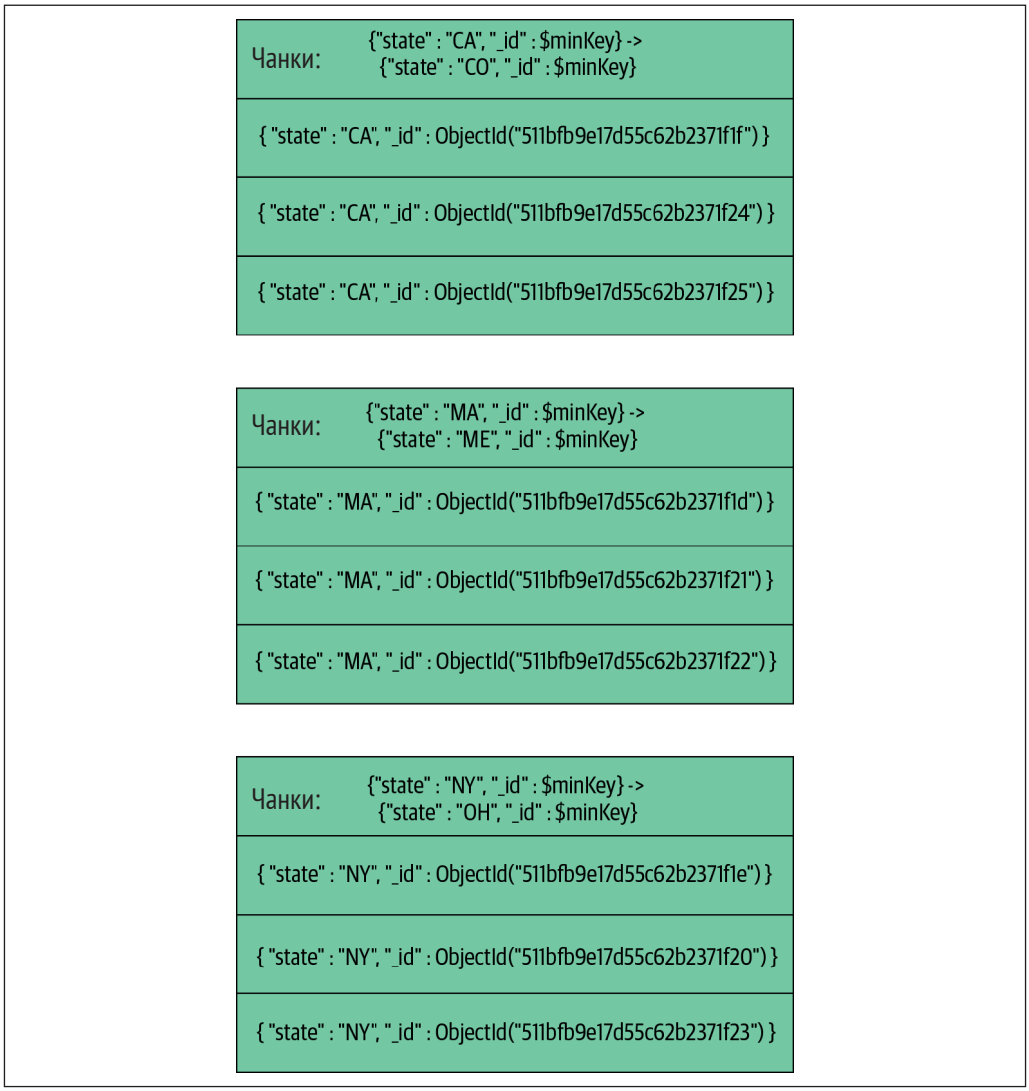

Рис. 16.8. Вставленные документы, разбитые на чанки (обратите внимание, что внутри каждого чанка `"_id"` значения возрастают)

Кардинальность

Независимо от того, «скачет» ли ваш ключ или постоянно увеличивается, важно выбрать ключ со значениями, которые будут варьироваться. Как и в случае с индексами, шардинг лучше работает с полями с высокой кардинальностью. Если, например, у вас есть ключ `"logLevel"`, у которого есть только значения `"DEBUG"`, `"WARN"` или `"ERROR"`, MongoDB не сможет разбить ваши данные на более чем три чанка (потому что для ключа шардинга будет только три разных значения). Если у вас есть ключ с небольшим из-

менением и вы все равно хотите использовать его в качестве ключа шардинга, это можно сделать путем создания составного ключа шардинга по этому ключу и ключа, который отличается сильнее, например `"logLevel"` и `"timestamp"`. Важно, чтобы комбинация ключей имела высокую кардинальность.

Управление распределением данных

Иногда автоматическое распределение данных не будет соответствовать вашим требованиям. В этом разделе приводится несколько вариантов, помимо выбора ключа шардинга и разрешения MongoDB делать все автоматически.

По мере того как ваш кластер становится больше или более занятым, эти решения становятся менее практичными. Однако для небольших кластеров вам может потребоваться больше контроля.

Использование кластера для нескольких баз данных и коллекций

MongoDB равномерно распределяет коллекции по каждому шарду в вашем кластере, что хорошо работает, если вы сохраняете однородные данные. Однако если у вас есть коллекция журналов, которая «имеет меньшее значение», чем другие данные, вы, возможно, не захотите, чтобы она занимала место на ваших более дорогих серверах. Или если у вас есть один мощный сервер, вы, возможно, захотите использовать его только для коллекции в режиме реального времени и запретить другим коллекциям использовать его. Можно создать отдельные кластеры, но также можно дать MongoDB конкретные указания относительно того, куда вы хотите поместить определенные данные.

Для этого используйте вспомогательную функцию `sh.addShardToZone()` в оболочке:

```
> sh.addShardToZone("shard0000", "high")
> // shard0001 - no zone
> // shard0002 - no zone
> // shard0003 - no zone
> sh.addShardToZone("shard0004", "low")
> sh.addShardToZone("shard0005", "low")
```

Затем можно назначить разным шардам разные коллекции. Например, если у вас есть суперважная коллекция в реальном времени:

```
> sh.updateZoneKeyRange("super.important", {"<shardKey>" : MinKey},
... {"<shardKey>" : MaxKey}, "high")
```

Этот код говорит: «для диапазона отрицательной бесконечности до бесконечности храните эту коллекцию на шардах, помеченных как "high"». Это означает, что никакие данные из коллекции *super.important* не будут храниться ни на одном другом сервере. Обратите внимание, что это не влияет на то, как распределяются другие коллекции: они все равно будут равномерно распределены между этим сервером и остальными серверами.

Вы можете выполнить аналогичную операцию, чтобы сохранить коллекцию журналов на низкокачественном сервере:

```
> sh.updateZoneKeyRange("some.logs", {"<shardKey>" : MinKey},
... {"<shardKey>" : MaxKey}, "low")
```

Коллекция журналов теперь будет равномерно распределена между *shard0004* и *shard0005*.

Присвоение коллекции зонального диапазона ключей не влияет на нее мгновенно. Это инструкция для балансировщика, утверждающая, что при запуске это жизнеспособные цели, куда нужно переместить коллекцию. Таким образом, если вся коллекция журналов находится в *shard0002* или равномерно распределена среди шардов, потребуется некоторое время, чтобы все чанки были перенесены в *shard0004* и *shard0005*.

Еще один пример. Возможно, у вас есть коллекция, которую вы не хотите размещать на шарде с зоной "high", но вам не важно, на какой шард она отправится. Можно связать все шарды с невысокой производительностью с зоной, чтобы создать новую группировку. У шардов может быть столько зон, сколько вам нужно:

```
> sh.addShardToZone("shard0001", "whatever")
> sh.addShardToZone("shard0002", "whatever")
> sh.addShardToZone("shard0003", "whatever")
> sh.addShardToZone("shard0004", "whatever")
> sh.addShardToZone("shard0005", "whatever")
```

Теперь можно указать, что вы хотите, чтобы эта коллекция (назовем ее *normal.coll*) была распределена по следующим пяти шардам:

```
> sh.updateZoneKeyRange("normal.coll", {"<shardKey>" : MinKey},
... {"<shardKey>" : MaxKey}, "whatever")
```

Нельзя назначать коллекции динамически, т. е. нельзя сказать, что «когда коллекция будет создана, случайным образом поместите ее в шард». Однако можно использовать задание cron, которое может сделать это за вас.

Если вы ошиблись или передумали, можно удалить шард из зоны с помощью функции `sh.removeShardFromZone()`:

```
> sh.removeShardFromZone("shard0005", "whatever")
```

Если вы удалите все шарды из зон, описанных диапазоном ключей (например, если вы удалите *shard0000* из зоны `"high"`), балансировщик не будет распределять данные, потому что в списке нет действительных местоположений. Все данные будут по-прежнему доступны для чтения и записи; они просто не смогут мигрировать, пока вы не измените свои теги или диапазоны тегов.

Чтобы удалить диапазон ключей из зоны, используйте функцию `sh.removeRangeFromZone()`. Ниже приведен пример. Указанный диапазон должен точно соответствовать диапазону, ранее определенному для пространства имен *some.logs* и данной зоны:

```
> sh.removeRangeFromZone("some.logs", {"<shardKey>" : MinKey},
... {"<shardKey>" : MaxKey})
```

Ручной шардинг

Иногда, в случае сложных требований или особых ситуаций, вы, возможно, предпочтете иметь полный контроль над тем, где какие данные распределяются. Вы можете отключить балансировщик, если не хотите, чтобы данные распространялись автоматически, и использовать команду `moveChunk`, чтобы распределять данные вручную.

Чтобы отключить балансировщик, подключитесь к процессу *mongos* (подойдет любой процесс *mongos*) с помощью оболочки *mongo* и отключите балансировщика с помощью функции `sh.stopBalancer()`:

```
> sh.stopBalancer()
```

Если в данный момент выполняется миграция, эта настройка не вступит в силу до тех пор, пока она не завершится. Однако после завершения всех миграций балансировщик перестанет перемещать данные. Чтобы убедиться, что после отключения миграция не выполняется, введите в оболочке *mongo* следующее:

```
> use config
> while(sh.isBalancerRunning()) {
... print("waiting...");
... sleep(1000);
... }
```

Как только балансировщик будет выключен, вы сможете перемещать данные вручную (при необходимости). Для начала выясните, где какие чанки находятся, взглянув на коллекцию *config.chunks*:

```
> db.chunks.find()
```

Теперь используйте команду `moveChunk` для переноса чанков на другие шарды. Укажите нижнюю границу чанка, которую нужно перенести, и имя шарда, в который вы хотите переместить чанк:

```
> sh.moveChunk(
... "test.manual.stuff",
... {user_id: NumberLong("-1844674407370955160")},
... "test-rs1")
```

Однако если вы не находитесь в исключительной ситуации, вы должны использовать автоматический шардинг, а не делать это вручную. Если в результате вы получите «горячую точку» на шарде, которую вы не ожидали увидеть, большая часть ваших данных может оказаться на этом шарде.

В частности, не совмещайте настройку необычных распределений вручную с запуском балансировщика. Если балансировщик обнаружит неравномерное количество чанков, он просто перетасует все, что вы делали, чтобы коллекция снова была равномерно сбалансирована. Если вам нужно неравномерное распределение чанков, используйте метод зонального шардинга, описанный в разделе «Использование кластера для нескольких баз данных и коллекций».

Глава 17

Администрирование шардинга

Как и в случае с наборами реплик, у вас есть несколько опций для администрирования разделенных кластеров.

Один из вариантов – ручное администрирование. В наши дни становится все более распространенным использование таких инструментов, как Ops Manager и Cloud Manager, а также облачного подхода «база данных как сервис», например Atlas, для администрирования всех кластеров. В этой главе мы покажем, как вручную управлять разделенным кластером, включая:

- проверку состояния кластера: кто его члены, где хранятся данные и какие соединения открыты;
- добавление, удаление и изменение членов кластера;
- администрирование перемещения данных и перемещение данных вручную.

Просмотр текущего состояния

Существует несколько вспомогательных функций, позволяющих выяснить, где какие данные находятся, где находятся шарды и что делает кластер.

Получение сводки с помощью функции sh.status()

Функция `sh.status()` предоставляет обзор шардов, баз данных и коллекций. Если у вас небольшое количество чанков, она также выведет разбивку, показывающую, где какие чанки находятся. В противном случае она просто выдаст ключ шардинга коллекции и сообщит, сколько чанков находится на каждом шарде:

```
> sh.status()
--- Sharding Status ---
```

```
sharding version: {
    "_id": 1,
    "minCompatibleVersion": 5,
    "currentVersion": 6,
    "clusterId": ObjectId("5bdf51ecf8c192ed922f3160")
}
shards:
    {   "_id": "shard01",
        "host": "shard01/localhost:27018,localhost:27019,localhost:27020",
        "state": 1
    }
    {   "_id": "shard02",
        "host": "shard02/localhost:27021,localhost:27022,localhost:27023",
        "state": 1
    }
    {    "_id": "shard03",
        "host": "shard03/localhost:27024,localhost:27025,localhost:27026",
        "state": 1
    }
active mongoses:
    "4.0.3": 1
autosplit:
    Currently enabled: yes
balancer:
    Currently enabled: yes
    Currently running: no
    Failed balancer rounds in last 5 attempts: 0
    Migration Results for the last 24 hours:
        6 : Success
    databases:
        { "_id": "config", "primary": "config", "partitioned": true }
          config.system.sessions
            shard key: { "_id": 1 }
            unique: false
            balancing: true
            chunks:
               shard01     1
               { "_id": { "$minKey": 1 } } -->>
               { "_id": { "$maxKey": 1 } } on : shard01 Timestamp(1,0)
        { "_id": "video","primary": "shard02","partitioned": true,
          "version":
               { "uuid": UUID("3d83d8b8-9260-4a6f-8d28-c3732d40d961"),
                 "lastMod": 1 } }
          video.movies
```

```
shard key: {"imdbId": "hashed"}
unique: false
balancing: true
chunks:
   shard01 3
   shard02 4
   shard03 3
   { "imdbId": { "$minKey": 1} } -->>
       { "imdbId": NumberLong("-7262221363006655132")} on :
       shard01 Timestamp(2,0)
   { "imdbId": NumberLong("-7262221363006655132") } -->>
       { "imdbId": NumberLong("-5315530662268120007")} on :
       shard03 Timestamp(3,0)
   { "imdbId": NumberLong("-5315530662268120007") } -->>
       { "imdbId": NumberLong("-3362204802044524341")} on :
       shard03 Timestamp(4,0)
   { "imdbId": NumberLong("-3362204802044524341") } -->>
       { "imdbId": NumberLong("-1412311662519947087")} on :
       shard01 Timestamp(5,0)
   { "imdbId": NumberLong("-1412311662519947087") } -->>
       { "imdbId": NumberLong("524277486033652998")} on :
       shard01 Timestamp(6,0)
   { "imdbId": NumberLong("524277486033652998") } -->>
       { "imdbId": NumberLong("2484315172280977547")} on :
       shard03 Timestamp(7,0)
   { "imdbId": NumberLong("2484315172280977547") } -->>
       { "imdbId": NumberLong("4436141279217488250")} on :
       shard02 Timestamp(7,1)
   { "imdbId": NumberLong("4436141279217488250") } -->>
       { "imdbId": NumberLong("6386258634539951337")} on :
       shard02 Timestamp(1,7)
   { "imdbId": NumberLong("6386258634539951337") } -->>
       { "imdbId": NumberLong("8345072417171006784")} on :
       shard02 Timestamp(1,8)
   { "imdbId": NumberLong("8345072417171006784") } -->>
       { "imdbId": { "$maxKey":1} } on:
       shard02 Timestamp(1,9)
```

Если чанки гораздо больше, эта функция резюмирует статистику по ним, вместо того чтобы выводить каждую. Чтобы увидеть все чанки, выполните функцию с параметром `true`: `sh.status(true)` (`true` велит функции `sh.status()` быть многословной).

Вся информация, которую показывает функция, берется из вашей базы данных *config*.

Просмотр информации о конфигурации

Вся информация о конфигурации вашего кластера хранится в коллекциях в базе данных *config* на конфигурационных серверах. У оболочки имеется несколько вспомогательных функций для предоставления этой информации более читабельным способом. Однако вы всегда можете сделать запрос к базе данных *config* напрямую, чтобы получить метаданные о вашем кластере.

Никогда не подключайтесь напрямую к своим конфигурационным серверам. Не нужно рисковать, чтобы случайно изменить или удалить данные сервера. Вместо этого подключитесь к процессу *mongos* и используйте базу данных *config*, чтобы увидеть ее данные, как в случае с любой другой базой данных:

```
> use config
```

Если вы манипулируете этими данными через процесс *mongos* (вместо того чтобы подключаться к конфигурационным серверам напрямую), *mongos* обеспечит синхронизацию всех ваших серверов и предотвратит различные опасные действия, такие как случайное удаление базы данных *config*.

В общем, вы не должны напрямую изменять какие-либо данные в базе данных *config* (за исключением случаев, отмеченных в последующих разделах). Если вы что-то измените, вам, как правило, придется перезапустить все ваши серверы *mongos*, чтобы увидеть эффект.

В базе данных *config* есть несколько коллекций. В этом разделе описывается, что содержится в каждой из них и как это можно использовать.

config.shards

Коллекция `shards` отслеживает все шарды в кластере. Типичный документ из этой коллекции может выглядеть примерно так:

```
> db.shards.find()
{
    "_id": "shard01",
    "host": "shard01/localhost:27018,localhost:27019,localhost:27020",
    "state": 1
}
{
    "_id": "shard02",
    "host": "shard02/localhost:27021,localhost:27022,localhost:27023",
    "state": 1
```

```
}
{
    "_id": "shard03",
    "host": "shard03/localhost:27024,localhost:27025,localhost:27026",
    "state": 1
}
```

`"_id"` шарда выбирается из имени набора реплик, поэтому каждый набор реплик в вашем кластере должен иметь уникальное имя.

Когда вы обновляете конфигурацию набора реплик (например, добавляете или удаляете членов), поле `"host"` будет обновляться автоматически.

config.databases

Коллекция *databases* отслеживает все базы данных, разделенные и нет, о которых известно кластеру:

```
> db.databases.find()
{ "_id" : "video", "primary" : "shard02", "partitioned" : true,
    "version" : { "uuid" : UUID("3d83d8b8-9260-4a6f-8d28-c3732d40d961"),
    "lastMod" : 1 } }
```

Если для базы данных была выполнена команда `enableSharding`, параметр `"partitioned"` будет иметь значение `true`. `"primary"` – «основное место дислокации» базы данных. По умолчанию все новые коллекции в этой базе данных будут создаваться на первичном шарде базы данных.

config.collections

Коллекция *collections* отслеживает все разделенные коллекции (неразделенные коллекции не показаны). Типичный документ выглядит примерно так:

```
> db.collections.find().pretty()
{
    "_id": "config.system.sessions",
    "lastmodEpoch": ObjectId("5bdf53122ad9c6907510c22d"),
    "lastmod": ISODate("1970-02-19T17:02:47.296Z"),
    "dropped": false,
    "key": {
        "_id": 1
    },
    "unique": false,
    "uuid": UUID("7584e4cd-fac4-4305-a9d4-bd73e93621bf")
}
{
    "_id": "video.movies",
```

```
    "lastmodEpoch": ObjectId("5bdf72c021b6e3be02fabe0c"),
    "lastmod": ISODate("1970-02-19T17:02:47.305Z"),
    "dropped": false,
    "key": {
        "imdbId": "hashed"
    },
    "unique": false,
    "uuid": UUID("e6580ffa-fcd3-418f-aa1a-0dfb71bc1c41")
}
```

Важные поля:

`"_id"`
: Пространство имен коллекции.

`"key"`
: Ключ шардинга. В данном случае это хешированный ключ на `"imdbId"`.

`"unique"`
: Указывает на то, что ключ шардинга не является уникальным индексом. По умолчанию он не является уникальным.

config.chunks

Коллекция *chunks* ведет учет каждому чанку во всех коллекциях. Типичный документ из этой коллекции выглядит примерно так:

```
> db.chunks.find().skip(1).limit(1).pretty()
{
    "_id": "video.movies-imdbId_MinKey",
    "lastmod": Timestamp(2, 0),
    "lastmodEpoch": ObjectId("5bdf72c021b6e3be02fabe0c"),
    "ns": "video.movies",
    "min": {
        "imdbId": { "$minKey": 1 }
    },
    "max": {
        "imdbId": NumberLong("-7262221363006655132")
    },
    "shard": "shard01",
    "history": [
        {
            "validAfter": Timestamp(1541370579, 3096),
            "shard": "shard01"
        }
    ]
}
```

Наиболее полезные поля:

`"_id"`
: Уникальный идентификатор чанков. Обычно это пространство имен, ключ шардинга и нижняя граница чанка.

`"ns"`
: Коллекция, к которой относится этот чанк.

`"min"`
: Наименьшее значение в диапазоне чанков (включительно).

`"max"`
: Все значения в чанке меньше этого значения.

`"shard"`
: На каком шарде находится чанк.

Поле `"lastmod"` отслеживает версионирование чанка. Например, если бы чанк `"video.movies-imdbId_MinKey"` был разделен на два, нам бы понадобился способ отличить новые более мелкие чанки `"video.movies-imdbId_MinKey"` от их предыдущего воплощения в виде единого чанка. Таким образом, первый компонент значения `Timestamp` отражает количество раз, когда чанк был перенесен в новый шард. Второй компонент этого значения отражает количество разбиений. Поле `lastmodEpoch` определяет эпоху создания коллекции. Оно используется для разграничения запросов на одно и то же имя коллекции в тех случаях, когда коллекция была удалена и тотчас же воссоздана.

Функция `sh.status()` использует коллекцию *config.chunks* для сбора большей части информации.

config.changelog

Коллекция *changelog* полезна для отслеживания того, что делает кластер, поскольку она записывает все произошедшие расщепления и миграции.

Расщепления записываются в документе, который выглядит следующим образом:

```
> db.changelog.find({what: "split"}).pretty()
{
    "_id": "router1-2018-11-05T09:58:58.915-0500-5be05ab2f8c192ed922ffbe7",
    "server": "bob",
    "clientAddr": "127.0.0.1:64621",
    "time": ISODate("2018-11-05T14:58:58.915Z"),
    "what": "split",
    "ns": "video.movies",
    "details": {
```

```
            "before": {
                "min": {
                    "imdbId": NumberLong("2484315172280977547")
                },
                "max": {
                    "imdbId": NumberLong("4436141279217488250")
                },
                "lastmod": Timestamp(9,1),
                "lastmodEpoch": ObjectId("5bdf72c021b6e3be02fabe0c")
            },
            "left": {
                "min": {
                    "imdbId": NumberLong("2484315172280977547")
                },
                "max": {
                    "imdbId": NumberLong("3459137475094092005")
                },
                "lastmod": Timestamp(9,2),
                "lastmodEpoch": ObjectId("5bdf72c021b6e3be02fabe0c")
            },
            "right": {
                "min": {
                    "imdbId": NumberLong("3459137475094092005")
                },
                "max": {
                    "imdbId": NumberLong("4436141279217488250")
                },
                "lastmod": Timestamp(9,3),
                "lastmodEpoch": ObjectId("5bdf72c021b6e3be02fabe0c")
            }
        }
    }
```

Поле `"details"` предоставляет информацию о том, как выглядел исходный документ и на что он был разбит.

В этом выводе показано, как выглядит разбиение первого чанка коллекции. Обратите внимание, что второй компонент `"lastmod"` для каждого нового чанка был обновлен, поэтому значениями являются `Timestamp(9, 2)` и `Timestamp(9, 3)` соответственно.

Миграции немного сложнее и фактически создают четыре отдельных документа журнала операций: один отмечает начало миграции, второй для шарда «из», третий для шарда «в» и четвертый для фиксации, которая происходит после окончания миграции. Средние два документа представляют интерес, поскольку они дают представление о том, сколько времени

занимал каждый шаг процесса. Это может дать вам представление о том, является ли диск, сеть или что-либо еще причиной проблем при переносе.

Например, документ, созданный шардом «из», выглядит следующим образом:

```
> db.changelog.findOne({what: "moveChunk.to"})
{
    "_id": "router1-2018-11-04T17:29:39.702-0500-5bdf72d32ad9c69075112f08",
    "server": "bob",
    "clientAddr": "",
    "time": ISODate("2018-11-04T22:29:39.702Z"),
    "what": "moveChunk.to",
    "ns": "video.movies",
    "details": {
        "min": {
            "imdbId": {
                "$minKey": 1
            }
        },
        "max": {
            "imdbId": NumberLong("-7262221363006655132")
        },
        "step 1 of 6": 965,
        "step 2 of 6": 608,
        "step 3 of 6": 15424,
        "step 4 of 6": 0,
        "step 5 of 6": 72,
        "step 6 of 6": 258,
        "note": "success"
    }
}
```

Каждый из шагов, перечисленных в поле `"details"`, синхронизирован, и сообщения типа `"step N of N"` показывают, сколько времени занял каждый шаг в миллисекундах.

Когда шард «из» получает команду `moveChunk` от процесса *mongos*, он:

1) проверяет параметры команды;
2) подтверждает с помощью конфигурационных серверов, что может установить распределенную блокировку для миграции;
3) пытается связаться с шардом «из»;
4) копирует данные. Это называется «критическим разделом»;
5) договаривается с шардом «в» и конфигурационными серверами для подтверждения миграции.

Обратите внимание, что шарды «в» и «из» должны находиться в тесном взаимодействии, начиная с `"step 4 of 6"`: шарды напрямую общаются друг с другом и конфигурационным сервером для выполнения миграции. Если сервер «из» имеет слабое сетевое подключение на последних этапах, он может оказаться в состоянии, когда не сможет отменить миграцию и двигаться дальше. В этом случае *mongod* отключится.

Документ из журнала изменений шарда «в» похож на файл шарда «из», но этапы несколько отличаются. Вот как он выглядит:

```
> db.changelog.find({what: "moveChunk.from","details.max.imdbId":
    NumberLong("-7262221363006655132")}).pretty()
    {
    "_id": "router1-2018-11-04T17:29:39.753-0500-5bdf72d321b6e3be02fabf0b",
    "server": "bob",
    "clientAddr": "127.0.0.1:64743",
    "time": ISODate("2018-11-04T22:29:39.753Z"),
    "what": "moveChunk.from",
    "ns": "video.movies",
    "details": {
        "min": {
            "imdbId": {
                "$minKey": 1
            }
        },
        "max": {
            "imdbId": NumberLong("-7262221363006655132")
        },
        "step 1 of 6": 0,
        "step 2 of 6": 4,
        "step 3 of 6": 191,
        "step 4 of 6": 17000,
        "step 5 of 6": 341,
        "step 6 of 6": 39,
        "to": "shard01",
        "from": "shard02",
        "note": "success"
    }
}
```

Когда шард «в» получает команду от шарда «из», он:

1) переносит индексы. Если на этом сервере никогда не было чанков из перенесенной коллекции, ему нужно знать, какие поля индексируются. Если это не первый раз, когда чанк из этой коллекции перемещается в этот сервер, это пустая операция;

2) удаляет любые существующие данные в диапазоне чанка. После неудачной миграции или процедуры восстановления могут остаться данные, которые мы не хотим мешать с текущими данными;
3) копирует все документы в чанки на сервер «в»;
4) повторяет все операции, которые произошли с этими документами во время копирования (на сервере «до»);
5) ждет, пока сервер «в» реплицирует недавно перенесенные данные на большинство серверов;
6) фиксирует миграцию, изменяя метаданные чанка, чтобы сообщить, что она находится на сервере «в».

config.settings

Эта коллекция содержит документы, представляющие текущие настройки балансировщика и размер чанка. Изменяя документы в этой коллекции, вы можете включать или выключать балансировщика или изменять размер чанка. Обратите внимание, что вы всегда должны подключаться к процессу *mongos*, а не напрямую к конфигурационным серверам, чтобы менять значения в этой коллекции.

Отслеживание сетевых подключений

Между компонентами кластера существует множество соединений. Этот раздел охватывает информацию, касающуюся шардинга (*см.* главу 24, где приводится дополнительная информация о работе в сети).

Получение статистики о соединениях

Команда `connPoolStats` возвращает информацию об открытых исходящих соединениях из текущего экземпляра базы данных с другими членами разделенного кластера или набора реплик.

Чтобы избежать вмешательства в любые выполняемые операции, эта команда не принимает никаких блокировок. Таким образом, количество может немного меняться, по мере того как она собирает информацию, что приводит к небольшим различиям между подсчетами соединений между хостами и пулами:

```
> db.adminCommand({ "connPoolStats": 1})
{
    "numClientConnections": 10,
    "numAScopedConnections": 0,
    "totalInUse": 0,
    "totalAvailable": 13,
    "totalCreated": 86,
```

```
"totalRefreshing": 0,
"pools": {
    "NetworkInterfaceTL-TaskExecutorPool-0": {
        "poolInUse": 0,
        "poolAvailable": 2,
        "poolCreated": 2,
        "poolRefreshing": 0,
        "localhost:27027": {
            "inUse": 0,
            "available": 1,
            "created": 1,
            "refreshing": 0
        },
        "localhost:27019": {
            "inUse": 0,
            "available": 1,
            "created": 1,
            "refreshing": 0
        }
    },
    "NetworkInterfaceTL-ShardRegistry": {
        "poolInUse": 0,
        "poolAvailable": 1,
        "poolCreated": 13,
        "poolRefreshing": 0,
        "localhost:27027": {
            "inUse": 0,
            "available": 1,
            "created": 13,
            "refreshing": 0
        }
    },
    "global": {
        "poolInUse": 0,
        "poolAvailable": 10,
        "poolCreated": 71,
        "poolRefreshing": 0,
        "localhost:27026": {
            "inUse": 0,
            "available": 1,
            "created": 8,
            "refreshing": 0
        },
        "localhost:27027": {
            "inUse": 0,
```

```
        "available": 1,
        "created": 1,
        "refreshing": 0
    },
    "localhost:27023": {
        "inUse": 0,
        "available": 1,
        "created": 7,
        "refreshing": 0
    },
    "localhost:27024": {
        "inUse": 0,
        "available": 1,
        "created": 6,
        "refreshing": 0
    },
    "localhost:27022": {
        "inUse": 0,
        "available": 1,
        "created": 9,
        "refreshing": 0
    },
    "localhost:27019": {
        "inUse": 0,
        "available": 1,
        "created": 8,
        "refreshing": 0
    },
    "localhost:27021": {
        "inUse": 0,
        "available": 1,
        "created": 8,
        "refreshing": 0
    },
    "localhost:27025": {
        "inUse": 0,
        "available": 1,
        "created": 9,
        "refreshing": 0
    },
    "localhost:27020": {
        "inUse": 0,
        "available": 1,
        "created": 8,
        "refreshing": 0
```

```
            },
            "localhost:27018": {
                "inUse": 0,
                "available": 1,
                "created": 7,
                "refreshing": 0
            }
        }
    },
    "hosts": {
        "localhost:27026": {
            "inUse": 0,
            "available": 1,
            "created": 8,
            "refreshing": 0
        },
        "localhost:27027": {
            "inUse": 0,
            "available": 3,
            "created": 15,
            "refreshing": 0
        },
        "localhost:27023": {
            "inUse": 0,
            "available": 1,
            "created": 7,
            "refreshing": 0
        },
        "localhost:27024": {
            "inUse": 0,
            "available": 1,
            "created": 6,
            "refreshing": 0
        },
        "localhost:27022": {
            "inUse": 0,
            "available": 1,
            "refreshing": 0
        },
        "localhost:27019": {
            "inUse": 0,
            "available": 2,
            "created": 9,
            "refreshing": 0
        },
```

```
        "localhost:27021": {
            "inUse": 0,
            "available": 1,
            "created": 8,
            "refreshing": 0
        },
        "localhost:27025": {
            "inUse": 0,
            "available": 1,
            "created": 9,
            "refreshing": 0
        },
        "localhost:27020": {
            "inUse": 0,
            "available": 1,
            "created": 8,
            "refreshing": 0
        },
        "localhost:27018": {
            "inUse": 0,
            "available": 1,
            "created": 7,
            "refreshing": 0
        }
    },
    "replicaSets": {
        "shard02": {
            "hosts": [
                {
                    "addr": "localhost:27021",
                    "ok": true,
                    "ismaster": true,
                    "hidden": false,
                    "secondary": false,
                    "pingTimeMillis": 0
                },
                {
                    "addr": "localhost:27022",
                    "ok": true,
                    "ismaster": false,
                    "hidden": false,
                    "secondary": true,
                    "pingTimeMillis": 0
                },
```

```
            {
                "addr": "localhost:27023",
                "ok": true,
                "ismaster": false,
                "hidden": false,
                "secondary": true,
                "pingTimeMillis": 0
            }
        ]
    },
    "shard03": {
        "hosts": [
            {
                "addr": "localhost:27024",
                "ok": true,
                "ismaster": false,
                "hidden": false,
                "secondary": true,
                "pingTimeMillis": 0
            },
            {
                "addr": "localhost:27025",
                "ok": true,
                "ismaster": true,
                "hidden": false,
                "secondary": false,
                "pingTimeMillis": 0
            },
            {
                "addr": "localhost:27026",
                "ok": true,
                "ismaster": false,
                "hidden": false,
                "secondary": true,
                "pingTimeMillis": 0
            }
        ]
    },
    "configRepl": {
        "hosts": [
            {
                "addr": "localhost:27027",
                "ok": true,
                "ismaster": true,
                "hidden": false,
```

```
                    "secondary": false,
                    "pingTimeMillis": 0
                }
            ]
        },
        "shard01": {
            "hosts": [
                {
                    "addr": "localhost:27018",
                    "ok": true,
                    "ismaster": false,
                    "hidden": false,
                    "secondary": true,
                    "pingTimeMillis": 0
                },
                {
                    "addr": "localhost:27019",
                    "ok": true,
                    "ismaster": true,
                    "hidden": false,
                    "secondary": false,
                    "pingTimeMillis": 0
                },
                {
                    "addr": "localhost:27020",
                    "ok": true,
                    "ismaster": false,
                    "hidden": false,
                    "secondary": true,
                    "pingTimeMillis": 0
                }
            ]
        }
    },
    "ok": 1,
    "operationTime": Timestamp(1541440424, 1),
    "$clusterTime": {
        "clusterTime": Timestamp(1541440424, 1),
        "signature": {
            "hash": BinData(0,
            "AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
            "keyId": NumberLong(0)
        }
    }
}
```

В этом выводе:

- `"totalAvailable"` показывает общее число доступных исходящих соединений из текущего экземпляра *mongod/mongos* с другими членами разделенного кластера или набора реплик;
- `"totalCreated"` сообщает общее число исходящих соединений, когда-либо созданных текущим экземпляром *mongod/mongos*, с другими членами разделенного кластера или набора реплик;
- `"totalInUse"` предоставляет общее число исходящих подключений из текущего экземпляра *mongod/mongos* с другими членами разделенного кластера или набора реплик, которые используются в данный момент;
- `"totalRefreshing"` отображает общее число исходящих подключений из текущего экземпляра *mongod/mongos* с другими членами разделенного кластера или набора реплик, которые в данный момент обновляются;
- `"numClientConnections"` определяет количество активных и сохраненных исходящих синхронных соединений из текущего экземпляра *mongod/mongos* с другими членами разделенного кластера или набора реплик. Они представляют собой подмножество соединений, о которых сообщают `"totalAvailable"`, `"totalCreated"` и `"totalInUse"`;
- `"numAScopedConnection"` сообщает о количестве активных и сохраненных исходящих синхронных соединений из текущего экземпляра *mongod/mongos* с другими членами разделенного кластера или набора реплик. Они представляют собой подмножество соединений, о которых сообщают `"totalAvailable"`, `"totalCreated"` и `"totalInUse"`;
- `"pools"` показывает статистику соединения (используется/доступно/создано/обновляется), сгруппированную по пулам соединений. У *mongod* или *mongos* есть два разных семейства пулов исходящих соединений:
  - пулы на базе DBClient («путь к операции записи», определяемый именем поля `"global"` в документе `"pools"`);
  - пулы на базе NetworkInterfaceTL («путь к операции чтения»);
- `"hosts"` показывает статистику по соединению (используется/доступно/создано/обновляется), сгруппированную по хостам. Он сообщает о соединениях между текущим экземпляром *mongod/mongos* и каждым членом разделенного кластера или набора реплик.

В выходных данных команды `connPoolStats` вы можете увидеть соединения с другими шардами. Это означает, что шарды подсоединяются к другим шардам для переноса данных. Первичный узел одного шарда будет

подключаться напрямую к первичному узлу другого шарда и «высасывать» его данные.

Когда происходит миграция, шард устанавливает `ReplicaSetMonitor` (процесс, который контролирует работоспособность набора реплик), чтобы отслеживать работоспособность шарда на другой стороне миграции. *mongod* никогда не разрушает этот процесс, поэтому в журнале одного набора реплик можно увидеть сообщения о членах другого набора. Это абсолютно нормально и не должно влиять на ваше приложение.

Ограничение числа соединений

Когда клиент подключается к процессу *mongos*, тот создает соединение хотя бы с одним шардом для передачи запроса клиента. Таким образом, каждое клиентское соединение в процессе *mongos* дает по крайней мере одно исходящее соединение от *mongos* к шардам.

Если у вас много процессов *mongos*, они могут создать больше соединений, чем ваши шарды способны обработать: по умолчанию процесс *mongos* будет принимать до 65 536 соединений (столько же, что и *mongod*), поэтому если у вас 5 процессов *mongos* и у каждого из них по 10 000 клиентских подключений, они могут попытаться создать 50 000 подключений к шарду!

Чтобы предотвратить это, можно использовать параметр `--maxConns` в конфигурации командной строки для ограничеия числа соединений, которые может создать процесс *mongos*. Приведенную ниже формулу можно использовать для расчета максимального количества соединений от одного процесса, которые шард может обработать:

$$maxConns = maxConnsPrimary - (numMembersPerReplicaSet \times 3) - (other \times 3) / numMongosProcesses.$$

Разберем фрагменты этой формулы:

maxConnsPrimary

Максимальное число подключений на первичном узле, как правило, составляет 20 000, чтобы не перегружать шард соединениями от *mongos*.

(*numMembersPerReplicaSet* × *3*)

Первичный узел создает соединение с каждым вторичным узлом, а каждый вторичный узел создает два соединения с первичным, всего три соединения.

(*other* × *3*)

other – это число разных процессов, которые могут подключаться к вашим серверам *mongod*, например мониторинг или резервное ко-

пирование агентов, подключения напрямую к оболочке (для администрирования) или подключения к другим шардам для миграции.

numMongosProcesses

Общее число процессов *mongos* в кластере.

Обратите внимание, что параметр `--maxConns` не позволяет процессам *mongos* создавать соединений больше, чем указано. Он не делает ничего особенно полезного при достижении этого предела: он просто блокирует запросы, ожидая, когда соединения будут «освобождены». Таким образом, вы должны запретить вашему приложению использовать такое количество соединений, особенно по мере роста числа процессов *mongos*.

Когда экземпляр MongoDB корректно завершает работу, он закрывает все соединения, перед тем как остановиться. Члены, которые были подключены к нему, незамедлительно получат ошибки сокета на этих соединениях и смогут обновить их. Однако если экземпляр MongoDB внезапно становится недоступен из-за потери питания, сбоя или проблем с сетью, он, вероятно, не сможет правильно закрыть все свои сокеты. В этом случае у других серверов в кластере может сложиться впечатление, что их соединение исправно, пока они не попытаются выполнить какую-нибудь операцию. В этот момент они получат сообщение об ошибке и обновят соединение (если член снова будет доступным в тот момент).

Это быстрый процесс, когда соединений всего несколько. Тем не менее если их тысячи и их нужно обновлять по одному, вы можете получить большое количество ошибок, потому что каждое соединение с недоступным членом должно быть проверено, определено как плохое и восстановлено. Это не очень хороший способ предотвратить подобное, не говоря уже о перезапуске процессов, которые увязли в шквале повторных подключений.

Администрирование сервера

По мере роста вашего кластера вам нужно будет добавить емкости или изменить настройки. В этом разделе описывается, как добавлять и удалять серверы в вашем кластере.

Добавление серверов

Можно добавлять новые процессы *mongos* в любое время. Убедитесь, что их параметр `--configdb` указывает правильный набор конфигурационных серверов и они должны быть доступны незамедлительно, чтобы клиенты могли к ним подключиться.

Чтобы добавить новые шарды, используйте команду `addShard`, как показано в главе 15.

Смена серверов в шарде

При использовании разделенного кластера у вас может появиться желание изменить серверы на отдельных шардах. Чтобы изменить членство в шарде, подключитесь напрямую к первичному узлу шарда (не через *mongos*) и выполните повторное конфигурирование набора реплик. Изменения будут приняты, и коллекция *config.shards* обновится автоматически. Не изменяйте ее вручную.

Единственное исключение – если вы запускаете кластер с автономными серверами в качестве шардов, а не наборов реплик.

Меняем шард с автономного сервера на набор реплик

Самый простой способ сделать это – добавить новый пустой шард набора реплик, а затем удалить автономный шард (как описано в следующем разделе). Миграции позаботятся о переносе ваших данных на новый сервер.

Удаление шарда

Как правило, шарды не стоит удалять из кластера. Если вы регулярно добавляете и удаляете их, вы подвергаете систему гораздо большему стрессу, чем необходимо. Если вы добавляете слишком много шардов, лучше позволить вашей системе расти в них, вместо того чтобы удалять их и добавлять позже. Однако при необходимости шарды можно удалить.

Сначала убедитесь, что балансировщик включен. Перед балансировщиком будет поставлена задача переместить все данные на шарде, который вы хотите удалить, в другие шарды в ходе процесса, который носит название *эвакуация*.

Чтобы начать эвакуацию, выполните команду `removeShard`. Эта команда берет имя шарда и «эвакуирует» все чанки, находящиеся на этом шарде, в другие шарды:

```
> db.adminCommand({
    "removeShard": "shard03"
})
{
    "msg": "draining started successfully",
    "state": "started",
    "shard": "shard03",
    "note": "you need to drop or movePrimary these databases",
    "dbsToMove": [],
    "ok": 1,
    "operationTime": Timestamp(1541450091,2),
    "$clusterTime": {
        "clusterTime": Timestamp(1541450091,2),
        "signature": {
```

```
            "hash": BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
            "keyId": NumberLong(0)
        }
    }
}
```

Эвакуация может занять много времени, если нужно переместить большое количество чанков или же если чанки большие. Если у вас есть большие чанки, т. н. jumbo-чанки (см. раздел «Неразделимые чанки»), вам может потребоваться временно увеличить размер чанков, чтобы их можно было эвакуировать.

Если вы хотите следить за тем, сколько было перемещено, снова выполните команду `removeShard`, чтобы получить текущий статус:

```
> db.adminCommand({"removeShard": "shard02"})
{
    "msg": "draining ongoing",
    "state": "ongoing",
    "remaining": {
        "chunks": NumberLong(3),
        "dbs": NumberLong(0)
    },
    "note": "you need to drop or movePrimary these databases",
    "dbsToMove": [
            "video"
        ],
    "ok": 1,
    "operationTime": Timestamp(1541450139,1),
    "$clusterTime": {
        "clusterTime": Timestamp(1541450139,1),
        "signature": {
            "hash": BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
            «keyId»: NumberLong(0)
        }
    }
}
```

Вы можете выполнять команду `removeShard` столько раз, сколько захотите.

Возможно, чанки нужно будет разбить, чтобы переместить их, поэтому вы можете увидеть, что число чанков в системе растет во время эвакуации. Например, предположим, у нас есть кластер из пяти шардов со следующими распределениями чанков:

```
test-rs0    10
test-rs1    10
test-rs2    10
```

```
test-rs3    11
test-rs4    11
```

Всего в этом кластере 52 чанка. Если мы удалим *test-rs3*, то можем получить это:

```
test-rs0    15
test-rs1    15
test-rs2    15
test-rs4    15
```

Кластер теперь насчитывает 60 чанков, 18 из которых были получены от шарда *test-rs3* (одиннадцать были там изначально, семь были созданы в ходе эвакуации).

После перемещения всех чанков, если еще есть базы данных, в которых удаленный шард является первичным, их необходимо удалить, прежде чем можно будет удалить шард. У каждой базы данных в разделенном кластере есть первичный шард. Если шард, который вы хотите удалить, также является первичным в одной из баз данных кластера, команда `removeShard` перечислит эту базу данных в поле `"dbsToMove"`. Чтобы завершить удаление шарда, вы должны либо переместить базу данных в новый шард после переноса всех данных из шарда, либо удалить ее, стерев связанные файлы данных. Вывод команды `removeShard` будет выглядеть примерно так:

```
> db.adminCommand({"removeShard": "shard02"})
{
    "msg": "draining ongoing",
    "state": "ongoing",
    "remaining": {
        "chunks": NumberLong(3),
        "dbs": NumberLong(0)
    },
    "note": "you need to drop or movePrimary these databases",
    "dbsToMove": [
        "video"
    ],
    "ok": 1,
    "operationTime": Timestamp(1541450139,1),
    "$clusterTime": {
        "clusterTime": Timestamp(1541450139,1),
        "signature": {
            "hash": BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
            "keyId": NumberLong(0)
        }
    }
```

```
}
```

Чтобы завершить удаление, переместите перечисленные базы данных с помощью команды `movePrimary`:

```
> db.adminCommand({ "movePrimary": "video","to": "shard01"})
{
    "ok": 1,
    "operationTime": Timestamp(1541450554, 12),
    "$clusterTime": {
        "clusterTime": Timestamp(1541450554, 12),
        "signature": {
            "hash": BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
            "keyId": NumberLong(0)
        }
    }
```

Сделав это, выполните команду `removeShard` еще раз:

```
> db.adminCommand({
    "removeShard": "shard02"
})
{
    "msg": "removeshard completed successfully",
    "state": "completed",
    "shard": "shard03",
    "ok": 1,
    "operationTime": Timestamp(1541450619,2),
    "$clusterTime": {
        "clusterTime": Timestamp(1541450619,2),
        "signature": {
            "hash": BinData(0, "AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
            "keyId": NumberLong(0)
        }
    }
}
```

Это не является строго обязательным, но подтверждает, что вы завершили процесс. Если баз данных, в которых этот шард является первичным, нет, вы получите этот ответ, как только все чанки из шарда будут перенесены.

Как только вы начнете эвакуацию, встроенного способа остановить ее нет.

Балансировка данных

В целом MongoDB автоматически заботится о балансировке данных. В этом разделе описывается, как активировать и отключить автоматическую балансировку, а также как вмешаться в процесс балансировки.

Балансировщик

Выключение балансировщика является обязательным условием практически любой административной деятельности. Существует вспомогательная функция оболочки, позволяющая сделать это проще:

```
> sh.setBalancerState(false)
{
    "ok": 1,
    "operationTime": Timestamp(1541450923,2),
    "$clusterTime": {
        "clusterTime": Timestamp(1541450923,2),
        "signature": {
            "hash": BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
            "keyId": NumberLong(0)
        }
    }
}
```

Когда балансировщик будет выключен, новый раунд балансировки не начнется, но это не значит, что текущий раунд тотчас же остановится – миграция, как правило, не может остановиться в мгновение ока. Таким образом, нужно проверить коллекцию *config.locks*, чтобы выяснить, продолжается балансировка или нет:

```
> db.locks.find({"_id" : "balancer"})["state"]
0
```

`0` означает, что балансировщик выключен.

Балансировка создает нагрузку на вашу систему: шард назначения должен запросить у исходного шарда все документы в чанк и вставить их, а затем исходный шард должен удалить их. В частности, есть два обстоятельства, когда миграции могут вызывать проблемы с производительностью:

1) использование ключа хот-спота вызовет постоянную миграцию (поскольку все новые чанки будут создаваться в хот-споте). Ваша система должна иметь возможность обрабатывать поток данных, поступающих с вашего хот-спота;
2) добавление нового шарда инициирует поток миграций, когда балансировщик попытается заполнить его.

Если вы обнаружите, что миграции влияют на производительность вашего приложения, то можете запланировать окно для балансировки в коллекции *config.settings*. Запустите следующее обновление, чтобы разрешить балансировку только между 13:00 и 16:00. Сначала убедитесь, что балансировщик включен, а затем запланируйте окно:

```
> sh.setBalancerState( true )
{
    "ok": 1,
    "operationTime": Timestamp(1541451846,4),
    "$clusterTime": {
        "clusterTime": Timestamp(1541451846,4),
        "signature": {
            "hash": BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
            "keyId": NumberLong(0)
        }
    }
}
> db.settings.update(
    { _id: "balancer"},
    { $set: { activeWindow : { start : "13:00", stop : "16:00"} } },
    { upsert: true}
)
WriteResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1})
```

Если вы установили окно балансировки, внимательно следите за ним, чтобы убедиться, что процесс *mongos* действительно может поддерживать ваш кластер сбалансированным на то время, на которое вы его выделили.

Вы должны быть осторожны, если планируете сочетать ручную балансировку с автоматическим балансировщиком, поскольку автоматический балансировщик всегда определяет, что нужно перемещать, основываясь на текущем состоянии набора, не принимая во внимание историю набора. Например, предположим, что у вас есть шарды *shardA* и *shardB*, каждый из которых содержит по 500 чанков. *shardA* получает много записей, поэтому вы выключаете балансировщик и перемещаете 30 самых активных чанков в *shardB*. Если в этот момент вы снова включите балансировщик, он сразу же включится и переместит 30 чанков (возможно, 30) обратно из *shardB* в *shardA*, чтобы сбалансировать количество чанков.

Чтобы предотвратить это, перед запуском балансировщика переместите 30 чанков из *shardB* в *shardA*. Таким образом, между шардами не будет дисбаланса, и балансировщик будет рад оставить все как есть. В качестве альтернативного варианта можно выполнить тридцать разбиений чанков *shardA*, чтобы выровнять счет чанков.

Обратите внимание, что балансировщик использует только количество чанков в качестве показателя, а не размер данных. Перемещение чанков называется миграцией, и именно так MongoDB балансирует данные в вашем кластере. Таким образом, шард с несколькими большими чанками может оказаться целью миграции из шарда со множеством маленьких чанков (но с меньшим размером данных).

Изменение размера чанков

В чанке может быть от нуля до миллионов документов. Как правило, чем больше чанки, тем больше времени требуется для перехода в другой шард. В главе 14 мы использовали размер чанков в 1 МБ, чтобы можно было легко и быстро увидеть перемещение чанков.

В живой системе это вообще непрактично; MongoDB будет выполнять много ненужной работы, чтобы сохранить размер шардов в пределах нескольких мегабайтов. По умолчанию размер чанков составляет 64 МБ, что обычно обеспечивает хороший баланс между простотой миграции и миграционным оттоком.

Иногда вы можете обнаружить, что миграция занимает слишком много времени при наличии чанков размером 64 МБ. Чтобы ускорить ее, можно уменьшить размер чанков. Для этого подключитесь к процессу *mongos* через оболочку и обновите коллекцию *config.settings*:

```
> db.settings.findOne()
{
    "_id" : "chunksize",
    "value" : 64
}
> db.settings.save({"_id" : "chunksize", "value" : 32})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

Предыдущее обновление изменило бы размер ваших чанков на 32 МБ. Однако существующие чанки не будут изменены тотчас же; автоматическое разбиение происходит только при вставке или обновлении. Таким образом, если вы уменьшите размер чанков, может потребоваться время, чтобы все чанки были разделены до нового размера.

Разбиение нельзя отменить. Если вы увеличиваете размер чанков, существующие чанки увеличиваются только за счет вставки или обновления, пока не достигнут нового размера. Допустимый диапазон размеров чанков – от 1 до 1024 МБ включительно.

Обратите внимание, что это настройка для всего кластера: она влияет на все коллекции и базы данных. Таким образом, если вам нужен маленький размер чанка для одной коллекции и большой размер для другой, вам, возможно, придется пойти на компромисс (или поместить коллекции в разные кластеры).

Если MongoDB выполняет слишком много миграций или у вас большие документы, можно увеличить размер чанков.

Перемещение чанков

Как упоминалось ранее, все данные в чанке хранятся на определенном шарде. Если в этом шарде будет больше чанков, чем в других, MongoDB удалит оттуда часть чанков.

Можно перемещать чанки вручную с помощью функции `sh.moveChunk()`:

```
> sh.moveChunk("video.movies", {imdbId: 500000}, "shard02")
{ "millis" : 4079, "ok" : 1 }
```

Здесь мы перемещаем чанк, содержащий документ с ключом `"imdbId"`, значение которого равно 500 000, в шард *shard02*. Вы должны использовать ключ шардинга (в данном случае `"imdbId"`), чтобы найти, какой чанк переместить. Как правило, самый простой способ указать чанк – его нижняя граница, хотя подойдет любое значение в чанке (но не верхняя граница, поскольку ее на самом деле нет в чанке). Мы перемещаем чанк перед возвратом, поэтому запуск может занять некоторое время. Журналы – лучшее место, чтобы увидеть, что делает команда, если этот процесс занимает много времени.

Если чанк превышает максимальный размер, процесс *mongos* откажется его перемещать:

```
> sh.moveChunk("video.movies",{imdbId: NumberLong("8345072417171006784")},
   "shard02")
{
    "cause": {
        "chunkTooBig": true,
        "estimatedChunkSize": 2214960,
        "ok": 0,
        "errmsg": "chunk too big to move"
    },
    "ok": 0,
    "errmsg": "move failed"
}
```

В этом случае вы должны вручную разделить чанк, перед тем как перемещать его, используя команду `splitAt`:

```
> db.chunks.find({ns: "video.movies","min.imdbId":
  NumberLong("6386258634539951337")}).pretty()
{
    "_id": "video.movies-imdbId_6386258634539951337",
    "ns": "video.movies",
    "min": {
        "imdbId": NumberLong("6386258634539951337")
    },
    "max": {
        "imdbId": NumberLong("8345072417171006784")
    },
    "shard": "shard02",
    "lastmod": Timestamp(1,9),
    "lastmodEpoch": ObjectId("5bdf72c021b6e3be02fabe0c"),
    "history": [
        {
            "validAfter": Timestamp(1541370559, 4),
            "shard": "shard02"
        }
    ]
}
> sh.splitAt("video.movies",{ "imdbId":
  NumberLong("7000000000000000000")})
{
    "ok": 1,
    "operationTime": Timestamp(1541453304,1),
    "$clusterTime": {
        "clusterTime": Timestamp(1541453306,5),
        "signature": {
            "hash": BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
            "keyId": NumberLong(0)
        }
    }
}
> db.chunks.find({ns: "video.movies", "min.imdbId":
    NumberLong("6386258634539951337")
}).pretty()
    {
    "_id": "video.movies-imdbId_6386258634539951337",
    "lastmod": Timestamp(15,2),
    "lastmodEpoch": ObjectId("5bdf72c021b6e3be02fabe0c"),
    "ns": "video.movies",
    "min": {
        "imdbId": NumberLong("6386258634539951337")
```

```
        },
        "max": {
            "imdbId": NumberLong("7000000000000000000")
        },
        "shard": "shard02",
        "history": [
            {
                "validAfter": Timestamp(1541370559,4),
                "shard": "shard02"
            }
        ]
    }
```

После того как чанк будет разделен на более мелкие части, она должна стать перемещаемой. Кроме того, можно увеличить максимальный размер чанка, а затем переместить его, но нужно разбивать большие фрагменты, когда это возможно. Однако иногда чанки нельзя разбить – мы рассмотрим эту ситуацию далее[1].

Неразделимые чанки

Предположим, вы выбрали поле `"date"` в качестве ключа шардинга. Поле `"date"` в этой коллекции представляет собой строку, которая выглядит как `"год/месяц/день"`. Это означает, что процессы *mongos* могут создавать не более одного чанка в день. Какое-то время это работает нормально, пока ваше приложение не разойдется по сети и не получит в тысячу раз больше трафика, чем обычно за один день.

Чанки этого дня будут намного больше, чем в любой другой день, и к тому же совершенно неразделимыми, потому что у каждого документа одно и то же значение ключа шардинга.

Если чанк превышает максимальный размер, заданный в коллекции *config.settings*, балансировщику запрещено перемещать чанки. Эти неразделимые и неподвижные чанки называются jumbo-чанки, с которыми неудобно иметь дело.

Возьмем пример. Предположим, у вас есть три шарда: *шард1*, *шард2* и *шард3*. Если вы используете шаблон ключа хот-спота, описанный в разделе «Монотонно возрастающие ключи», все ваши записи будут идти в один шард, например *шард1*. Первичный сервер *mongod* потребует, чтобы *балансировщик* равномерно перемещал каждый новый верхний чанк между другими шардами, но единственные чанки, которые может перемещать балансировщик, – это чанки, не относящиеся к категории jumbo, поэтому он перенесет все маленькие чанки с хот-спота.

[1] В MongoDB версии 4.4 в функцию `moveChunk` планируется добавить новый параметр (`forceJumbo`), а также новую настройку конфигурации балансировщика `attemptToBalanceJumboChunks` для работы с неразделимыми чанками. Подробности см. на странице https://jira.mongodb.org/browse/SERVER-42273.

Теперь у всех шардов будет примерно одинаковое количество чанков, но размер всех чанков *шарда2* и *шарда3* будет составлять менее 64 МБ. И если создаются неразделимые и неперемещаемые чанки, все больше и больше чанков *шарда1* будут иметь размер более 64 МБ. Таким образом, *шард1* будет заполняться намного быстрее, чем два других шарда, даже притом, что количество чанков идеально сбалансировано между ними.

Таким образом, один из признаков того, что у вас проблемы с неразделимыми чанками, – это то, что размер одного шарда растет намного быстрее, чем у других. Вы также можете просмотреть вывод функции `sh.status()`, чтобы увидеть, есть ли у вас неразделимые чанки, – они будут помечены атрибутом `jumbo`:

```
> sh.status()
...
    { "x" : -7 } -->> { "x" : 5 } on : shard0001
    { "x" : 5 } -->> { "x" : 6 } on : shard0001 jumbo
    { "x" : 6 } -->> { "x" : 7 } on : shard0001 jumbo
    { "x" : 7 } -->> { "x" : 339 } on : shard0001
...
```

Вы можете использовать команду `dataSize` для проверки размеров чанков. Сначала используйте коллекцию *config.chunks*, чтобы найти диапазоны чанков:

```
> use config
> var chunks = db.chunks.find({"ns" : "acme.analytics"}).toArray()
Затем используйте эти диапазоны, чтобы найти неразделимые чанки:
> use <dbName>
> db.runCommand({"dataSize" : "<dbName.collName>",
... "keyPattern" : {"date" : 1}, // shard key
... "min" : chunks[0].min,
... "max" : chunks[0].max})
{
    "size" : 33567917,
    "numObjects" : 108942,
    "millis" : 634,
    "ok" : 1,
    "operationTime" : Timestamp(1541455552, 10),
    "$clusterTime" : {
        "clusterTime" : Timestamp(1541455552, 10),
        "signature" : {
            "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
            "keyId" : NumberLong(0)
        }
```

```
    }
}
```

Однако будьте осторожны – команда `dataSize` должна просканировать данные чанков, чтобы выяснить, насколько она велика. Если у вас есть такая возможность, сузьте область поиска, используя то, что вам известно о своих данных: были ли созданы неразделимые чанки в определенный день? Например, если 1 июля был действительно напряженным днем, ищите чанки с этим днем в их диапазоне ключей шардинга.

Если вы используете GridFS и шардинг по `"files_id"`, можете поискать в коллекции *fs.files*, чтобы найти размер файла.

Распределение неразделимых чанков

Чтобы исправить кластер, выведенный из равновесия неразделимыми чанками, нужно равномерно распределить их по шардам.

Это сложный ручной процесс, но он не должен вызывать простоев (это может привести к замедлению, поскольку вы будете переносить много данных). В последующем описании шард с неразделимыми чанками называется шардом «из». Шарды, в которые переносятся неразделимые чанки, называются шардами «в». Обратите внимание, что у вас может быть несколько шардов «из», из которых вы хотите переместить чанки. Повторите эти шаги для каждого из них.

1. Выключите балансировщик. Вам не нужно, чтобы балансировщик пытался «помочь» вам во время этого процесса:

```
> sh.setBalancerState(false)
```

2. MongoDB не позволит вам перемещать чанки больше максимального размера, поэтому временно увеличьте размер чанков. Запишите исходный размер чанков, а затем измените его на что-то большое, например 10 000. Размер чанка указывается в мегабайтах:

```
> use config
> db.settings.findOne({"_id" : "chunksize"})
{
    "_id" : "chunksize",
    "value" : 64
}
> db.settings.save({"_id" : "chunksize", "value" : 10000})
```

3. Используйте команду `moveChunk`, чтобы убрать неразделимые чанки из шарда «из».
4. Выполните команду `splitChunk` для оставшихся чанков на шарде «из», пока в нем не будет примерно столько же чанков, сколько в шардах «в».
5. Установите размер чанков в исходное значение:

```
> db.settings.save({"_id" : "chunksize", "value" : 64})
```

6. Включите балансировщик:

```
> sh.setBalancerState(true)
```

Когда балансировщик снова включится, он опять не сможет перемещать неразделимые чанки; они, по сути, удерживаются на месте по своим размерам.

Предотвращение появления неразделимых чанков

По мере того как объем хранимых данных растет, ручной процесс, описанный в предыдущем разделе, становится неприемлемым. Таким образом, если у вас проблемы с неразделимыми чанками, ваша первоочередная задача – сделать так, чтобы они не появлялись.

Чтобы избежать возникновения неразделимых чанков, измените свой ключ шардинга, дабы добавить ему избирательности. Вам нужно, чтобы почти каждый документ имел уникальное значение ключа шардинга или, по крайней мере, никогда не имел значения данных, превышающего размер куска, с одним значением ключа шарда.

Например, если вы использовали описанный выше ключ типа «год/месяц/день», его можно быстро сделать более высокоизбирательным, добавив часы, минуты и секунды. Точно так же, если вы имеете дело с чем-то низкоизбирательным, например уровень журнала, можно добавить к своему ключу шардинга второе поле с высокой избирательностью, например хеш MD5 или UUID.

Тогда вы всегда можете разделить чанки, даже если первое поле одинаково для многих документов.

Обновление конфигураций

В заключение стоит отметить, что иногда процессы *mongos* будут некорректно обновлять свою конфигурацию из конфигурационных серверов. Если вы когда-нибудь получите конфигурацию, которую не ожидали увидеть, или процесс *mongos*, похоже, устарел или не может найти данные, но вы знаете, что они там есть, используйте команду `flushRouterConfig`, чтобы очистить все кеши вручную:

```
> db.adminCommand({"flushRouterConfig" : 1})
```

Если эта команда не сработает, перезапуск всех ваших процессов *mongos* или *mongod* удалит все кешированные данные.

Часть V

Администрирование приложений

Глава 18

Смотрим, что делает ваше приложение

После того как приложение запущено, как узнать, что оно делает? В этой главе рассказывается, как выяснить, какие запросы выполняет MongoDB, сколько данных записывается, и другие подробности о том, что на самом деле она делает.

Вы узнаете, как:

- находить медленные операции и останавливать их;
- получать и интерпретировать статистику о своих коллекциях и базах данных;
- использовать инструменты командной строки, чтобы получить представление о том, что делает MongoDB.

Просмотр текущих операций

Простой способ найти медленные операции – посмотреть, что у вас запущено. Наверняка появится что-нибудь медленное, что работает уже длительное время. Гарантий нет, но это хорошее начало, чтобы увидеть, что может замедлять работу приложения.

Чтобы увидеть выполняемые операции, используйте функцию `db.currentOp()`:

```
> db.currentOp()
{
    "inprog": [
        {
            "type": "op",
            "host": "eoinbrazil-laptop-osx:27017",
            "desc": "conn3",
            "connectionId": 3,
            "client": "127.0.0.1:57181",
            "appName": "MongoDB Shell",
```

```
"clientMetadata": {
    "application": {
        "name": "MongoDB Shell"
    },
    "driver": {
        "name": "MongoDB Internal Client",
        "version": "4.2.0"
    },
    "os": {
        "type": "Darwin",
        "name": "Mac OS X",
        "architecture": "x86_64",
        "version": "18.7.0"
    }
},
"active": true,
"currentOpTime": "2019-09-03T23:25:46.380+0100",
"opid": 13594,
"lsid": {
    "id": UUID("63b7df66-ca97-41f4-a245-eba825485147"),
    "uid": BinData(0,"47DEQpj8HBSa+/TImW+5JCeuQeRkm5NMpJWZG3hSuFU=")
},
"secs_running": NumberLong(0),
"microsecs_running": NumberLong(969),
"op": "insert",
"ns": "sample_mflix.items",
"command": {
    "insert": "items",
    "ordered": false,
    "lsid": {
        "id": UUID("63b7df66-ca97-41f4-a245-eba825485147")
    },
    "$readPreference": {
        "mode": "secondaryPreferred"
    },
    "$db": "sample_mflix"
},
"numYields": 0,
"locks": {
    "ParallelBatchWriterMode": "r",
    "ReplicationStateTransition": "w",
    "Global": "w",
    "Database": "w",
    "Collection": "w"
},
```

```
            "waitingForLock": false,
            "lockStats": {
                "ParallelBatchWriterMode": {
                    "acquireCount": {
                        "r": NumberLong(4)
                    }
                },
                "ReplicationStateTransition": {
                    "acquireCount": {
                        "w": NumberLong(4)
                    }
                },
                "Global": {
                    "acquireCount": {
                        "w": NumberLong(4)
                    }
                },
                "Database": {
                    "acquireCount": {
                        "w": NumberLong(4)
                    }
                },
                "Collection": {
                    "acquireCount": {
                        "w": NumberLong(4)
                    }
                },
                "Mutex": {
                    "acquireCount": {
                        "r": NumberLong(196)
                    }
                }
            },
            "waitingForFlowControl": false,
            "flowControlStats": {
                "acquireCount": NumberLong(4)
            }
        }
    ],
    "ok": 1
}
```

Тут приведен список операций, которые выполняет база данных. Вот некоторые из наиболее важных полей в этом выводе:

`"opid"`
: Уникальный идентификатор операции. Вы можете использовать этот номер, чтобы остановить операцию (*см.* «Остановка операций»).

`"active"`
: Работает ли эта операция. Если это поле имеет значение `false`, это означает, что операция уступила или ожидает блокировки.

`"secs_running"`
: Продолжительность этой операции в секундах. Это поле можно использовать, чтобы найти запросы, которые занимают слишком много времени.

`"microsecs_running"`
: Продолжительность этой операции в микросекундах. Это поле можно использовать, чтобы найти запросы, которые занимают слишком много времени.

`"op"`
: Тип операции. Обычно это `"query"`, `"insert"`, `"update"` или `"remove"`. Обратите внимание, что команды базы данных обрабатываются как запросы.

`"desc"`
: Идентификатор клиента. Может быть связан с сообщениями в журналах. Каждое сообщение журнала, относящееся к соединению в нашем примере, будет иметь префикс `[conn3]`, поэтому вы можете использовать это для поиска в журналах соответствующей информации.

`"locks"`
: Описание типов блокировок, принятых этой операцией.

`"waitingForLock"`
: Является ли эта операция в настоящее время блокирующей, ожидая установки блокировки.

`"numYields"`
: Количество раз, которое эта операция уступила, освобождая свою блокировку, чтобы разрешить другие операции. Как правило, любая операция, которая ищет документы (запросы, обновления и удаления), может уступить. Операция уступит только в том случае, если есть другие операции, поставленные в очередь и ожидающие своей блокировки. В основном если операций в состоянии `"waitingForLock"` нет, текущие операции уступать не будут.

`"lockstats.timeAcquiringMicros"`
: Сколько времени понадобилось этой операции, чтобы установить необходимые блокировки.

Вы можете отфильтровать `currentOp`, чтобы искать только те операции, которые соответствуют определенным критериям, например операции в определенном пространстве имен или операции, которые выполнялись в течение определенного периода времени. Вы фильтруете результаты, передавая аргумент запроса:

```
> db.currentOp(
    {
        "active": true,
        "secs_running": {
            "$gt": 3
        },
        "ns": /^db1\./
    }
)
```

Вы можете запросить любое поле в `currentOp`, используя все обычные операторы запросов.

Поиск проблемных операций

Чаще всего функция `db.currentOp()` используется для поиска медленных операций. Можно использовать метод фильтрации, описанный в предыдущем разделе, чтобы найти все запросы, которые занимают больше определенного количества времени, что может указывать на отсутствие индекса или неправильную фильтрацию полей.

Иногда люди обнаруживают, что выполняются неожиданные запросы, как правило, из-за того, что на сервере приложений запущена старая версия программного обеспечения или версия с ошибками. Поле `"client"` может помочь вам отследить, откуда идут непредвиденные операции.

Остановка операций

Если вы нашли операцию, которую хотите остановить, это можно сделать, передав функции `db.killOp()` ее идентификатор:

```
> db.killOp(123)
```

Не все операции можно остановить. В целом операции можно остановить только тогда, когда они уступают, поэтому все операции обновления, поиска и удаления можно остановить, но операции, удерживающие или ожидающие блокировки, обычно остановить нельзя.

Как только вы отправите операции сообщение об остановке, в выводе функции `db.currentOp()` появится поле `"killed"`. Однако на самом деле операция не будет считаться остановленной, пока не исчезнет из списка текущих операций.

В MongoDB версии 4.0 метод `killOP` был расширен, чтобы иметь возможность работать с процессами *mongos*. Теперь он может останавливать запросы (операции чтения), которые выполняются на более чем одном шарде кластера. В предыдущих версиях это включало в себя выполнение команды остановки вручную для каждого шарда на соответствующем первичном сервере *mongod*.

Ложные срабатывания

Когда вы ищете медленные операции, то можете увидеть в списке некоторые длительные внутренние операции. MongoDB может иметь несколько длительных запросов в зависимости от ваших настроек. Наиболее распространенными являются потоки репликации (которые будут продолжать извлекать больше операций из источника синхронизации как можно дольше) и слушатель обратной записи для шардинга. Любой длительный запрос к *local.oplog.rs* можно игнорировать, равно как и любые команды слушателя (https://oreil.ly/95e3x).

Если вы остановите любую из этих операций, MongoDB просто перезапустит их. Тем не менее, как правило, делать этого не стоит. Остановка потока репликации на короткое время прекратит репликацию, а остановка слушателя обратной записи может привести к тому, что процессы *mongos* пропустят допустимые ошибки записи.

Предотвращение фантомных операций

Существует странная проблема MongoDB, с которой вы можете столкнуться, особенно если вы загружаете данные в коллекцию. Предположим, у вас есть задание, которое запускает тысячи операций обновления в MongoDB, а MongoDB останавливается. Вы быстро останавливаете задание и отключаете все обновления, которые происходят в данный момент. Тем не менее вы по-прежнему видите, что новые обновления появляются сразу после того, как вы остановили старые, хотя задание уже не запущено!

Если вы загружаете данные с использованием неподтвержденных операций записи, ваше приложение будет запускать записи в MongoDB потенциально быстрее, чем MongoDB сможет их обработать. При резервном копировании эти записи будут накапливаться в буфере сокетов операционной системы. Когда вы останавливаете записи, над которыми работает MongoDB, это позволяет MongoDB начать обработку записей в буфере. Даже если вы запретите клиенту отправлять записи, любые записи, кото-

рые попали в буфер, будут обрабатываться MongoDB, поскольку они уже были «получены» (просто не были обработаны).

Лучший способ предотвратить появление таких фантомных операций записи – выполнять *подтвержденные* операции: заставить каждую запись ждать завершения предыдущей, а не только до тех пор, пока предыдущая запись не будет помещена в буфер на сервере базы данных.

Использование системного профилировщика

Чтобы найти медленные операции, можно использовать системный профилировщик, который записывает операции в специальную коллекцию *system.profile*. Профилировщик может предоставить вам массу информации об операциях, которые занимают много времени, но за это придется заплатить: он замедляет общую производительность сервера *mongod*. Таким образом, вы можете только периодически включать его для захвата куска трафика. Если ваша система уже сильно загружена, можно использовать другой метод диагностики проблем, описанный в этой главе.

По умолчанию профилировщик отключен и ничего не записывает. Его можно включить, выполнив функцию `db.setProfilingLevel()` в оболочке:

```
> db.setProfilingLevel(2)
{ "was" : 0, "slowms" : 100, "ok" : 1 }
```

Уровень 2 означает «профилировать все». Каждый запрос на чтение и запись, полученный базой данных, будет записан в коллекцию *system.profile* текущей базы данных. Профилирование включается для каждой базы данных и влечет за собой серьезные потери производительности: потребуется дополнительное время, чтобы сделать каждую запись, и каждая операция чтения должна захватывать блокировку записи (потому что должна сделать запись в коллекцию *system.profile*). Однако это даст вам исчерпывающий список того, что делает ваша система:

```
> db.foo.insert({x:1})
> db.foo.update({},{$set:{x:2}})
> db.foo.remove()
> db.system.profile.find().pretty()
{
    "op": "insert",
    "ns": "sample_mflix.foo",
    "command": {
        "insert": "foo",
        "ordered": true,
        "lsid": {
            "id": UUID("63b7df66-ca97-41f4-a245-eba825485147")
```

```
        },
        "$readPreference": {
            "mode": "secondaryPreferred"
        },
        "$db": "sample_mflix"
    },
    "ninserted": 1,
    "keysInserted": 1,
    "numYield": 0,
    "locks": { ...
    },
    "flowControl": {
        "acquireCount": NumberLong(3)
    },
    "responseLength": 45,
    "protocol": "op_msg",
    "millis": 33,
    "client": "127.0.0.1",
    "appName": "MongoDB Shell",
    "allUsers": [ ],
    "user": ""
}
{
    "op": "update",
    "ns": "sample_mAflix.foo",
    "command": {
        "q": {
        },
        "u": {
            "$set": {
                "x": 2
            }
        },
        "multi": false,
        "upsert": false
    },
    "keysExamined": 0,
    "docsExamined": 1,
    "nMatched": 1,
    "nModified": 1,
    "numYield": 0,
    "locks": { ...
    },
    "flowControl": {
        "acquireCount": NumberLong(1)
```

```
    },
    "millis": 0,
    "planSummary": "COLLSCAN",
    "execStats": { ...
    "inputStage": {
    ...
        }
    },
    "ts": ISODate("2019-09-03T22:39:33.856Z"),
    "client": "127.0.0.1",
    "appName": "MongoDB Shell",
    "allUsers": [ ],
    "user": ""
}
    {
    "op": "remove",
    "ns": "sample_mflix.foo",
    "command": {
        "q": {
        },
        "limit": 0
    },
    "keysExamined": 0,
    "docsExamined": 1,
    "ndeleted": 1,
    "keysDeleted": 1,
    "numYield": 0,
    "locks": { ... },
    "flowControl": {
        "acquireCount": NumberLong(1)
    },
    "millis": 0,
    "planSummary": "COLLSCAN",
    "execStats": { ...
        "inputStage": { ...}
    },
    "ts": ISODate("2019-09-03T22:39:33.858Z"),
    "client": "127.0.0.1",
    "appName": "MongoDB Shell",
    "allUsers": [ ],
    "user": ""
}
```

Можно использовать поле `"client"`, чтобы увидеть, какие пользователи какие операции отправляют в базу данных. Если вы используете аутенти-

фикацию, вы также можете видеть, какой пользователь выполняет каждую операцию.

Часто большинство операций, выполняемых вашей базой данных, вам не интересно, вас интересуют только медленные. Для этого вы можете установить уровень профилирования на 1. По умолчанию уровень 1 профилирует операции, которые занимают более 100 мс. Вы также можете указать второй аргумент, который определяет, что значит для вас слово «медленный». Так мы запишем все операции, которые заняли более 500 мс:

```
> db.setProfilingLevel(1, 500)
{ "was" : 2, "slowms" : 100, "ok" : 1 }
```

Чтобы отключить профилирование, установите уровень профилирования на 0:

```
> db.setProfilingLevel(0)
{ "was" : 1, "slowms" : 500, "ok" : 1 }
```

Как правило, не рекомендуется устанавливать низкое значение для параметра `slowms`. Даже при отключенном профилировании `slowms` влияет на сервер *mongod*: он устанавливает порог для вывода медленных операций в журнале. Таким образом, если вы установите для него значение 2, каждая операция, которая занимает более 2 мс, будет отображаться в журнале, даже если профилирование отключено. Поэтому если вы снизите значение, чтобы профилировать что-то, возможно, у вас возникнет желание снова повысить его перед выключением профилирования.

Текущий уровень профилирования можно увидеть с помощью функции `db.getProfilingLevel()`. Уровень профилирования не является постоянным: перезапуск базы данных очищает его.

Существуют параметры командной строки для настройки уровня профилирования, а именно `--profile` *уровень* и `--slowms` *время*, но повышение уровня профилирования обычно является временной мерой отладки. Это не то, что нужно добавлять в свою конфигурацию в долгосрочной перспективе.

В MongoDB версии 4.2 записи профилировщика и сообщения журнала диагностики были расширены для операций чтения и записи, чтобы помочь улучшить идентификацию медленных запросов, с добавлением полей `queryHash` и `planCacheKey`. Строка `queryHash` – это хеш формы запроса, зависит только от формы запроса. Каждая форма запроса связана с `queryHash`, что облегчает выделение запросов с помощью той же формы. `PlanCacheKey` – это хеш ключа для плановой записи кеша, связанной с запросом. Он включает в себя сведения как формы запроса, так и доступные на данный момент индексы для формы. Они помогают сопоставить доступную информацию от профилировщика, чтобы помочь с диагностикой производительности запросов.

Вычисление размеров

Чтобы обеспечить правильный объем диска и оперативной памяти, полезно знать, сколько места занимают документы, индексы, коллекции и базы данных. *См.* раздел «Вычисление рабочего множества» для получения информации о расчете рабочего множества.

Документы

Самый простой способ получить размер документа – использовать функцию оболочки `Object.bsonsize()`. Передайте любой документ, чтобы получить размер, который был бы при хранении в MongoDB.

Например, можно увидеть, что хранить идентификаторы в качестве идентификаторов объекта более эффективно, нежели хранить их в виде строк:

```
> Object.bsonsize({_id:ObjectId()})
22
> // ""+ObjectId() конвертирует идентификатор объекта в строку;
> Object.bsonsize({_id:""+ObjectId()})
39
```

На практике можно передавать документы напрямую из своих коллекций:

```
> Object.bsonsize(db.users.findOne())
```

Это точно показывает, сколько байтов документ занимает на диске. Однако тут не учитываются отступы или индексы, которые часто могут быть существенными факторами для размера коллекции.

Коллекции

Чтобы просмотреть информацию о всей коллекции, есть функция `stats`:

```
>db.movies.stats()
{
    "ns": "sample_mflix.movies",
    "size": 65782298,
    "count": 45993,
    "avgObjSize": 1430,
    "storageSize": 45445120,
    "capped": false,
    "wiredTiger": {
        "metadata": {
            "formatVersion": 1
        },
```

```
"creationString": "access_pattern_hint=none,allocation_size=4KB,\
app_metadata=(formatVersion=1),assert=(commit_timestamp=none,\
read_timestamp=none),block_allocation=best,block_compressor=\
snappy,cache_resident=false,checksum=on,colgroups=,collator=,\
columns=,dictionary=0,encryption=(keyid=,name=),exclusive=\
false,extractor=,format=btree,huffman_key=,huffman_value=,\
ignore_in_memory_cache_size=false,immutable=false,internal_item_\
max=0,internal_key_max=0,internal_key_truncate=true,internal_\
page_max=4KB,key_format=q,key_gap=10,leaf_item_max=0,leaf_key_\
max=0,leaf_page_max=32KB,leaf_value_max=64MB,log=(enabled=true),\
lsm=(auto_throttle=true,bloom=true,bloom_bit_count=16,bloom_\
config=,bloom_hash_count=8,bloom_oldest=false,chunk_count_limit\
=0,chunk_max=5GB,chunk_size=10MB,merge_custom=(prefix=,start_\
generation=0,suffix=),merge_max=15,merge_min=0),memory_page_image\
_max=0,memory_page_max=10m,os_cache_dirty_max=0,os_cache_max=0,\
prefix_compression=false,prefix_compression_min=4,source=,split_\
deepen_min_child=0,split_deepen_per_child=0,split_pct=90,type=file,\
value_format=u",
"type": "file",
"uri": "statistics:table:collection-14--2146526997547809066",
"LSM": {
    "bloom filter false positives": 0,
    "bloom filter hits": 0,
    "bloom filter misses": 0,
    "bloom filter pages evicted from cache": 0,
    "bloom filter pages read into cache": 0,
    "bloom filters in the LSM tree": 0,
    "chunks in the LSM tree": 0,
    "highest merge generation in the LSM tree": 0,
    "queries that could have benefited from a Bloom filter
        that did not exist" : 0,
    "sleep for LSM checkpoint throttle": 0,
    "sleep for LSM merge throttle": 0,
    "total size of bloom filters": 0
},
"block-manager": {
    "allocations requiring file extension": 0,
    "blocks allocated": 1358,
    "blocks freed": 1322,
    "checkpoint size": 39219200,
    "file allocation unit size": 4096,
    "file bytes available for reuse": 6209536,
    "file magic number": 120897,
    "file major version number": 1,
    "file size in bytes": 45445120,
```

```
        "minor version number": 0
    },
    "btree": {
        "btree checkpoint generation": 22,
        "column-store fixed-size leaf pages": 0,
        "column-store internal pages": 0,
        "column-store variable-size RLE encoded values": 0,
        "column-store variable-size deleted values": 0,
        "column-store variable-size leaf pages": 0,
        "fixed-record size": 0,
        "maximum internal page key size": 368,
        "maximum internal page size": 4096,
        "maximum leaf page key size": 2867,
        "maximum leaf page size": 32768,
        "maximum leaf page value size": 67108864,
        "maximum tree depth": 0,
        "number of key/value pairs": 0,
        "overflow pages": 0,
        "pages rewritten by compaction": 1312,
        "row-store empty values": 0,
        "row-store internal pages": 0,
        "row-store leaf pages": 0
    },
    "cache": {
        "bytes currently in the cache": 40481692,
        "bytes dirty in the cache cumulative": 40992192,
        "bytes read into cache": 37064798,
        "bytes written from cache": 37019396,
        "checkpoint blocked page eviction": 0,
        "data source pages selected for eviction unable to be evicted": 32,
        "eviction walk passes of a file": 0,
        "eviction walk target pages histogram - 0-9": 0,
        "eviction walk target pages histogram - 10-31": 0,
        "eviction walk target pages histogram - 128 and higher": 0,
        "eviction walk target pages histogram - 32-63": 0,
        "eviction walk target pages histogram - 64-128": 0,
        "eviction walks abandoned": 0,
        "eviction walks gave up because they restarted their walk twice": 0,
        "eviction walks gave up because they saw too many pages
        and found no candidates" : 0,
        "eviction walks gave up because they saw too many pages
        and found too few candidates" : 0,
        "eviction walks reached end of tree": 0,
        "eviction walks started from root of tree": 0,
        "eviction walks started from saved location in tree": 0,
```

```
        "hazard pointer blocked page eviction": 0,
        "in-memory page passed criteria to be split": 0,
        "in-memory page splits": 0,
        "internal pages evicted": 8,
        "internal pages split during eviction": 0,
        "leaf pages split during eviction": 0,
        "modified pages evicted": 1312,
        "overflow pages read into cache": 0,
        "page split during eviction deepened the tree": 0,
        "page written requiring cache overflow records": 0,
        "pages read into cache": 1330,
        "pages read into cache after truncate": 0,
        "pages read into cache after truncate in prepare state": 0,
        "pages read into cache requiring cache overflow entries": 0,
        "pages requested from the cache": 3383,
        "pages seen by eviction walk": 0,
        "pages written from cache": 1334,
        "pages written requiring in-memory restoration": 0,
        "tracked dirty bytes in the cache": 0,
        "unmodified pages evicted": 8
    },
    "cache_walk": {
        "Average difference between current eviction generation
        when the page was last considered" : 0,
        "Average on-disk page image size seen": 0,
        "Average time in cache for pages that have been visited
        by the eviction server" : 0,
        "Average time in cache for pages that have not been visited
        by the eviction server" : 0,
        "Clean pages currently in cache": 0,
        "Current eviction generation": 0,
        "Dirty pages currently in cache": 0,
        "Entries in the root page": 0,
        "Internal pages currently in cache": 0,
        "Leaf pages currently in cache": 0,
        "Maximum difference between current eviction generation
        when the page was last considered" : 0,
        "Maximum page size seen": 0,
        "Minimum on-disk page image size seen": 0,
        "Number of pages never visited by eviction server": 0,
        "On-disk page image sizes smaller than a single allocation unit": 0,
        "Pages created in memory and never written": 0,
        "Pages currently queued for eviction": 0,
        "Pages that could not be queued for eviction": 0,
        "Refs skipped during cache traversal": 0,
```

```
        "Size of the root page": 0,
        "Total number of pages currently in cache": 0
    },
    "compression": {
        "compressed page maximum internal page size
        prior to compression" : 4096,
        "compressed page maximum leaf page size
        prior to compression " : 131072,
        "compressed pages read": 1313,
        "compressed pages written": 1311,
        "page written failed to compress": 1,
        "page written was too small to compress": 22
    },
    "cursor": {
        "bulk loaded cursor insert calls": 0,
        "cache cursors reuse count": 0,
        "close calls that result in cache": 0,
        "create calls": 1,
        "insert calls": 0,
        "insert key and value bytes": 0,
        "modify": 0,
        "modify key and value bytes affected": 0,
        "modify value bytes modified": 0,
        "next calls": 0,
        "open cursor count": 0,
        "operation restarted": 0,
        "prev calls": 1,
        "remove calls": 0,
        "remove key bytes removed": 0,
        "reserve calls": 0,
        "reset calls": 2,
        "search calls": 0,
        "search near calls": 0,
        "truncate calls": 0,
        "update calls": 0,
        "update key and value bytes": 0,
        "update value size change": 0
    },
    "reconciliation": {
        "dictionary matches": 0,
        "fast-path pages deleted": 0,
        "internal page key bytes discarded using suffix compression": 0,
        "internal page multi-block writes": 0,
        "internal-page overflow keys": 0,
        "leaf page key bytes discarded using prefix compression": 0,
```

```
            "leaf page multi-block writes": 0,
            "leaf-page overflow keys": 0,
            "maximum blocks required for a page": 1,
            "overflow values written": 0,
            "page checksum matches": 0,
            "page reconciliation calls": 1334,
            "page reconciliation calls for eviction": 1312,
            "pages deleted": 0
        },
        "session": {
            "object compaction": 4
        },
        "transaction": {
            "update conflicts": 0
        }
    },
    "nindexes": 5,
    "indexBuilds": [],
    "totalIndexSize": 46292992,
    "indexSizes": {
        "_id_": 446464,
        "$**_text": 44474368,
        "genres_1_imdb.rating_1_metacritic_1": 724992,
        "tomatoes_rating": 307200,
        "getMovies": 339968
    },
    "scaleFactor": 1,
    "ok": 1
}
```


Вначале у нас идет пространство имен (`"sample_mflix.movies"`), а затем количество всех документов в коллекции. Следующая пара полей связана с размером коллекции. `"size"` – это то, что вы бы получили, если бы вызвали функцию `Object.bsonsize()` для каждого элемента в коллекции и добавили все размеры: это фактическое количество байтов в памяти, которые обрабатываются в коллекции при распаковке. Также если вы возьмете `"avgObjSize"` и умножите его на `"count"`, то получите размер без сжатия в памяти.

Как упоминалось ранее, общее количество байтов документов не позволяет сохранить пространство, сэкономленное за счет сжатия коллекции. `"storageSize"` может быть меньше, чем `"size"`, отражающий пространство, сэкономленное за счет сжатия.

`"nindexes"` – это количество индексов в коллекции. Индекс не учитывается в `"nindexes"` до тех пор, пока не завершится его построение, и его нельзя использовать, пока он не появится в этом списке.

В целом индексы будут намного больше, чем объем данных, которые они хранят. Можно минимизировать это свободное пространство, используя индексы с балансировкой вправо (как описано в разделе «Знакомство с составными индексами»). Случайно распределенные индексы обычно будут занимать примерно 50 % свободного пространства, тогда как индексы в порядке возрастания будут занимать 10 %.

По мере того как ваши коллекции становятся больше, чтение вывода функции `stats` с размерами в миллиарды байтов или более может оказаться затруднительным. Поэтому вы можете передать коэффициент масштабирования: 1024 для килобайтов, `1024*1024` для мегабайтов и т. д. Например, так можно получить статистику коллекции в терабайтах:

```
> db.big.stats(1024*1024*1024*1024)
```

Базы данных

У баз данных есть функция `stats`, напоминающая аналогичную функцию для коллекций:

```
> db.stats()
{
    "db": "sample_mflix",
    "collections": 5,
    "views": 0,
    "objects": 98308,
    "avgObjSize": 819.8680982219148,
    "dataSize": 80599593,
    "storageSize": 53620736,
    "numExtents": 0,
    "indexes": 12,
    "indexSize": 47001600,
    "scaleFactor": 1,
    "fsUsedSize": 355637043200,
    "fsTotalSize": 499963174912,
    "ok": 1
}
```

Вначале у нас идет имя базы данных, количество коллекций, которые она содержит, и количество просмотров для базы данных. Поле `"objects"` – общее количество документов во всех коллекциях в этой базе данных.

Основная часть документа содержит информацию о размере ваших данных.

Поле `"fsTotalSize"` всегда должно быть самым большим: это общий размер емкости диска в файловой системе, где экземпляр MongoDB хранит данные. Поле `"fsUsedSize"` обозначает общее пространство, используемое

MongoDB в этой файловой системе в настоящее время. Оно должно соответствовать общему пространству, используемому всеми файлами в вашем каталоге данных.

Следующим по величине полем обычно будет `"dataSize"`. Это размер несжатых данных, хранящихся в этой базе данных. Оно не совпадает с полем `"storageSize"`, потому что данные обычно сжимаются в WiredTiger. `"indexSize"` – объем пространства, занимаемого всеми индексами для этой базы данных.

Функция `db.stats()` может принимать аргумент масштаба так же, как и функция коллекций `stats`. Если вызвать `db.stats()` для несуществующей базы данных, все значения будут равны нулю.

Помните, что перечисление баз данных в системе с высоким процентом блокировки может быть очень медленным и может блокировать другие операции. Избегайте этого по возможности.

Использование утилит mongotop и mongostat

MongoDB поставляется с утилитами командной строки, которые могут помочь вам определить, что она делает, выводя статистику каждые несколько секунд.

mongotop похожа на утилиту Unix *top*: она дает представление о том, какие коллекции наиболее загружены. Вы также можете выполнить команду `mongotop` с параметром `--locks`, чтобы получить статистику блокировок для каждой базы данных.

mongostat предоставляет информацию для всего сервера. По умолчанию она выводит список статистики один раз в секунду, хотя ее можно настроить, передавая различное количество секунд в командной строке. Каждое из полей подсчитывает, сколько раз имела место активность с момента последнего вывода поля:

`insert/query/update/delete/getmore/command`
: Простые подсчеты количества выполненных операций.

`flushes`
: Сколько раз сервер *mongod* сбрасывал данные на диск.

`mapped`
: Количество памяти, отображенное *mongod*. В целом оно приблизительно равно размеру вашего каталога данных.

`vsize`
: Количество виртуальной памяти, которую использует *mongod*. Обычно оно в два раза больше вашего каталога данных (один раз для сопоставленных файлов, еще раз для ведения журнала).

`res`
: Количество памяти, которое использует *mongod*. Как правило, оно должно быть как можно ближе к количеству всей памяти на машине.

`locked db`
: База данных, которая провела большую часть времени в заблокированном состоянии в последнем временном интервале. В этом поле указывается процент времени, в течение которого база данных была заблокирована, и длительность глобальной блокировки. Это означает, что это значение может превышать 100 %.

`idx miss %`
: Процент обращений к индексу, которые были вызваны ошибкой страницы (потому что индексной записи или раздела индекса, который искали, не было в памяти, поэтому *mongod* пришлось перейти на диск). Это поле с самым непонятным названием в выводе.

`qr|qw`
: Размер очереди для операций чтения и записи (то есть сколько операций чтения и записи блокируется, ожидая обработки).

`ar|aw`
: Количество активных клиентов (то есть клиентов, которые в настоящее время выполняют операции чтения и записи).

`netIn`
: Количество сетевых байтов в подсчете MongoDB (не обязательно такое же, как определила бы ОС).

`netOut`
: Количество сетевых байтов, подсчитанных MongoDB.

`conn`
: Количество соединений, открытых на этом сервере, как входящих, так и исходящих.

`time`
: Время, когда были взяты эти статистические данные.

Можно запустить утилиту *mongostat* для набора реплик или сегментированного кластера. Если вы используете параметр `--discover`, *mongostat* попытается найти все члены набора или кластера от члена, к которому он изначально подключен, и будет выводить по одной строке на сервер в секунду для каждого. В случае с большим кластером этот процесс быстро может стать неуправляемым, но это может быть полезно для небольших кластеров и инструментов, которые могут потреблять данные и представлять их в более читабельном виде.

mongostat – отличный способ получить быстрый снэпшот того, что делает ваша база данных, однако для долгосрочного мониторинга предпочтительнее использовать MongoDB Atlas или Ops Manager (*см.* главу 22).

Глава 19

Обеспечение безопасности в MongoDB

Чтобы защитить свой кластер MongoDB и хранящиеся в нем данные, необходимо применить следующие меры безопасности:

- включить авторизацию и обеспечить аутентификацию;
- зашифровать обмен данными;
- зашифровать данные.

В этой главе показано, как обеспечить выполнение первых двух пунктов с помощью руководства по использованию в MongoDB поддержки стандарта x.509 для настройки аутентификации и шифрования на транспортном уровне, чтобы обеспечить безопасный обмен данными между клиентами и серверами в наборе реплик MongoDB. Мы коснемся шифрования данных на уровне хранилища в следующей главе.

Аутентификация и авторизация в MongoDB

Хотя аутентификация и авторизация тесно связаны, важно отметить, что аутентификация отличается от авторизации. Целью аутентификации является проверка личности пользователя, в то время как авторизация определяет доступ верифицированного пользователя к ресурсам и операциям.

Механизмы аутентификации

Включение авторизации в кластере MongoDB обеспечивает аутентификацию и гарантирует, что пользователи могут выполнять только те действия, для которых они авторизованы, в зависимости от своих ролей. Версия MongoDB для сообщества обеспечивает поддержку механизма хранения данных и протокола аутентификации посредством пароля SCRAM (Salted Challenge Response Authentication Mechanism) и стандарта x.509. В дополнение к SCRAM и x.509 MongoDB Enterprise поддерживает сетевой протокол аутентификации Kerberos и протокол LDAP. *См*. документацию

на странице https://docs.mongodb.com/manual/core/authentication/ для получения подробной информации о различных механизмах аутентификации, которые поддерживает MongoDB. В этой главе мы сосредоточимся на аутентификации на основе сертификата x.509. Цифровой сертификат x.509 использует общепринятый стандарт инфраструктуры открытых ключей (PKI) x.509 для проверки того, что открытый ключ принадлежит тому, кто его предъявляет.

При добавлении пользователя в MongoDB вы должны создать пользователя в определенной базе данных.

Эта база данных является базой данных аутентификации для пользователя; вы можете использовать любую базу данных для этой цели. База данных имени пользователя и аутентификации служит уникальным идентификатором для пользователя. Однако привилегии пользователя не ограничиваются его базой данных аутентификации.

Авторизация

При добавлении пользователя в MongoDB вы должны создать пользователя в определенной базе данных. Эта база данных является базой данных аутентификации пользователя; для этой цели можно использовать любую базу данных. База данных имени пользователя и аутентификации служит уникальным идентификатором пользователя. Однако привилегии пользователя не ограничиваются базой данных аутентификации.

При создании пользователя вы можете указать операции, которые пользователь может выполнять с любыми ресурсами, к которым у него должен быть доступ. Ресурсы включают в себя кластер, базы данных и коллекции.

MongoDB предоставляет ряд встроенных ролей, которые предоставляют обычно необходимые права для пользователей базы данных. К ним относятся:

`read`
: Чтение данных обо всех несистемных коллекциях и системных коллекциях *system.indexes*, *system.js* и *system.namespaces*.

`readWrite`
: Предоставляет те же привилегии, что и `read`, плюс возможность изменять данные во всех несистемных коллекциях и коллекции *system.js*.

`dbAdmin`
: Выполнение административных задач, например задач, связанных со схемой, индексирование и сбор статистики (не предоставляет привилегии для управления пользователями и ролями).

`userAdmin`
: Создание и изменение ролей и пользователей в текущей базе данных.

`dbOwner`
: Сочетает привилегии, предоставленные ролями *readWrite*, *dbAdmin* и *userAdmin*.

`clusterManager`
: Выполнение действий по управлению и мониторингу в кластере.

`clusterMonitor`
: Предоставляет доступ только для чтения к таким инструментам мониторинга, как MongoDB Cloud Manager и агент мониторинга Ops Manager.

`hostManager`
: Мониторинг и управление серверами.

`clusterAdmin`
: Сочетает привилегии, предоставляемые ролями *clusterManager*, *clusterMonitor* и *host-Manager*, а также действием *dropDatabase*.

`backup`
: Предоставляет достаточные привилегии для использования агента резервного копирования MongoDB Cloud Manager или агента резервного копирования Ops Manager либо применения *mongodump* для резервного копирования всего экземпляра сервера *mongod*.

`restore`
: Предоставляет привилегии, необходимые для восстановления данных из резервных копий, которые не включают данные из коллекции *system.profile*.

`readAnyDatabase`
: Предоставляет те же привилегии, что и *read* для всех баз данных, кроме *local* и *config*, плюс действие *listDatabases* в кластере как единое целое.

`readWriteAnyDatabase`
: Предоставляет те же привилегии, что и *readWrite* во всех базах данных, кроме *local* и *config* (фактически роль суперпользователя).

`root`
: Предоставляет доступ к операциям и всем ресурсам ролей *readWriteAnyDatabase*, *dbAdminAnyDatabase*, *userAdminAnyDatabase*, *clusterAdmin*, *restore* и *backup*.

Вы также можете создавать так называемые «определяемые пользователем роли». Это специальные роли, которые группируют полномочия на выполнение определенных операций и помечают их именем, чтобы вы могли легко предоставить этот набор прав нескольким пользователям.

Детальное рассмотрение этих типов ролей выходит за рамки данной главы. Однако это знакомство должно дать вам довольно хорошее представление о том, какие возможности предоставляет авторизация в MongoDB. Для получения более подробной информации, пожалуйста, обратитесь к соответствующему разделу документации к MongoDB (https://docs.mongodb.com/manual/core/authorization/).

Чтобы убедиться, что вы можете добавлять новых пользователей по мере необходимости, для начала нужно создать пользователя с правами администратора. MongoDB не создает пользователя root или пользователя с правами администратора по умолчанию при активации аутентификации и авторизации, независимо от того, какой режим аутентификации вы используете (x.509 не исключение).

В MongoDB аутентификация и авторизация не активируются по умолчанию. Необходимо явно включить их, используя параметр `--auth` для команды `mongod` или указав значение `"enabled"` для параметра `security.authorization` в конфигурационном файле MongoDB.

Чтобы настроить набор реплик, сначала запустите его без включенной аутентификации и авторизации, затем создайте пользователя с правами администратора и пользователей, которые вам понадобятся для каждого клиента.

Использование сертификатов x.509 для аутентификации членов и клиентов

Учитывая, что все производственные кластеры MongoDB состоят из нескольких членов, для защиты кластера важно, чтобы все службы, обменивающиеся данными в кластере, проходили аутентификацию друг с другом. Каждый член набора реплик должен аутентифицироваться с другими для обмена данными. Аналогично, клиенты должны проходить аутентификацию с первичным и всеми вторичными узлами, с которыми они общаются.

Когда речь идет о стандарте x.509, необходимо, чтобы все сертификаты были подписаны доверенным центром сертификации. Подписание удостоверяет, что названному субъекту сертификата принадлежит открытый ключ, связанный с этим сертификатом. Центр сертификации выполняет роль доверенной третьей стороны, чтобы предотвращать атаки посредника (man-in-the-midle).

На рис. 19.1 показана аутентификация на основе сертификата x.509, используемая для защиты набора реплики MongoDB из трех членов. Обратите внимание на аутентификацию между клиентом и членами набора реплик и доверительные отношения с центром сертификации (ЦС).

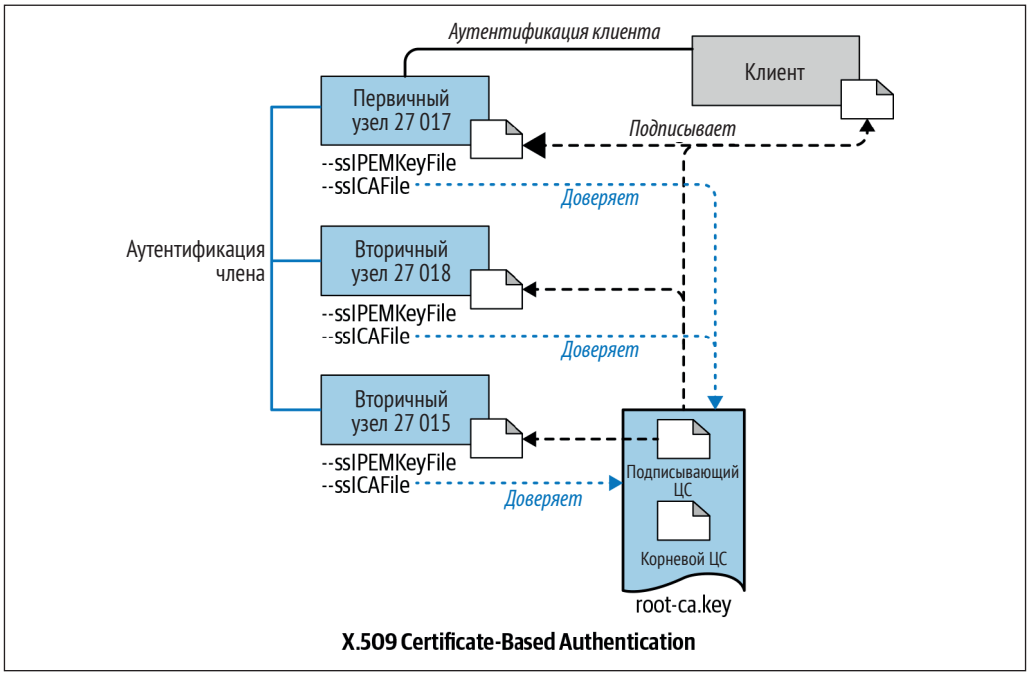

Рис. 19.1. Обзор иерархии доверия аутентификации на основе сертификата x.509 для набора реплик из трех участников, используемого в этой главе

У членов и клиента есть собственный сертификат, подписанный ЦС. При производственном использовании в вашем развертывании MongoDB должны использоваться действительные сертификаты, созданные и подписанные одним центром сертификации. Вы или ваша компания можете создавать и поддерживать независимый центр сертификации или использовать сертификаты, созданные сторонним поставщиком TLS/SSL.

Мы будем ссылаться на сертификаты, используемые для внутренней аутентификации для проверки членства в кластере, в качестве сертификатов членов. Сертификаты участников и клиентские сертификаты (используются для аутентификации клиентов) имеют структуру, похожую на эту:

```
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 1 (0x1)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, ST=NY, L=New York, O=MongoDB, CN=CA-SIGNER
        Validity
            Not Before: Nov 11 22:00:03 2018 GMT
            Not After : Nov 11 22:00:03 2019 GMT
        Subject: C=US, ST=NY, L=New York, O=MongoDB, OU=MyServers, CN=server1
        Subject Public Key Info:
```

```
        Public Key Algorithm: rsaEncryption
            Public-Key: (2048 bit)
             Modulus:
                00:d3:1c:29:ba:3d:29:44:3b:2b:75:60:95:c8:83:
                fc:32:1a:fa:29:5c:56:f3:b3:66:88:7f:f9:f9:89:
                ff:c2:51:b9:ca:1d:4c:d8:b8:5a:fd:76:f5:d3:c9:
                95:9c:74:52:e9:8d:5f:2e:6b:ca:f8:6a:16:17:98:
                dc:aa:bf:34:d0:44:33:33:f3:9d:4b:7e:dd:7a:19:
                1b:eb:3b:9e:21:d9:d9:ba:01:9c:8b:16:86:a3:52:
                a3:e6:e4:5c:f7:0c:ab:7a:1a:be:c6:42:d3:a6:01:
                8e:0a:57:b2:cd:5b:28:ee:9d:f5:76:ca:75:7a:c1:
                7c:42:d1:2a:7f:17:fe:69:17:49:91:4b:ca:2e:39:
                b4:a5:e0:03:bf:64:86:ca:15:c7:b2:f7:54:00:f7:
                02:fe:cf:3e:12:6b:28:58:1c:35:68:86:3f:63:46:
                75:f1:fe:ac:1b:41:91:4f:f2:24:99:54:f2:ed:5b:
                fd:01:98:65:ac:7a:7a:57:2f:a8:a5:5a:85:72:a6:
                9e:fb:44:fb:3b:1c:79:88:3f:60:85:dd:d1:5c:1c:
                db:62:8c:6a:f7:da:ab:2e:76:ac:af:6d:7d:b1:46:
                69:c1:59:db:c6:fb:6f:e1:a3:21:0c:5f:2e:8e:a7:
                d5:73:87:3e:60:26:75:eb:6f:10:c2:64:1d:a6:19:
                f3:0b
            Exponent: 65537 (0x10001)
    Signature Algorithm: sha256WithRSAEncryption
        5d:dd:b2:35:be:27:c2:41:4a:0d:c7:8c:c9:22:05:cd:eb:88:
        9d:71:4f:28:c1:79:71:3c:6d:30:19:f4:9c:3d:48:3a:84:d0:
        19:00:b1:ec:a9:11:02:c9:a6:9c:74:e7:4e:3c:3a:9f:23:30:
        50:5a:d2:47:53:65:06:a7:22:0b:59:71:b0:47:61:62:89:3d:
        cf:c6:d8:b3:d9:cc:70:20:35:bf:5a:2d:14:51:79:4b:7c:00:
        30:39:2d:1d:af:2c:f3:32:fe:c2:c6:a5:b8:93:44:fa:7f:08:
        85:f0:01:31:29:00:d4:be:75:7e:0d:f9:1a:f5:e9:75:00:9a:
        7b:d0:eb:80:b1:01:00:c0:66:f8:c9:f0:35:6e:13:80:70:08:
        5b:95:53:4b:34:ec:48:e3:02:88:5c:cd:a0:6c:b4:bc:65:15:
        4d:c8:41:9d:00:f5:e7:f2:d7:f5:67:4a:32:82:2a:04:ae:d7:
        25:31:0f:34:e8:63:a5:93:f2:b5:5a:90:71:ed:77:2a:a6:15:
        eb:fc:c3:ac:ef:55:25:d1:a1:31:7a:2c:80:e3:42:c2:b3:7d:
        5e:9a:fc:e4:73:a8:39:50:62:db:b1:85:aa:06:1f:42:27:25:
        4b:24:cf:d0:40:ca:51:13:94:97:7f:65:3e:ed:d9:3a:67:08:
        79:64:a1:ba
-----BEGIN CERTIFICATE-----
MIIDODCCAiACAQEwDQYJKoZIhvcNAQELBQAwWTELMAkGA1UEBhMCQ04xCzAJBgNV
BAgMAkdEMREwDwYDVQQHDAhTaGVuemhlbjEWMBQGA1UECgwNTW9uZ29EQiBDaGlu
YTESMBAGA1UEAwwJQ0EtU0lHTkVSMB4XDTE4MTExMTIyMDAwM1oXDTE5MTExMTIy
MDAwM1owazELMAkGA1UEBhMCQ04xCzAJBgNVBAgMAkdEMREwDwYDVQQHDAhTaGVu
emhlbjEWMBQGA1UECgwNTW9uZ29EQiBDaGluYTESMBAGA1UECwwJTXlTZXJ2ZXJz
MRAwDgYDVQQDDAdzZXJ2ZXIxMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKC
```

```
AQEA0xwpuj0pRDsrdWCVyIP8Mhr6KVxW87NmiH/5+Yn/wlG5yh1M2Lha/Xb108mV
nHRS6Y1fLmvK+GoWF5jcqr800EQzM/OdS37dehkb6zueIdnZugGcixaGo1Kj5uRc
9wyrehq+xkLTpgGOCleyzVso7p31dsp1esF8QtEqfxf+aRdJkUvKLjm0peADv2SG
yhXHsvdUAPcC/s8+EmsoWBw1aIY/Y0Z18f6sG0GRT/IkmVTy7Vv9AZhlrHp6Vy+o
pVqFcqae+0T7Oxx5iD9ghd3RXBzbYoxq99qrLnasr219sUZpwVnbxvtv4aMhDF8u
jqfVc4c+YCZ1628QwmQdphnzCwIDAQABMA0GCSqGSIb3DQEBCwUAA4IBAQBd3bI1
vifCQUoNx4zJIgXN64idcU8owXlxPG0wGfScPUg6hNAZALHsqRECyaacdOdOPDqf
IzBQWtJHU2UGpyILWXGwR2FiiT3Pxtiz2cxwIDW/Wi0UUXlLfAAwOS0dryzzMv7C
xqW4k0T6fwiF8AExKQDUvnV+Dfka9el1AJp70OuAsQEAwGb4yfA1bhOAcAhblVNL
NOxI4wKIXM2gbLS8ZRVNyEGdAPXn8tf1Z0oygioErtclMQ806GOlk/K1WpBx7Xcq
phXr/MOs71Ul0aExeiyA40LCs31emvzkc6g5UGLbsYWqBh9CJyVLJM/QQMpRE5SX
f2U+7dk6Zwh5ZKG6
-----END CERTIFICATE-----
```

Чтобы использовать стандарт x.509 в MongoDB, сертификаты членов должны обладать следующими свойствами:

- единый ЦС должен выдавать все сертификаты x.509 для членов кластера;
- уникальное имя (DN), находящееся в субъекте сертификата члена, должно указывать непустое значение хотя бы для одного из следующих атрибутов: организация (O), организационное подразделение (OU) или компонент домена (DC);
- атрибуты O, OU и DC должны совпадать с атрибутами из сертификатов для других членов кластера;
- общее имя (CN) или альтернативное имя субъекта (SAN) должно совпадать с именем хоста сервера, используемого другими членами кластера.

Руководство по аутентификации в MongoDB и шифрованию на транспортном уровне

В этом разделе мы настроим корневой и промежуточный ЦС. Рекомендуется подписывать сертификаты сервера и клиента с помощью промежуточного ЦС.

Создание центра сертификации

Прежде чем мы сможем генерировать подписанные сертификаты для членов нашего набора реплик, для начала нужно решить проблему центра сертификации. Как упоминалось ранее, мы можем создать и поддерживать независимый центр сертификации либо использовать сертификаты, созданные сторонним поставщиком TLS/SSL. Мы создадим свой собствен-

ный ЦС, который будем использовать в качестве примера в этой главе. Обратите внимание, что доступ ко всем примерам кода, приведенным в данной главе, можно получить из репозитория GitHub, поддерживаемого для этой книги. Примеры взяты из сценария, который можно использовать для развертывания защищенного набора реплик. В этих примерах вы увидите комментарии к данному сценарию.

Создание корневого центра сертификации

Для создания нашего ЦС мы будем использовать библиотеку OpenSSL. Чтобы продолжить, убедитесь, что у вас есть доступ к ней на вашем локальном компьютере.

Корневой центр сертификации находится в верхней части цепочки сертификатов. Это основной источник доверия. В идеале должен использоваться сторонний центр сертификации. Однако в случае изолированной сети (что типично для среды крупных предприятий) или в целях тестирования вам понадобится использовать локальный ЦС.

Для начала мы инициализируем некоторые переменные:

```
dn_prefix="/C=US/ST=NY/L=New York/O=MongoDB"
ou_member="MyServers"
ou_client="MyClients"
mongodb_server_hosts=( "server1" "server2" "server3" )
mongodb_client_hosts=( "client1" "client2" )
mongodb_port=27017
```

Затем создадим пару ключей и сохраним ее в файле *root-ca.key*:

```
# !!! При работе в производственных системах нужно будет защитить ключи с помощью пароля
# openssl genrsa -aes256 -out root-ca.key 4096
openssl genrsa -out root-ca.key 4096
```

Далее мы создадим конфигурационный файл для хранения наших настроек OpenSSL, которые будем использовать для генерации сертификатов:

```
# For the CA policy
[ policy_match ]
countryName = match
stateOrProvinceName = match
organizationName = match
organizationalUnitName = optional
commonName = supplied
emailAddress = optional

[ req ]
default_bits = 4096
```

```
default_keyfile = server-key.pem
default_md = sha256
distinguished_name = req_dn
req_extensions = v3_req
x509_extensions = v3_ca # Расширения, добавляемые к самозаверяющему сертификату;

[ v3_req ]
subjectKeyIdentifier = hash
basicConstraints = CA:FALSE
keyUsage = critical, digitalSignature, keyEncipherment
nsComment = "OpenSSL Generated Certificate"
extendedKeyUsage = serverAuth, clientAuth

[ req_dn ]
countryName = Country Name (2-letter code)
countryName_default = US
countryName_min = 2
countryName_max = 2

stateOrProvinceName = State or Province Name (full name)
stateOrProvinceName_default = NY
stateOrProvinceName_max = 64

localityName = Locality Name (eg, city)
localityName_default = New York
localityName_max = 64

organizationName = Organization Name (eg, company)
organizationName_default = MongoDB
organizationName_max = 64

organizationalUnitName = Organizational Unit Name (eg, section)
organizationalUnitName_default = Education
organizationalUnitName_max = 64

commonName = Common Name (eg, YOUR name)
commonName_max = 64

[ v3_ca ]
# Расширения для типичного ЦС;

subjectKeyIdentifier = hash
basicConstraints = critical,CA:true
authorityKeyIdentifier = keyid:always,issuer:always

# Использование ключа: это типично для сертификата ЦС. Однако, поскольку это
```

```
# не даст использовать его в качестве тестового самозаверяющего сертификата,
# его лучше оставить по умолчанию.
keyUsage = critical,keyCertSign,cRLSign
```

Затем, используя команду `openssl req`, мы создадим корневой сертификат. Поскольку корень – самая верхняя часть цепочки полномочий, мы самостоятельно подпишем этот сертификат с помощью закрытого ключа, который мы создали на предыдущем этапе (он хранится в файле *root-ca.key*). Параметр `-x509` сообщает команде `openssl req`, что мы хотим самостоятельно подписать сертификат, используя закрытый ключ, предоставленный параметру `-key`. В результате получается файл с именем *root-ca.crt*:

```
openssl req -new -x509 -days 1826 -key root-ca.key -out root-ca.crt \
  -config openssl.cnf -subj "$dn_prefix/CN=ROOTCA"
```

Если вы посмотрите на этот файл, то обнаружите, что он содержит открытый сертификат для корневого ЦС. Вы можете проверить содержимое, взглянув на удобочитаемую версию сертификата, созданного этой командой:

```
openssl x509 -noout -text -in root-ca.crt
```

Вывод команды будет выглядеть примерно так:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            1e:83:0d:9d:43:75:7c:2b:d6:2a:dc:7e:a2:a2:25:af:5d:3b:89:43
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, ST = NY, L = New York, O = MongoDB, CN = ROOTCA
        Validity
            Not Before: Sep 11 21:17:24 2019 GMT
            Not After : Sep 10 21:17:24 2024 GMT
        Subject: C = US, ST = NY, L = New York, O = MongoDB, CN = ROOTCA
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (4096 bit)
                Modulus:
                    00:e3:de:05:ae:ba:c9:e0:3f:98:37:18:77:02:35:
                    e7:f6:62:bc:c3:ae:38:81:8d:04:88:da:6c:e0:57:
                    c2:90:86:05:56:7b:d2:74:23:54:f8:ca:02:45:0f:
                    38:e7:e2:0b:69:ea:f6:c8:13:8f:6c:2d:d6:c1:72:
                    64:17:83:4e:68:47:cf:de:37:ed:6e:38:b2:ab:3a:
                    e4:45:a8:fa:08:90:a0:f3:0d:3a:14:d8:9a:8d:69:
                    e7:cf:93:1a:71:53:4f:13:29:50:b0:2f:b6:b8:19:
```

```
                    2a:40:21:15:90:43:e7:d8:d8:f3:51:e5:95:58:87:
                    6c:45:9f:61:fc:b5:97:cf:5b:4e:4a:1f:72:c9:0c:
                    e9:8c:4c:d1:ca:df:b3:a4:da:b4:10:83:81:01:b1:
                    c8:09:22:76:c7:1e:96:c7:e6:56:27:8d:bc:fb:17:
                    ed:d9:23:3f:df:9c:ef:03:20:cc:c3:c4:55:cc:9f:
                    ad:d4:8d:81:95:c3:f1:87:f8:d4:5a:5e:e0:a8:41:
                    27:c8:0d:52:91:e4:2b:db:25:d6:b7:93:8d:82:33:
                    7a:a7:b8:e8:cd:a8:e2:94:3d:d6:16:e1:4e:13:63:
                    3f:77:08:10:cf:23:f6:15:7c:71:24:97:ef:1c:a2:
                    68:0f:82:e2:f7:24:b3:aa:70:1a:4a:b4:ca:4d:05:
                    92:5e:47:a2:3d:97:82:f6:d8:c8:04:a7:91:6c:a4:
                    7d:15:8e:a8:57:70:5d:50:1c:0b:36:ba:78:28:f2:
                    da:5c:ed:4b:ea:60:8c:39:e6:a1:04:26:60:b3:e2:
                    ee:4f:9b:f9:46:3c:7e:df:82:88:29:c2:76:3e:1a:
                    a4:81:87:1f:ce:9e:41:68:de:6c:f3:89:df:ae:02:
                    e7:12:ee:93:20:f1:d2:d6:3d:36:58:ee:71:bf:b3:
                    c5:e7:5a:4b:a0:12:89:ed:f7:cc:ec:34:c7:b2:28:
                    a8:1a:87:c6:8b:5e:d2:c8:25:71:ba:ff:d0:82:1b:
                    5e:50:a9:8a:c6:0c:ea:4b:17:a6:cc:13:0a:53:36:
                    c6:9d:76:f2:95:cc:ac:b9:64:d5:72:fc:ab:ce:6b:
                    59:b1:3a:f2:49:2f:2c:09:d0:01:06:e4:f2:49:85:
                    79:82:e8:c8:bb:1a:ab:70:e3:49:97:9f:84:e0:96:
                    c2:6d:41:ab:59:0c:2e:70:9a:2e:11:c8:83:69:4b:
                    f1:19:97:87:c3:76:0e:bb:b0:2c:92:4a:07:03:6f:
                    57:bf:a9:ec:19:85:d6:3d:f8:de:03:7f:1b:9a:2f:
                    6c:02:72:28:b0:69:d5:f9:fb:3d:2e:31:8f:61:50:
                    59:a6:dd:43:4b:89:e9:68:4b:a6:0d:9b:00:0f:9a:
                    94:61:71
                Exponent: 65537 (0x10001)
            X509v3 extensions:
                X509v3 Subject Key Identifier:
                    8B:D6:F8:BD:B7:82:FC:13:BC:61:3F:8B:FA:84:24:3F:A2:14:C8:27
                X509v3 Basic Constraints: critical
                    CA:TRUE
                X509v3 Authority Key Identifier:
                   keyid:8B:D6:F8:BD:B7:82:FC:13:BC:61:3F:8B:FA:84:24:3F:A2:14:C8:27
                   DirName:/C=US/ST=NY/L=New York/O=MongoDB/CN=ROOTCA
                   serial:1E:83:0D:9D:43:75:7C:2B:D6:2A:DC:7E:A2:A2:25:AF:5D:3B:89:43
                X509v3 Key Usage: critical
                   Certificate Sign, CRL Sign
    Signature Algorithm: sha256WithRSAEncryption
        c2:cc:79:40:8b:7b:a1:87:3a:ec:4a:71:9d:ab:69:00:bb:6f:
        56:0a:25:3b:8f:bd:ca:4d:4b:c5:27:28:3c:7c:e5:cf:84:ec:
        2e:2f:0d:37:35:52:6d:f9:4b:07:fb:9b:da:ea:5b:31:0f:29:
        1f:3c:89:6a:10:8e:ae:20:30:8f:a0:cf:f1:0f:41:99:6a:12:
```

```
5f:5c:ce:15:d5:f1:c9:0e:24:c4:81:70:df:ad:a0:e1:0a:cc:
52:d4:3e:44:0b:61:48:a9:26:3c:a3:3d:2a:c3:ca:4f:19:60:
da:f7:7a:4a:09:9e:26:42:50:05:f8:74:13:4b:0c:78:f1:59:
39:1e:eb:2e:e1:e2:6c:cc:4d:96:95:79:c2:8b:58:41:e8:7a:
e6:ad:37:e4:87:d7:ed:bb:7d:fa:47:dd:46:dd:e7:62:5f:e9:
fe:17:4b:e3:7a:0e:a1:c5:80:78:39:b7:6c:a6:85:cf:ba:95:
d2:8d:09:ab:2d:cb:be:77:9b:3c:22:12:ca:12:86:42:d8:c5:
3c:31:a0:ed:92:bc:7f:3f:91:2d:ec:db:01:bd:26:65:56:12:
a3:56:ba:d8:d3:6e:f3:c3:13:84:98:2a:c7:b3:22:05:68:fa:
8e:48:6f:36:8e:3f:e5:4d:88:ef:15:26:4c:b1:d3:7e:25:84:
8c:bd:5b:d2:74:55:cb:b3:fa:45:3f:ee:ef:e6:80:e9:f7:7f:
25:a6:6e:f2:c4:22:f7:b8:40:29:02:f1:5e:ea:8e:df:80:e0:
60:f1:e5:3a:08:81:25:d5:cc:00:8f:5c:ac:a6:02:da:27:c0:
cc:4e:d3:f3:14:60:c1:12:3b:21:b4:f7:29:9b:4c:34:39:3c:
2a:d1:4b:86:cc:c7:de:f3:f7:5e:8f:9d:47:2e:3d:fe:e3:49:
70:0e:1c:61:1c:45:a0:5b:d6:48:49:be:6d:f9:3c:49:26:d8:
8b:e6:a1:b2:61:10:fe:0c:e8:44:2c:33:cd:3c:1d:c2:de:c2:
06:98:7c:92:7b:c4:06:a5:1f:02:8a:03:53:ec:bd:b7:fc:31:
f3:2a:c1:0e:6a:a5:a8:e4:ea:4d:cc:1d:07:a9:3f:f6:0e:35:
5d:99:31:35:b3:43:90:f3:1c:92:8e:99:15:13:2b:8f:f6:a6:
01:c9:18:05:15:2a:e3:d0:cc:45:66:d3:48:11:a2:b9:b1:20:
59:42:f7:88:15:9f:e0:0c:1d:13:ae:db:09:3d:bf:7a:9d:cf:
b2:41:1e:7a:fa:6b:35:20:03:58:a1:6c:02:19:21:5f:25:fc:
ba:2f:fc:79:d7:92:e7:37:77:14:10:d9:33:b6:e5:fb:7a:46:
ab:d1:86:70:88:92:59:c3
```

Создание промежуточного центра сертификации для подписи

Теперь, когда мы создали наш корневой центр сертификации, мы создадим промежуточный центр сертификации для подписи сертификатов участников и клиентов. Промежуточный ЦС – не что иное, как сертификаты, подписанные с использованием нашего корневого сертификата. Рекомендуется использовать промежуточный ЦС для подписи сертификатов сервера (т. е. члена) и клиента. Как правило, ЦС будет использовать разные промежуточные ЦС для подписи сертификатов разных категорий. Если промежуточный ЦС скомпрометирован и сертификат должен быть отозван, это затронет только часть дерева доверия, а не все сертификаты, подписанные ЦС, как в случае, если бы корневой ЦС был использован для подписи всех сертификатов.

```
# Опять же, при работе в производственных системах нужно будет защитить ключ
# подписи с помощью пароля:
# openssl genrsa -aes256 -out signing-ca.key 4096
openssl genrsa -out signing-ca.key 4096

openssl req -new -key signing-ca.key -out signing-ca.csr \
```

```
    -config openssl.cnf -subj "$dn_prefix/CN=CA-SIGNER"
openssl x509 -req -days 730 -in signing-ca.csr -CA root-ca.crt -CAkey \
root-ca.key -set_serial 01 -out signing-ca.crt -extfile openssl.cnf \
-extensions v3_ca
```

Обратите внимание, что в приведенных выше инструкциях мы используем команду `openssl req`, а затем команду `openssl ca`, чтобы подписать наш сертификат подписи, используя корневой сертификат.

Команда `openssl req` создает запрос на подпись, а команда `openssl ca` использует этот запрос в качестве входных данных для создания подписанного промежуточного сертификата.

В качестве последнего шага при создании нашего ЦС мы объединяем корневой сертификат (содержащий наш корневой открытый ключ) и сертификат подписи (содержащий наш открытый ключ для подписи) в единый файл формата pem. Этот файл будет предоставлен нашему серверу *mongod* или клиентскому процессу позже в качестве значения параметра `--tlsCAFile`.

```
cat root-ca.crt > root-ca.pem
cat signing-ca.crt >> root-ca.pem
```

Создание и подпись сертификатов членов

Сертификаты членов обычно называют серверными сертификатами x.509. Используйте этот тип сертификата для сервера *mongod* и процессов *mongos*. Члены кластера MongoDB применяют эти сертификаты для проверки членства в кластере. Проще говоря, один из *mongod* аутентифицируется с другими членами набора реплик, используя сертификат сервера.

Чтобы сгенерировать сертификаты для членов нашего набора реплик, мы будем использовать цикл `for`.

```
# Обратите внимание на фрагмент, касающийся OU, в субъекте в команде
"openssl req" для for host in "${mongodb_server_hosts[@]}"; do
    echo "Generating key for $host"
    openssl genrsa -out ${host}.key 4096
        openssl req -new -key ${host}.key -out ${host}.csr -config openssl.cnf \
        -subj "$dn_prefix/OU=$ou_member/CN=${host}"
        openssl x509 -req -days 365 -in ${host}.csr -CA signing-ca.crt -CAkey \
        signing-ca.key -CAcreateserial -out ${host}.crt -extfile openssl.cnf \
        -extensions v3_req
    cat ${host}.crt > ${host}.pem
    cat ${host}.key >> ${host}.pem
done
```

Три шага связаны с каждым сертификатом:

- используйте команду `openssl genrsa` для создания новой пары ключей;
- используйте команду `openssl req` для генерации запроса на подпись для ключа;
- используйте команду `openssl x509` для подписи и вывода сертификата с использованием подписывающего ЦС.

Обратите внимание на переменную `$ou_member`. Она обозначает разницу между сертификатами сервера и клиентскими сертификатами. Сертификаты сервера и клиента должны быть разными в организационной части уникальных имен. Говоря более конкретно, они должны отличаться по меньшей мере одним из значений – O, OU или DC.

Генерация и подписание клиентских сертификатов

Клиентские сертификаты используются оболочкой mongo, MongoDB Compass, утилитами и инструментами MongoDB и, конечно же, приложениями, использующими драйвер MongoDB. Генерация клиентских сертификатов, по сути, происходит так же, как и создание сертификатов членов. Единственным отличием является использование переменной `$ou_client`. Это гарантирует, что сочетание значений O, OU и DC будет отличаться от значений сертификатов сервера, сгенерированных выше.

```
# Обратите внимание на фрагмент, касающийся OU, в субъекте в команде
"openssl req" для for host in "${mongodb_client_hosts[@]}"; do
    echo "Generating key for $host"
    openssl genrsa -out ${host}.key 4096
    openssl req -new -key ${host}.key -out ${host}.csr -config openssl.cf \
-subj "$dn_prefix/OU=$ou_client/CN=${host}"
    openssl x509 -req -days 365 -in ${host}.csr -CA signing-ca.crt -CAkey \
       signing-ca.key -CAcreateserial -out ${host}.crt -extfile openssl.cf \
       -extensions v3_req
    cat ${host}.crt > ${host}.pem
    cat ${host}.key >> ${host}.pem
done
```

Создание набора реплик без включенной аутентификации и авторизации

Мы можем запустить каждый член нашего набора реплик без включенной аутентификации следующим образом. Ранее при работе с наборами реплик мы не активировали аутентификацию, поэтому это должно выглядеть знакомо. Здесь мы снова используем несколько переменных, которые мы определили в разделе «Создание корневого центра сертификации»

(или *см.* полный сценарий для этой главы) и цикл для запуска каждого члена (*mongod*) нашего набора реплик.

```
mport=$mongodb_port
for host in "${mongodb_server_hosts[@]}"; do
    echo "Starting server $host in non-auth mode"
    mkdir -p ./db/${host}
    mongod --replSet set509 --port $mport --dbpath ./db/$host \
        --fork --logpath ./db/${host}.log
    let "mport++"
done
```

После запуска всех членов мы можем, используя их, инициализировать набор реплик.

```
myhostname=`hostname`
cat > init_set.js <<EOF
rs.initiate();
mport=$mongodb_port;
mport++;
rs.add("localhost:" + mport);
mport++;
rs.add("localhost:" + mport);
EOF
mongo localhost:$mongodb_port init_set.js
```

Обратите внимание, что в приведенном выше коде мы просто создаем последовательность команд, сохраняем эти команды в файле JavaScript, а затем запускаем оболочку *mongo* для выполнения небольшого созданного сценария. Вместе эти команды при выполнении в оболочке *mongo* будут подключаться к серверу *mongod*, работающему на порту 27017 (значение переменной `$mongodb_port` мы установили в разделе «Создание корневого центра сертификации»), инициировать набор реплик, а затем добавят два других сервера (на портах 27018 и 27019) в набор реплик.

Создание пользователя с правами администратора

Теперь мы создадим пользователя с правами администратора на основе одного из клиентских сертификатов, которые мы создали в разделе «Создание и подписание клиентских сертификатов». Мы будем аутентифицироваться как этот пользователь при подключении из оболочки *mongo* или как еще один клиент для выполнения административных задач. Чтобы аутентифицироваться с помощью клиентского сертификата, для начала нужно добавить значение субъекта из сертификата клиента как пользователя MongoDB. Каждый уникальный клиентский сертификат x.509 соответствует одному пользователю MongoDB; то есть нельзя использовать

один сертификат клиента для аутентификации более чем одного пользователя MongoDB. Надо добавить пользователя в базу данных $external; то есть база данных аутентификации – это база данных $external.

Сначала мы получим субъекта из нашего клиентского сертификата с помощью команды *openssl x509*.

```
openssl x509 -in client1.pem -inform PEM -subject -nameopt RFC2253 | grep subject
```

Что должен дать следующий вывод:

```
subject= CN=client1,OU=MyClients,O=MongoDB,L=New York,ST=NY,C=US
```

Чтобы создать пользователя с правами администратора, мы сначала подключимся к первичному узлу нашего набора реплик с помощью оболочки *mongo*.

```
mongo --norc localhost:27017
```

Из оболочки мы выполним следующую команду:

```
db.getSiblingDB("$external").runCommand(
   {
     createUser: "CN=client1,OU=MyClients,O=MongoDB,L=New York,ST=NY,C=US",
     roles: [
         { role: "readWrite", db: 'test'},
         { role: "userAdminAnyDatabase", db: "admin"},
         { role: "clusterAdmin", db: "admin"}
     ],
     writeConcern: {w: "majority", wtimeout: 5000}
   }
);
```

Обратите внимание на использование базы данных $external в этой команде и тот факт, что мы указали субъекта нашего клиентского сертификата в качестве имени пользователя.

Перезапуск набора реплик с включенной аутентификацией и авторизацией

Теперь, когда у нас есть пользователь с правами администратора, мы можем перезапустить набор реплик с включенной аутентификацией и авторизацией и подключиться в качестве клиента. Если бы не было никакого пользователя, подключиться к набору реплик при включенной аутентификации было бы невозможно.

Давайте остановим набор реплик, установленный в его текущей форме (без авторизации):

```
kill $(ps -ef | grep mongod | grep set509 | awk '{print $2}')
```

Теперь мы готовы перезапустить набор реплик с включенной аутентификацией. В производственной среде мы будем копировать каждый файл сертификатов и ключей на соответствующие хосты. Здесь мы делаем все на локальном хосте, чтобы было проще. Чтобы инициировать защищенный набор реплик, мы добавим следующие параметры командной строки для каждого вызова *mongod*:

- `--tlsMode`;
- `--clusterAuthMode`;
- `--tlsCAFile` – файл корневого ЦС (*root-ca.key*);
- `--tlsCertificateKeyFile` – файл сертификата для *mongod*;
- `--tlsAllowInvalidHostnames` – используется только для тестирования; допускает недействительные имена хостов.

Здесь файл, который мы предоставляем в качестве значения параметра `tlsCAFile`, используется для создания цепочки доверия. Как вы помните, файл *root-ca.key* содержит сертификат корневого ЦС, а также подписывающий ЦС. Предоставляя этот файл процессу *mongod*, мы заявляем о своем желании доверять сертификату, содержащемуся в этом файле, а также всем другим сертификатам, подписанным этими сертификатами.

Итак, приступим.

```
mport=$mongodb_port
for host in "${mongodb_server_hosts[@]}"; do
    echo "Starting server $host"
    mongod --replSet set509 --port $mport --dbpath ./db/$host \
       --tlsMode requireTLS --clusterAuthMode x509 --tlsCAFile root-ca.pem \
       --tlsAllowInvalidHostnames --fork --logpath ./db/${host}.log \
       --tlsCertificateKeyFile ${host}.pem --tlsClusterFile ${host}.pem \
       --bind_ip 127.0.0.1
       let "mport++"
    done
```

После этого у нас есть набор реплик из трех членов, защищенный с помощью сертификатов x.509 для аутентификации и шифрования на транспортном уровне. Осталось только подключиться с помощью оболочки mongo. Мы будем использовать сертификат client1 для аутентификации, потому что это тот сертификат, для которого мы создали администратора.

```
mongo --norc --tls --tlsCertificateKeyFile client1.pem --tlsCAFile root-ca.pem \
--tlsAllowInvalidHostnames --authenticationDatabase "\$external" \
--authenticationMechanism MONGODB-X509
```

После подключения мы рекомендуем вам поэкспериментировать, вставив какие-нибудь данные в коллекцию. Вы также должны попытаться подключиться, используя любого другого пользователя (например, `client2.pem`). Попытки подключения приведут к возникновению ошибок, похожих на эти:

```
mongo --norc --tls --tlsCertificateKeyFile client2.pem --tlsCAFile root-ca.pem \
--tlsAllowInvalidHostnames --authenticationDatabase "\$external" \
--authenticationMechanism MONGODB-X509
MongoDB shell version v4.2.0
2019-09-11T23:18:31.696+0100 W  NETWORK  [js] The server certificate does not
match the host name. Hostname: 127.0.0.1 does not match
2019-09-11T23:18:31.702+0100 E  QUERY    [js] Error: Could not find user
"CN=client2,OU=MyClients,O=MongoDB,L=New York,ST=NY,C=US" for db "$external" :
connect@src/mongo/shell/mongo.js:341:17
@(connect):3:6
2019-09-11T23:18:31.707+0100 F  -        [main] exception: connect failed
2019-09-11T23:18:31.707+0100 E  -        [main] exiting with code 1
```

В учебном руководстве этой главы мы рассмотрели пример использования сертификатов x.509 в качестве основы для аутентификации и шифрования обмена данными между клиентами и членами набора реплик. Та же процедура подходит и для разделенных кластеров. Что касается защиты кластера MongoDB, пожалуйста, помните следующее:

- каталоги, корневой ЦС и подписывающий ЦС, а также сам хост, на котором вы генерируете и подписываете сертификаты для членов или клиентов, должны быть защищены от несанкционированного доступа;
- для простоты ключи корневого и подписывающего центров не защищены паролем в этом руководстве. При промышленной эксплуатации необходимо использовать пароли для защиты ключа от несанкционированного использования.

Мы рекомендуем вам скачать предоставленные нами демонстрационные сценарии для этой главы в репозитории GitHub для данной книги и поэкспериментировать с ними.

Глава 20

Долговечность

Долговечность – это свойство СУБД, которое гарантирует, что операции записи, зафиксированные в базе данных, будут существовать постоянно. Например, если система бронирования билетов сообщает, что ваши билеты на концерт были забронированы, ваши места останутся забронированными даже в случае сбоя какой-либо части системы бронирования. Что касается MongoDB, мы должны рассмотреть долговечность на уровне кластера (или, точнее, набора реплик).

В этой главе мы рассмотрим, как:

- MongoDB гарантирует долговечность на уровне членов набора реплик посредством журналирования;
- MongoDB гарантирует долговечность на уровне кластера, используя гарантии записи;
- настроить приложение и кластер MongoDB, чтобы обеспечить необходимый вам уровень долговечности;
- MongoDB гарантирует долговечность на уровне кластера, используя гарантии чтения;
- установить уровень долговечности для транзакций в наборах реплик.

В этой главе мы будем обсуждать долговечность в наборах реплик. Набор реплик из трех членов – это самый базовый кластер, рекомендуемый для производственных приложений. Обсуждение здесь относится к наборам реплик с большим количеством членов и разделенным кластерам.

Долговечность на уровне членов с помощью журналирования

Чтобы обеспечить долговечность в случае сбоя сервера, MongoDB использует журнал упреждающей записи, именуемый *журналом*. Журнал упреждающей записи – широко используемый метод обеспечения долговечности в СУБД. Идея состоит в том, что мы просто записываем представление

изменений, которые должны быть внесены в базу данных, на долговечный носитель (то есть на диск), перед тем как применить эти изменения к самой базе данных. Во многих СУБД журнал упреждающей записи также используется для предоставления свойства атомарности. Однако MongoDB использует другие методы для обеспечения атомарных операций записи.

Начиная с MongoDB версии 4.0, когда приложение выполняет записи в набор реплик, для данных во всех реплицированных коллекциях MongoDB создает записи журнала, используя тот же формат, что и для журнала операций[1]. Как обсуждалось в главе 11, MongoDB использует логическую репликацию на базе журнала операций. Команды в этом журнале являются представлением фактических изменений MongoDB, внесенных в каждый документ, затронутый операцией записи. Таким образом, команды журнала операций легко применить к любому члену набора реплик, независимо от версии, оборудования или любых других различий между членами набора. Кроме того, каждая команда журнала идемпотентна, а это означает, что ее можно применять любое количество раз, и в результате в базе данных всегда будут одинаковые изменения.

Как и большинство СУБД, MongoDB поддерживает находящиеся в памяти представления журнала и файлов данных базы данных. По умолчанию она сбрасывает записи журнала на диск каждые 50 миллисекунд, а файлы базы данных – на диск каждые 60 секунд. 60-секундный интервал для очистки файлов данных называется *контрольной точкой*. Журнал используется для обеспечения долговечности для данных, записанных с момента последней контрольной точки. Что касается гарантий долговечности, если сервер внезапно останавливается, при перезапуске журнал можно использовать для воспроизведения любых записей, которые не были сброшены на диск до завершения работы.

Для файлов журнала MongoDB создает подкаталог с именем *journal* в каталоге *dbPath*. Файлы журнала WiredTiger (подсистема хранения по умолчанию в MongoDB) имеют имена в формате *WiredTigerLog.<последовательность>,* где <*последовательность*> – номер с нулевой добавкой, начиная с *0000000001*. За исключением очень маленьких записей журнала, MongoDB сжимает данные, записанные в журнал. Максимальный размер файла журнала составляет приблизительно 100 МБ. Когда размер файла журнала превысит это ограничение, MongoDB создает новый файл журнала и начинает производить новые записи. Поскольку файлы журнала нужны только для восстановления данных с момента последней контрольной точки, MongoDB автоматически удаляет «старые» файлы журнала, т. е. записанные до самой последней контрольной точки, после записи новой контрольной точки.

[1] MongoDB использует другой формат для записи в локальную базу данных, в которой хранятся данные, применяемые в процессе репликации, и другие специфичные для экземпляра данные, но принципы и применение аналогичны.

Если происходит сбой (или `kill -9`), *mongod* будет воспроизводить файлы журнала при запуске. По умолчанию наибольшая степень потерянных операций записей – те, что были сделаны за последние 100 мс плюс время, необходимое для сброса операций записи журнала на диск.

Если вашему приложению требуется более короткий интервал для сброса журнала, у вас есть два варианта. Один из них – изменение интервала с помощью параметра `--journalCommitInterval` для команды *mongod*. Этот параметр принимает значения в диапазоне от 1 до 500 мс. Другой вариант, который мы рассмотрим в следующем разделе, – указать в гарантии записи, что все операции записи должны записываться на диск. Сокращение интервала записи на диск отрицательно скажется на производительности, поэтому вы должны быть уверены в последствиях для своих приложений, прежде чем изменять журналирование по умолчанию.

Долговечность на уровне кластера при использовании гарантии записи

Что касается гарантии записи, вы можете указать, какой уровень подтверждения требуется вашему приложению в ответ на запросы на запись. В наборе реплик сбои в работе сети, сервера или простои центров обработки данных могут препятствовать репликации операций записи на каждого члена или даже большинство членов. Когда нормальное состояние восстанавливается в набор реплик, возможно, что операции записи, не реплицированные на большинство членов, будут отменены. В таких ситуациях клиенты и база данных могут иметь разное представление о том, какие данные были зафиксированы.

Существуют приложения, для которых в некоторых случаях откат операций записи может быть допустим. Например, можно было бы откатить небольшое количество комментариев в каком-нибудь приложении социальной сети. MongoDB поддерживает ряд гарантий долговечности на уровне кластера, что позволяет разработчикам приложений выбирать уровень долговечности, который лучше всего подходит для их случая использования.

Опции w и wtimeout для параметра writeConcern

Язык запросов MongoDB поддерживает указание гарантии записи для всех методов вставки и обновления. В качестве примера предположим, что у нас имеется приложение для онлайн-торговли, и мы хотим убедиться, что все заказы долговечны. Запись заказа в базу данных может выглядеть примерно так:

```
try {
    db.products.insertOne(
```

```
        { sku: "H1100335456", item: "Electric Toothbrush Head", quantity: 3 },
        { writeConcern: {w: "majority", wtimeout : 100 } }
    );
}
catch (e) {
    print (e);
}
```

Все методы вставки и обновления принимают второй параметр – документ. В этом документе вы можете указать значение для `writeConcern`. В предыдущем примере указанная нами гарантия записи сообщает о том, что мы хотим видеть подтверждение от сервера относительно того, что операция записи завершилась успешно, только если операция была успешно реплицирована в большинство членов набора реплик нашего приложения. Кроме того, операция записи должна возвращать ошибку, если она не реплицируется на большинство членов набора реплик в течение 100 мс или меньше. В случае такой ошибки MongoDB не отменяет успешные изменения данных, выполненные до того, как гарантия записи превысила ограничение по времени, – приложению решать, как обрабатывать тайм-ауты в таких ситуациях. В общем, вы должны настроить значение `wtimeout` так, чтобы приложение испытывало тайм-ауты только в необычных обстоятельствах, и любые действия, предпринимаемые вашим приложением в ответ на ошибку тайм-аута, обеспечат правильное состояние ваших данных. В большинстве случаев ваше приложение должно попытаться определить, был ли тайм-аут результатом временного замедления сетевых коммуникаций или произошло нечто более значительное.

В качестве значения опции `w` в документе гарантии записи можно указать `"majority"` (как было сделано в этом примере). Как альтернативу можно указать целое число от нуля до количества членов в наборе реплик. Наконец, можно пометить члены набора реплик, скажем, чтобы идентифицировать тех, кто находится на твердотельных накопителях по сравнению с вращающимися дисками, или тех, что используются для создания отчетов по сравнению с рабочими нагрузками OLTP. В качестве значения опции `w` можно указать набор тегов, чтобы гарантировать, что операции записи будут подтверждены только после фиксации хотя бы одного члена набора реплик, соответствующего указанному набору тегов.

Опция j (ведение журнала) для параметра writeConcern

Помимо предоставления значения опции `w`, вы также можете запросить подтверждение того, что операция записи была записана в журнал, используя опцию `j` в документе гарантии записи. Если для этой опции установлено значение `true`, MongoDB подтверждает успешную операцию записи только после того, как запрошенное число членов (значение `w`) записали

операцию в свой журнал на диске. Продолжим. Если мы хотим, чтобы все операции записи фиксировались в журнале для большинства участников, можно обновить код, как показано ниже:

```
try {
    db.products.insertOne(
       { sku: "H1100335456", item: "Electric Toothbrush Head", quantity: 3 },
       { writeConcern: {w: "majority", wtimeout: 100, j: true } }
    );
}
catch (e) {
    print (e);
}
```

Не дожидаясь журналирования, существует короткое окно длительностью около 100 мс для каждого члена, когда при сбое серверного процесса или оборудования запись может быть потеряна. Однако ожидание журналирования перед подтверждением операций записи членам набора реплик снижает производительность.

Очень важно, чтобы при решении проблем долговечности ваших приложений вы тщательно оценивали требования, предъявляемые к вашему приложению, и взвешивали влияние на производительность выбранных вами параметров долговечности.

Долговечность на уровне кластера при использовании гарантии чтения

В MongoDB гарантии чтения позволяют конфигурировать время чтения результатов. Это может позволить клиентам увидеть результаты записи до того, как эти записи станут долговечными. Гарантию чтения можно использовать с гарантией записи для контроля над уровнем согласованности и гарантией доступности, предоставляемых приложению. Их не следует путать с предпочтениями чтения, которые касаются того, откуда данные считываются; в частности, предпочтения чтения определяют членов, несущих данные, в наборе реплик. Настройка по умолчанию – чтение с первичного узла.

Гарантии чтения определяют свойства согласованности и изоляции читаемых данных. По умолчанию уровень опции `readConcern` – это `local`, поэтому данные возвращаются без каких-либо гарантий того, что они были записаны в большинство членов набора реплик, несущих данные. Это может привести к откату данных в будущем. Гарантия чтения `majority` возвращает только надежные данные (отката данных не будет), которые были подтверждены большинством членов набора реплик. В MongoDB

версии 3.4 была добавлена гарантия чтения `linearizable`. Она гарантирует, что возвращаемые данные отражают все успешные операции записи, признанные большинством, которые были завершены до начала операции чтения, и может ожидать завершения одновременного выполнения операций записи, прежде чем предоставлять результаты.

Таким же образом, когда речь идет о гарантиях записи, вам нужно будет сопоставить влияние производительности гарантий чтения с гарантиями долговечности и изоляции, которые они предоставляют, прежде чем выбрать подходящий вариант для своего приложения.

Долговечность транзакций с использованием гарантии записи

В MongoDB операции с отдельными документами являются атомарными. Вы можете использовать встроенные документы и массивы для выражения отношений между сущностями в одном документе, вместо того чтобы использовать нормализованную модель данных, разделяющую сущности и связи между несколькими коллекциями. В результате многим приложениям не нужны многодокументные транзакции.

Однако для случаев использования, которые требуют атомарности для обновлений нескольких документов, MongoDB предоставляет возможность выполнять транзакции с несколькими документами в отношении наборов реплик. Многодокументные транзакции могут использоваться в нескольких операциях, документах, коллекциях и базах данных.

Транзакции требуют, чтобы все изменения данных в транзакции были успешными. В случае сбоя какой-либо операции транзакция прерывается, и все изменения данных отбрасываются. Если все операции успешны, все изменения данных, сделанные в транзакции, сохраняются, и записи становятся видимыми для последующего чтения.

Как и в случае с отдельными операциями записи, можно указать гарантию записи для транзакций, которая устанавливается на уровне транзакции, а не на уровне отдельной операции. Во время фиксации транзакции используют гарантию записи на уровне транзакций для фиксации операций записи. Гарантии записи, установленные для отдельных операций внутри транзакции, будут игнорироваться.

Вы можете установить гарантию записи для фиксации транзакции в начале транзакции. Гарантия записи, равная 0, не поддерживается для транзакций. Если вы используете гарантию записи, равную 1, ее можно откатить в случае обработки отказа. Вы можете использовать гарантию записи `"majority"` для обеспечения долговечности транзакций перед лицом сбоев сети и сервера, которые могут вызвать обработку отказа в наборе реплик. Вот пример:

```
function updateEmployeeInfo(session)
{
   employeesCollection = session.getDatabase("hr").employees;
   eventsCollection = session.getDatabase("reporting").events;

   session.startTransaction( { writeConcern: { w: "majority" } } );

   try {
        employeesCollection.updateOne( { employee: 3},
                                       { $set: { status: "Inactive"}});
        eventsCollection.insertOne( { employee: 3, status: { new: "Inactive",
                                     old: "Active" } } );
    }
    catch (error) {
        print("Caught exception during transaction, aborting.");
        session.abortTransaction();
        throw error;
    }
    commitWithRetry(session);
}
```

Чего MongoDB не гарантирует

Есть несколько ситуаций, когда MongoDB не может гарантировать долговечность, например если есть проблемы с оборудованием или ошибки файловой системы. В частности, если жесткий диск поврежден, MongoDB ничего не сможет сделать для защиты ваших данных.

Кроме того, различные варианты аппаратного и программного обеспечения могут иметь разные гарантии долговечности. Например, некоторые более дешевые или старые жесткие диски сообщают об успешности операции записи, когда операция ставится в очередь для записи, а не тогда, когда она действительно была записана. MongoDB не может защитить от искажения информации на этом уровне: в случае сбоя системы данные могут быть потеряны.

По сути, MongoDB безопасна только в качестве базовой системы: если аппаратное обеспечение или файловая система уничтожает данные, MongoDB не может предотвратить это. Используйте репликацию для защиты от системных проблем. Если одна машина выйдет из строя, будем надеяться, что другая будет работать правильно.

Проверка на предмет наличия повреждений

Команду `validate` можно использовать для проверки коллекции на наличие повреждений. Чтобы выполнить эту команду для коллекции *movies*:

```
db.movies.validate( { full: true } )
{
    "ns" : "sample_mflix.movies",
    "nInvalidDocuments" : NumberLong(0),
    "nrecords" : 45993,
    "nIndexes" : 5,
    "keysPerIndex" : {
        "_id_" : 45993,
        "$**_text" : 3671341,
        "genres_1_imdb.rating_1_metacritic_1" : 94880,
        "tomatoes_rating" : 45993,
        "getMovies" : 45993
    },
    "indexDetails" : {
        "$**_text" : {
                "valid" : true
        },
        "_id_" : {
                "valid" : true
        },
        "genres_1_imdb.rating_1_metacritic_1" : {
                "valid" : true
        },
        "getMovies" : {
                "valid" : true
        },
        "tomatoes_rating" : {
                "valid" : true
        }
    },
    "valid" : true,
    "warnings" : [ ],
    "errors" : [ ],
    "extraIndexEntries" : [ ],
    "missingIndexEntries" : [ ],
    "ok" : 1
}
```

Основное поле, которое вы ищете, – `"valid"`, что, будем надеяться, будет правдой. Если это не так, команда `validate` предоставит сведения о найденном повреждении.

Большая часть вывода этой команды описывает внутренние структуры коллекции и временные метки, используемые для понимания порядка операций в кластере. Они не особенно полезны для отладки. (*См.* приложение B для получения дополнительной информации о внутреннем устройстве коллекции.)

Эту команду можно выполнять только для коллекций, и она также проверит ассоциированные индексы в поле `indexDetails`. Тем не менее для этого нужна более тщательная проверка, которая настраивается с помощью параметра `{ full: true }`.

Часть VI

Администрирование сервера

Глава 21

Настройка MongoDB в рабочем окружении

Во второй главе мы рассмотрели основы запуска MongoDB. В этой главе будет более подробно рассказано о том, какие параметры важны для настройки MongoDB в рабочей среде, включая:

- обычно используемые параметры;
- запуск и выключение MongoDB;
- параметры, связанные с безопасностью;
- особенности журналирования.

Запуск из командной строки

Сервер MongoDB запускается с исполняемого файла `mongod`, у которого имеется множество настраиваемых параметров запуска; чтобы просмотреть их все, выполните команду `mongod` с параметром `--help` из командной строки. Есть несколько параметров, которые широко используются, и о них важно знать:

`--dbpath`
: Указываем альтернативный каталог для использования в качестве каталога данных; по умолчанию это */data/db/* (или, в Windows, *\data\db* на томе двоичного файла MongoDB). Каждому процессу `mongod` на машине нужен свой собственный каталог данных, поэтому если вы запускаете три экземпляра `mongod` на одном компьютере, вам потребуются три отдельных каталога данных. При запуске `mongod` в его каталоге данных создается файл *mongod.lock*, что не позволяет любому другому процессу `mongod` использовать этот каталог. Если вы попытаетесь запустить еще один сервер MongoDB, используя тот же каталог данных, он выдаст ошибку:

```
exception in initAndListen: DBPathInUse: Unable to lock the
    lock file: \ data/db/mongod.lock (Resource temporarily unavailable).
```

```
Another mongod instance is already running on the data/db directory,
\ terminating
```

`--port`

Указываем номер порта для сервера для прослушивания. По умолчанию `mongod` использует порт 27017, который вряд ли будет использоваться еще одним процессом (кроме других процессов `mongod`). Если вы хотите запустить несколько процессов `mongod` на одной машине, вам нужно будет указать разные порты для каждой из них. Если вы попытаетесь запустить `mongod` на порту, который уже используется, он выдаст ошибку:

```
Failed to set up listener: SocketException: Address already in use.
```

`--fork`

В системах на базе Unix отсоединяет процесс от терминала, запуская MongoDB в качестве демона. Если вы запускаете `mongod` впервые (с пустым каталогом данных), файловой системе может потребоваться несколько минут для выделения файлов базы данных. Родительский процесс не вернется, пока не будет выполнено предварительное распределение, и `mongod` не готов, чтобы начать принимать соединения. Таким образом, может казаться, что отсоединение зависло. Вы можете следить за журналом, чтобы увидеть, что он делает. Нужно использовать параметр `--logpath`, если указали параметр `--fork`.

`--logpath`

Отправляем весь вывод в указанный файл, вместо того чтобы выводить его в командной строке. Будет создан файл, если он не существует, при условии что у вас есть права на запись в каталог. Он также перезапишет файл журнала, если он уже существует, удалив все старые записи журнала. Если вы хотите сохранить старые журналы, используйте параметр `--logappend` в дополнение к параметру `--logpath` (настоятельно рекомендуется).

`--directoryperdb`

Помещаем каждую базу данных в свой каталог. Это позволяет монтировать разные базы данных на разных дисках, если это необходимо или желательно. Обычно используется для размещения локальной базы данных на своем диске (репликация) или перемещения базы данных на другой диск, если исходный заполнен. Вы также можете разместить базы данных, которые обрабатывают большую нагрузку на более быстрых дисках, и базы данных с меньшей нагрузкой на более медленных дисках. Это в основном дает вам больше гибкости, чтобы заниматься перемещениями позже.

`--config`
: Используем конфигурационный файл для дополнительных параметров, не указанных в командной строке. Обычно используется, чтобы удостовериться, что параметры одинаковы между перезапусками. Подробнее *см.* раздел «Конфигурирование на базе файлов».

Например, чтобы запустить сервер в качестве демона, слушающего на порту 5586 и отправляющего весь вывод в *mongodb.log*, мы могли бы запустить это:

```
$ ./mongod --dbpath data/db --port 5586 --fork --logpath
    mongodb.log --logappend 2019-09-06T22:52:25.376-0500 I CONTROL [main]
    Automatically disabling TLS 1.0, \ to force-enable TLS 1.0 specify
    --sslDisabledProtocols 'none' about to fork child process, waiting until
    server is ready for connections. forked process: 27610 child process
    started successfully, parent exiting
```

Когда вы впервые устанавливаете и запускаете MongoDB, рекомендуется посмотреть журнал. Это можно легко упустить, особенно если MongoDB запускается из сценария инициализации, но в журнале часто содержатся важные предупреждения, которые предотвращают возникновение последующих ошибок. Если вы не видите никаких предупреждений в журнале MongoDB при запуске, то у вас все готово. (Предупреждения при запуске также появятся при запуске оболочки.) Если на баннере запуска есть какие-либо предупреждения, примите их к сведению. MongoDB будет предупреждать вас о множестве проблем: что вы работаете на 32-битном компьютере (для которого MongoDB не предназначен), что у вас включена поддержка NUMA (что может замедлить обход содержимого) или что ваша система не допускает достаточного количества открытых файловых дескрипторов (MongoDB использует много файловых дескрипторов).

Преамбула журнала не изменится при перезапуске базы данных, поэтому не стесняйтесь запускать MongoDB из сценария инициализации и игнорировать журналы, как только вы узнаете, что в них говорится. Тем не менее рекомендуется делать проверку каждый раз, когда вы выполняете установку, обновление или восстановление после сбоя, просто чтобы убедиться, что MongoDB и ваша система находятся на одной волне.

Когда вы запускаете базу данных, MongoDB запишет в коллекцию *local.startup_log* документ, описывающий версию MongoDB, базовую систему и используемые флаги. Мы можем посмотреть на этот документ, используя оболочку *mongo*:

```
> use local
switched to db local
> db.startup_log.find().sort({startTime: -1}).limit(1).pretty()
```

```
{
    "_id": "server1-1544192927184",
    "hostname": "server1.example.net",
    "startTime": ISODate("2019-09-06T22:50:47Z"),
    "startTimeLocal": "Fri Sep 6 22:57:47.184",
    "cmdLine": {
        "net": {
            "port": 5586
        },
        "processManagement": {
            "fork": true
        },
        "storage": {
            "dbPath": "data/db"
        },
        "systemLog": {
            "destination": "file",
            "logAppend": true,
            "path": "mongodb.log"
        }
    },
    "pid": NumberLong(27278),
    "buildinfo": {
        "version": "4.2.0",
        "gitVersion": "a4b751dcf51dd249c5865812b390cfd1c0129c30",
        "modules": [
            "enterprise"
        ],
        "allocator": "system",
        "javascriptEngine": "mozjs",
        "sysInfo": "deprecated",
        "versionArray": [
            4,
            2,
            0,
            0
        ],
        "openssl": {
            "running": "Apple Secure Transport"
        },
        "buildEnvironment": {
            "distmod": "",
            "distarch": "x86_64",
            "cc": "gcc: Apple LLVM version 8.1.0 (clang-802.0.42)",
```

```
            "ccflags": "-mmacosx-version-min=10.10 -fno-omit\
                        -frame-pointer -fno-strict-aliasing \
                        -ggdb -pthread -Wall
                        -Wsign-compare -Wno-unknown-pragmas \
                        -Winvalid-pch -Werror -O2 -Wno-unused\
                        -local-typedefs -Wno-unused-function
                        -Wno-unused-private-field \
                        -Wno-deprecated-declarations \
                        -Wno-tautological-constant-out-of\
                        -range-compare
                        -Wno-unused-const-variable -Wno\
                        -missing-braces -Wno-inconsistent\
                        -missing-override
                        -Wno-potentially-evaluated-expression \
                        -Wno-exceptions -fstack-protector\
                        -strong -fno-builtin-memcmp",
            "cxx": "g++: Apple LLVM version 8.1.0 (clang-802.0.42)",
            "cxxflags": "-Woverloaded-virtual -Werror=unused-result \
                        -Wpessimizing-move -Wredundant-move \
                        -Wno-undefined-var-template -stdlib=libc++ \
                        -std=c++14",
            "linkflags": "-mmacosx-version-min=10.10 -Wl, \
                        -bind_at_load -Wl,-fatal_warnings \
                        -fstack-protector-strong \
                        -stdlib=libc++",
            "target_arch": "x86_64",
            "target_os": "macOS"
        },
        "bits": 64,
        "debug": false,
        "maxBsonObjectSize": 16777216,
        "storageEngines": [
            "biggie",
            "devnull",
            "ephemeralForTest",
            "inMemory",
            "queryable_wt",
            "wiredTiger"
        ]
    }
}
```

Данная коллекция может быть полезна для отслеживания обновлений и изменений в поведении.

Конфигурирование на базе файлов

MongoDB поддерживает чтение информации о конфигурировании из файла. Это может быть полезно, если у вас есть большой набор опций, которые вы хотите использовать, или вы автоматизируете задачу запуска MongoDB. Чтобы указать серверу получать параметры из конфигурационного файла, используйте флаги `-f` или `--config`. Например, выполните команду `mongod --config ~/.mongodb.conf`, чтобы использовать *~/.mongodb.conf* в качестве конфигурационного файла.

Параметры, поддерживаемые в конфигурационном файле, совпадают с параметрами, принятыми в командной строке. Тем не менее формат отличается. Начиная с версии 2.6 конфигурационные файлы MongoDB используют формат YAML. Вот пример файла:

```
systemLog:
   destination: file
   path: "mongod.log"
   logAppend: true
storage:
   dbPath: data/db
processManagement:
   fork: true
net:
   port: 5586
...
```

В этом файле указаны те же параметры, которые мы использовали ранее при запуске с обычными аргументами командной строки. Обратите внимание, что эти же параметры отражены в документе коллекции *startup_log*, который мы рассмотрели в предыдущем разделе. Единственное реальное отличие состоит в том, что параметры указываются с использованием формата JSON, а не YAML.

В MongoDB 4.2 были добавлены директивы расширения, чтобы разрешить загрузку определенных параметров конфигурационных файлов или загрузку всего файла конфигурации. Преимущество директив расширения заключается в том, что конфиденциальная информация, такая как пароли и сертификаты безопасности, не должна храниться в файле конфигурации напрямую. Параметр командной строки `--configExpand` активирует эту функцию и должен включать в себя директивы расширения, которые вы хотите активировать. `__rest` и `__exec` – текущая реализация директив расширения в MongoDB. Директива расширения `__rest` загружает определенные значения файла конфигурации или весь файл конфигурации из конечной точки REST. Директива расширения `__exec` загружает определенные значения файла конфигурации или весь файл конфигурации из команды оболочки или терминала.

Остановка MongoDB

Возможность безопасно остановить работающий сервер MongoDB так же важна, как и возможность его запустить. Есть несколько разных способов сделать это эффективно.

Самый простой способ отключить работающий сервер – использовать команду `shutdown`: `{"shutdown" : 1}`. Это команда администратора, и она должна выполняться в базе данных *admin*. Оболочка имеет вспомогательную функцию, чтобы это было проще сделать:

```
> use admin
switched to db admin
> db.shutdownServer()
server should be down...
```

При запуске на первичном узле команда `shutdown` понижает первичный узел в ранге и ждет, пока вторичный узел не синхронизируется, прежде чем завершить работу сервера. Это сводит к минимуму вероятность отката, но отключение не гарантируется. Если нет доступного вторичного узла, который может синхронизироваться в течение нескольких секунд, команда `shutdown` потерпит неудачу, и (прежний) первичный узел не будет отключен:

```
> db.shutdownServer()
{
    "closest": NumberLong(1349465327),
    "difference": NumberLong(20),
    "errmsg": "no secondaries within 10 seconds of my optime",
    "ok": 0
}
```

Можно принудительно заставить команду `shutdown` завершить работу первичного узла, используя параметр `force`:

```
db.adminCommand({"shutdown" : 1, "force" : true})
```

Это эквивалентно отправке сигнала SIGINT или SIGTERM (все три этих параметра приводят к чистому отключению, но могут быть нереплицированные данные). Если сервер работает как основной процесс в терминале, SIGINT можно отправить, нажав сочетание клавиш **Ctrl**-**C**. В противном случае для отправки сигнала можно использовать команду типа `kill`. Если бы у *mongod* в качестве идентификатора процесса стояло 10014, команда выглядела бы так: `kill -2 10014` (SIGINT) или `kill 10014` (SIGTERM).

Когда *mongod* получит сигнал SIGINT или SIGTERM, он выполнит чистое отключение. Это означает, что он будет ожидать завершения любых вы-

полняющихся операций или предварительных выделений файлов (это может занять некоторое время), закроет все открытые соединения, сбросит все данные на диск и остановится.

Не устанавливайте публично адресуемые серверы MongoDB. Вы должны максимально ограничить доступ между внешним миром и MongoDB. Лучший способ сделать это – настроить межсетевые экраны и разрешить доступ к MongoDB только на внутренних сетевых адресах. Глава 24 посвящена тому, какие соединения необходимо разрешить между серверами и клиентами MongoDB.

Помимо брандмауэров, есть несколько параметров, которые можно добавить в свой конфигурационный файл, чтобы сделать его более безопасным:

`--bind_ip`
: Указывает интерфейсы, которые необходимо слушать в MongoDB. Как правило нужно чтобы это был внутренний IP-адрес: к которому могут обращается серверы приложений и другие члены вашего кластера, но недоступные для внешнего мира. *Локальный хост* подходит для процессов *mongos*, если на одном сервере запущен и сервер, и приложение. Когда речь идет о конфигурационных серверах и шардах, нужно иметь возможность обращаться к ним с других машин, поэтому придерживайтесь адресов типа non-localhost.

  Начиная с MongoDB версии 3.6 процессы *mongod* и *mongos* по умолчанию привязаны к локальному хосту. При подключении только к локальному хосту процессы *mongod* и *mongos* будут принимать подключения лишь от клиентов, работающих на одном и том же компьютере. Это помогает ограничить доступ незащищенных экземпляров MongoDB. Для привязки к другим адресам используйте параметр конфигурационного файла `net.bindIp` либо параметр командной строки `--bind_ip`, чтобы указать список имен хостов или IP-адресов.

`--nounixsocket`
: Отключите прослушивание на сокете домена UNIX. Если вы не планируете подключаться через сокет файловой системы, можете запретить его. Вы можете подключиться только через сокет файловой системы на компьютере, где также работает сервер приложений.

`--noscripting`
: Отключите выполнение сценариев JavaScript на стороне сервера. Некоторые проблемы безопасности, о которых сообщалось в MongoDB, были связаны с JavaScript, поэтому, как правило, безопаснее будет запретить это, если позволяет ваше приложение.

Некоторые вспомогательные функции оболочки предполагают, что JavaScript доступен на сервере, особенно функция `sh.status()`. Если вы попытаетесь выполнить любую из этих функций с отключенным JavaScript, то увидите ошибки.

Шифрование данных

Шифрование данных доступно в MongoDB Enterprise. Эти параметры не поддерживаются в версии для сообщества MongoDB.

Процесс шифрования данных включает в себя следующие этапы:

- генерирование мастер-ключа;
- генерирование ключей для каждой базы данных;
- шифрование данных с помощью ключей базы данных;
- шифрование ключей базы данных с помощью мастер-ключа.

При использовании шифрования данных все файлы данных шифруются в файловой системе. Данные не шифруются только в памяти и во время передачи. Для шифрования всего сетевого трафика MongoDB можно использовать протокол TLS/SSL. Параметры шифрования данных, которые пользователи MongoDB Enterprise могут добавить в свои конфигурационные файлы:

`--enableEncryption`
: Включает шифрование в подсистеме хранения WiredTiger. С помощью этой опции данные, хранящиеся в памяти и на диске, будут зашифрованы. Иногда это называют «шифрованием в состоянии покоя». Вы должны установить для него значение `true`, чтобы передать ключи шифрования и настроить шифрование. По умолчанию параметр имеет значение `false`.

`--encryptionCipherMode`
: Устанавливает режим шифрования для шифрования в состоянии покоя в WiredTiger. Доступны два режима: AES256-CBC и AES256-GCM. AES256-CBC – аббревиатура выражения *256-bit Advanced Encryption Standard in Cipher Block Chaining Mode*. AES256-GCM использует Galois/Counter Mode (режим счетчика с аутентификацией Галуа). Оба являются стандартными режимами шифрования. Начиная с MongoDB версии 4.0 MongoDB Enterprise больше не поддерживает AES256-GCM для Windows.

`--encryptionKeyFile`
: Укажите путь к локальному файлу ключей, если вы управляете ключами с помощью процесса, отличного от Протокола совместным управлением ключами (KMIP).

MongoDB Enterprise также поддерживает управление ключами с помощью этого протокола. Обсуждение Протокола совместным управлением ключами выходит за рамки этой книги. Пожалуйста, просмотрите документацию по MongoDB для получения подробной информации об использовании пртокола KMIP с MongoDB (https://docs.mongodb.com/manual/tutorial/configure-encryption/).

SSL-соединения

Как мы видели в главе 18, MongoDB поддерживает шифрование транспорта с использованием протоколов TLS и SSL. Данная функция доступна во всех выпусках MongoDB. По умолчанию соединения с MongoDB передают данные в незашифрованном виде. Однако протокол TLS/SSL обеспечивает шифрование транспорта. MongoDB использует собственные библиотеки TSL/SSL, доступные в вашей операционной системе. Используйте параметр `--tlsMode` и связанные параметры для настройки протокола TLS/SSL. Обратитесь к главе 18, чтобы получить более подробную информацию, и ознакомьтесь с документацией вашего драйвера относительно того, как создавать соединения TLS/SSL с использованием вашего языка.

Протоколирование

По умолчанию *mongod* отправляет свои журналы на stdout. Большинство сценариев инициализации используют параметр `--logpath` для отправки журналов в файл. Если у вас несколько экземпляров MongoDB на одном компьютере (например, *mongod* и *mongos*), убедитесь, что их журналы хранятся в отдельных файлах. Убедитесь, что вы знаете, где находятся журналы, и что у вас есть доступ для чтения к этим файлам.

MongoDB выдает большое количество сообщений журналов, но, пожалуйста, не используйте для запуска параметр `--quiet` (это будет подавлять некоторые из них). Оставьте уровень журналов по умолчанию. Как правило, это идеальный вариант: информации для базовой отладки достаточно (почему так медленно, почему не запускается и т. д.), но журналы не занимают слишком много места.

Если вы заняты отладкой конкретной проблемы, связанной с вашим приложением, существует несколько вариантов для получения дополнительной информации из журналов. Вы можете изменить уровень протоколирования, выполнив команду `setParameter`, или настроить уровень во время запуска, передав его в виде строки с помощью параметра `--setParameter`.

```
> db.adminCommand({"setParameter" : 1, "logLevel" : 3})
```

Вы также можете изменить уровень протоколирования для конкретного компонента. Это полезно, если вы отлаживаете определенный аспект ва-

шего приложения и вам требуется дополнительная информация, но только от этого компонента. В данном примере мы устанавливаем по умолчанию уровень детализации протоколирования на 1, а уровень детализации компонента запроса на 2:

```
> db.adminCommand({"setParameter" : 1, logComponentVerbosity:
    { verbosity: 1, query: { verbosity: 2 }}})
```

Не забудьте уменьшить уровень протоколирования до 0, когда закончите отладку, иначе ваши журналы могут стать излишне шумными. Вы можете установить уровень до 5, после чего `mongod` будет выводить практически все необходимые действия, включая содержимое каждого обработанного запроса. Это может привести к появлению большого количества операций ввода/вывода, поскольку *mongod* записывает все в файл журнала, что может замедлить работу занятой системы. Включение профилирования – лучший вариант, если вам нужно видеть каждую операцию по мере ее совершения.

По умолчанию MongoDB регистрирует информацию о запросах, выполнение которых занимает более 100 мс. Если 100 мс слишком мало или слишком долго для вашего приложения, можно изменить порог с помощью функции `setProfilingLevel`:

```
> //Протоколируем только те запросы, которые занимают более 500 мс;
> db.setProfilingLevel(1, 500)
{ "was" : 0, "slowms" : 100, "ok" : 1 }
> db.setProfilingLevel(0)
{ "was" : 1, "slowms" : 500, "ok" : 1 }
```

Вторая строка отключит профилирование, но значение в миллисекундах, указанное в первой строке, будет по-прежнему использоваться в качестве порогового значения для протоколирования (во всех базах данных). Вы также можете установить этот параметр, перезапустив MongoDB с параметром `--slowms`.

Наконец, настройте задание cron, которое будет ротировать ваш журнал каждый день или неделю. Если MongoDB была запущена с использованием параметра `--logpath`, отправка процессу сигнала SIGUSR1 заставит его ротировать журнал. Также существует команда `logRotate`, которая делает то же самое:

```
> db.adminCommand({"logRotate" : 1})
```

Нельзя ротировать журналы, если при MongoDB не использовался параметр `--logpath`.

Глава 22

Мониторинг MongoDB

Перед развертыванием важно настроить мониторинг. Мониторинг должен позволить вам отслеживать, что делает ваш сервер, и предупреждать вас, если что-то пойдет не так. В этой главе рассказывается о том, как:

- отслеживать использование памяти в MongoDB;
- отслеживать показатели производительности приложений;
- диагностировать проблемы, связанные с репликацией.

Мы будем использовать примеры графиков из Ops Manager, чтобы продемонстрировать, на что обращать внимание при мониторинге (*см.* инструкции по установке Ops Manager по адресу https://oreil.ly/D4751). Возможности мониторинга в MongoDB Atlas (облачной базе данных MongoDB) очень похожи. MongoDB также предлагает бесплатный сервис для мониторинга, который отслеживает работу автономных серверов и наборов реплик. Он сохраняет данные мониторинга в течение 24 часов после их загрузки и предоставляет приблизительную статистику по времени выполнения операций, использованию памяти и процессора и количеству операций.

Если вы не хотите использовать Ops Manager, Atlas или бесплатный сервис мониторинга MongoDB, используйте какой-либо тип мониторинга. Это поможет вам обнаружить потенциальные проблемы, прежде чем они вызовут проблемы, и диагностировать их, когда они возникнут.

Мониторинг использования памяти

Доступ к данным в памяти осуществляется быстро, а доступ к данным на диске – медленно. К сожалению, память дорогая (а диск дешевый), и, как правило, MongoDB использует память раньше любого другого ресурса. В этом разделе рассказывается о том, как отслеживать взаимодействие MongoDB с ЦП, диском и памятью и что нужно отслеживать.

Знакомство с памятью компьютера

Компьютеры, как правило, имеют небольшой объем быстродействующей памяти и большой объем медленно работающего диска. Когда вы за-

прашиваете страницу данных, которая хранится на диске (и пока еще не в памяти), ваша системная страница дает сбой и копирует страницу с диска в память. Затем она может очень быстро получить доступ к странице в памяти. Если ваша программа перестает регулярно пользоваться страницей, а ваша память заполняется другими страницами, старая страница будет удалена из памяти и снова будет находиться только на диске.

Копирование страницы с диска в память занимает намного больше времени, чем чтение страницы из памяти. Таким образом, чем меньше MongoDB копирует данные с диска, тем лучше. Если MongoDB может работать почти полностью в памяти, он сможет получить доступ к данным намного быстрее. Таким образом, использование памяти в MongoDB является одной из самых важных характеристик для отслеживания.

Отслеживание использования памяти

MongoDB сообщает о трех «типах» памяти в Ops Manager. Это резидентная память, виртуальная память и расширенная. Резидентная память – это память, которой MongoDB явно владеет в ОЗУ. Например, если вы запрашиваете документ и он выгружается в память, эта страница добавляется в резидентную память MongoDB.

MongoDB дается адрес этой страницы. Он не является буквальным адресом страницы в ОЗУ; это виртуальный адрес. MongoDB может передать его ядру, и ядро будет искать, где на самом деле находится страница. Таким образом, если ядру необходимо удалить страницу из памяти, MongoDB по-прежнему сможет использовать адрес для доступа к ней. MongoDB запросит память у ядра, ядро просмотрит свой кеш страниц, увидит, что страницы там нет, далее последует отказ страницы, чтобы скопировать страницу в память и вернуть ее в MongoDB.

Если ваши данные полностью помещаются в память, резидентная память должна приблизительно соответствовать размеру ваших данных. Когда мы говорим о том, что данные находятся «в памяти», мы всегда имеем в виду, что данные находятся в оперативной памяти.

Расширяемая память включает в себя все данные, к которым когда-либо обращалась MongoDB (все страницы данных, для которых у нее есть адреса). Обычно она будет размером с ваш набор данных.

Виртуальная память – это абстракция, предоставляемая операционной системой, которая скрывает детали физического хранилища от программного процесса. Каждый процесс видит непрерывное адресное пространство памяти, которое он может использовать. В Ops Manager использование виртуальной памяти MongoDB обычно в два раза больше размера расширяемой памяти.

На рис. 22.1 показан график Ops Manager с информацией о памяти, который описывает, сколько виртуальной, резидентной и расширяемой

памяти использует MongoDB. Расширяемая память актуальна только для более старых (до 4.0) развертываний с использованием подсистемы хранения MMAP. Теперь, когда MongoDB применяет подсистему хранения WiredTiger, вы должны увидеть нулевое использование расширяемой памяти. На выделенной машине резидентная память должна быть немного меньше, чем общий объем памяти (при условии что ваше рабочее множество настолько же большое, как и память, или даже больше). Резидентная память – это статистика, которая фактически отслеживает, сколько данных находится в физической памяти, но само по себе это не говорит вам о том, как MongoDB использует память.

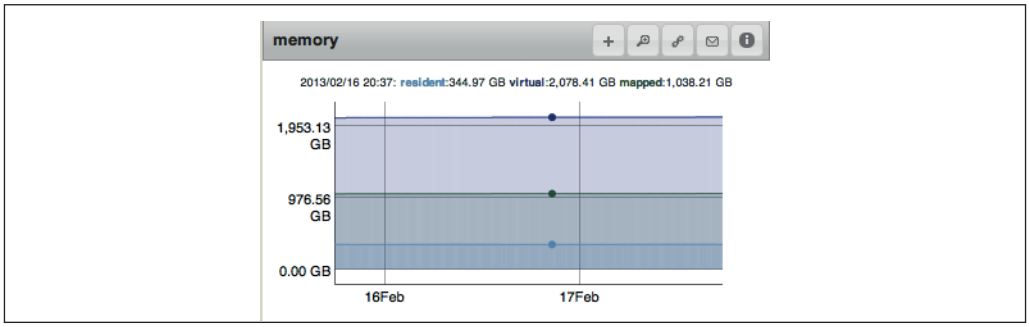

Рис. 22.1. От верхней линии к нижней:
виртуальная, резидентная и расширяемая память

Как видно по рис. 22.1, показатели памяти, как правило, довольно устойчивы, но по мере роста набора данных виртуальная память (верхняя линия) будет расти вместе с ней. Резидентная память (средняя линия) увеличится до размера доступной оперативной памяти и останется неизменной.

Отслеживание отказов страницы

Можно использовать другую статистику, чтобы узнать, как MongoDB использует память, а не только сколько у нее памяти каждого типа. Например, это количество отказов страницы. Так вы узнаете, как часто данные, которые ищет MongoDB, находятся не в оперативной памяти. На рис. 22.2 и 22.3 изображены графики, которые показывают отказы страницы с течением времени. На рис. 22.3 количество отказов меньше, чем на рис. 22.2, но сама по себе эта информация не очень полезна. Если диск, показанный на рис. 22.2, может обработать столько отказов, а приложение может обработать задержку времени подвода головки, то особых проблем с таким количеством (или более) отказов нет. С другой стороны, если ваше приложение не может справиться с повышенной задержкой чтения данных с диска, у вас нет другого выбора, кроме как сохранить все свои данные в памяти (или использовать твердотельные накопители).

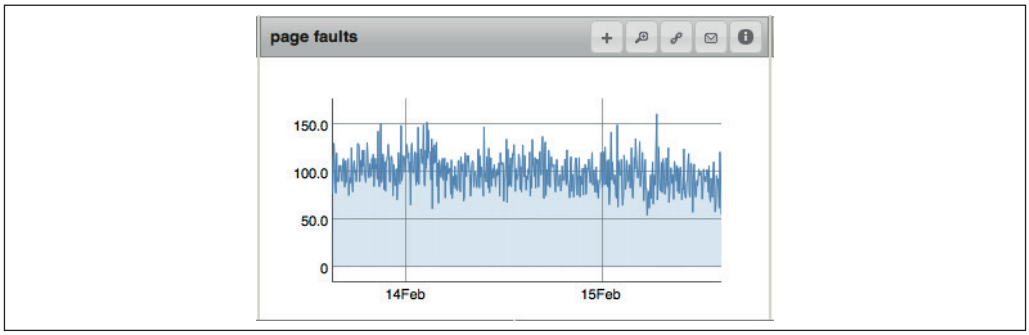

Рис. 22.2. Отказы при обращении к допустимым страницам сотни раз в минуту

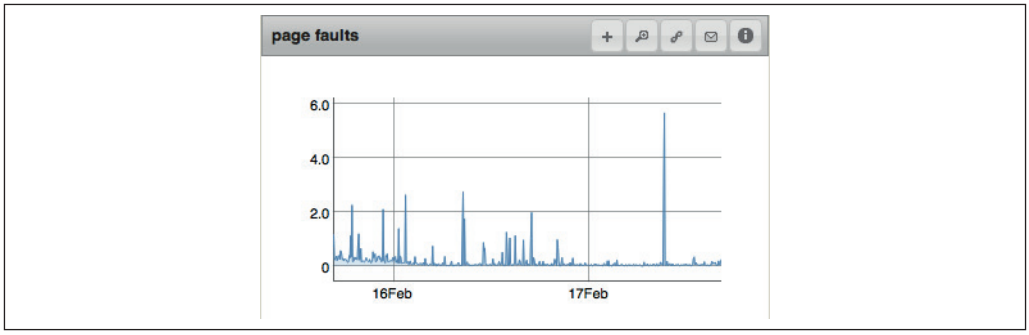

Рис. 22.3. Отказы при обращении к допустимым страницам по нескольку раз в минуту

Независимо от того, насколько снисходительно ваше приложение, отказы страниц становятся проблемой, когда диск перегружен. Объем нагрузки, с которой может справиться диск, не является линейным: как только диск начинает перегружаться, каждой операции придется стоять в очереди все дольше и дольше, создавая цепную реакцию. Обычно наступает переломный момент, когда производительность диска начинает быстро падать. Таким образом, неплохо было бы держаться подальше от максимальной нагрузки, с которой ваш диск может справиться.

Отслеживайте число отказов страниц с течением времени. Если ваше приложение ведет себя хорошо при определенном количестве отказов, это базовый показатель того, сколько отказов может обработать система. Если отказы страниц начинают накапливаться и производительность ухудшается, у вас появляется порог и нужно бить тревогу.

Статистику по отказам страниц для каждой базы данных можно увидеть, посмотрев в поле `"page_faults"` вывода команды `serverStatus`:

```
> db.adminCommand({"serverStatus": 1})["extra_info"]
{ "note" : "fields vary by platform", "page_faults" : 50 }
```

В этом поле указано, сколько раз MongoDB приходилось обращаться к диску (с момента запуска).

Время ожидания ввода/вывода

Отказы страниц в целом тесно связаны с тем, как долго ЦП бездействует в ожидании диска, что называется временем ожидания ввода/вывода. Небольшое время ожидания ввода/вывода нормально; MongoDB иногда обращается к диску, и хотя она пытается ничего не блокировать, полностью избежать этого она не может. Важно то, что время ожидания ввода/вывода не увеличивается или приближается к 100 %, как показано на рис. 22.4. Это указывает на то, что диск перегружен.

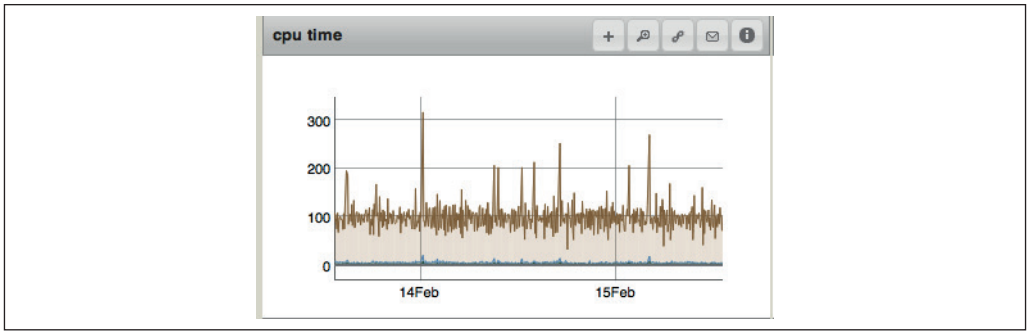

Рис. 22.4. Ожидание ввода/вывода колеблется примерно на отметке 100 %

Вычисление рабочего множества

В целом чем больше данных у вас в памяти, тем быстрее будет работать MongoDB. Таким образом, в порядке убывания от самого быстрого до самого медленного у приложения может быть:

1) весь набор данных в памяти, что приятно, но часто слишком дорого или неосуществимо. Это может быть необходимо для приложений, которые зависят от быстрого времени отклика;
2) рабочее множество в памяти. Это самый распространенный вариант. Ваше рабочее множество – это данные и индексы, которые использует ваше приложение. Это может быть что угодно, но обычно существует основной набор данных (например, коллекция *users* и последний месяц активности), который покрывает 90 % запросов. Если это рабочее множество помещается в ОЗУ, MongoDB, как правило, будет работать быстро: ей нужно обращаться к диску только в случае нескольких «необычных» запросов;

3) индексы в памяти;
4) рабочий набор индексов в памяти;
5) отсутствие полезного подмножества данных в памяти. По возможности избегайте этого. Приложение будет работать медленно.

Вы должны знать, какое у вас рабочее множество (и насколько оно большое), чтобы знать, можете ли вы сохранить его в памяти. Лучший способ рассчитать его размер – отслеживать распространенные операции, чтобы узнать, сколько приложение читает и пишет. Например, предположим, что ваше приложение создает 2 ГБ новых данных в неделю и к 800 МБ этих данных регулярно выполняется обращение. Как правило, пользователи обращаются к данным в возрасте до месяца, а данные, которые старше, в основном не используются. Размер вашего рабочего множества, вероятно, составит около 3,2 ГБ (800 МБ /в неделю × 4 недели), плюс поправка на индексы, поэтому пусть будет 5 ГБ.

Например, можно отслеживать данные, к которым обращались с течением времени, как показано на рис. 22.5. Если вы выберете отсечение, на которое приходится 90 % ваших запросов, как показано на рис. 22.6, тогда данные (и индексы), сгенерированные за этот период времени, образуют ваше рабочее множество. Вы можете выполнить измерения за этот период времени, чтобы выяснить, насколько растет ваш набор данных. Обратите внимание, что в этом примере используется время, но, возможно, есть другой шаблон доступа, который больше подходит для вашего приложения (время – наиболее распространенный вариант).

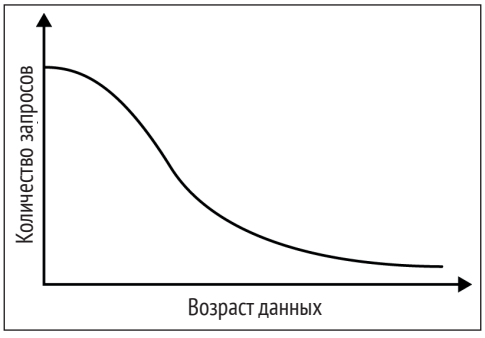

Рис. 22.5. График доступа к данным по возрасту данных

Рис. 22.6. Рабочее множество – это данные, используемые в запросах до отсечки «частых запросов» (обозначено вертикальной линией на графике)

Примеры рабочего множества

Предположим, у вас есть рабочее множество на 40 ГБ. В общей сложности на него приходится 90 % запросов, а 10 % – это другие данные. Если у вас

есть 500 ГБ данных и 50 ГБ ОЗУ, ваше рабочее множество полностью помещается в ОЗУ. После того как ваше приложение получило доступ к данным, к которым оно обычно обращается (этот процесс называется *разогревом*), оно не должно снова обращаться к диску за рабочим множеством. Тогда у него есть 10 ГБ свободного места для 460 ГБ реже используемых данных. Очевидно, что MongoDB почти всегда придется обращаться к диску за данными из нерабочего множества.

С другой стороны, предположим, что ваше рабочее множество не помещается в ОЗУ, например если у вас всего 35 ГБ ОЗУ. Тогда, как правило, оно будет занимать большую часть оперативной памяти. Рабочее множество имеет более высокую вероятность остаться в ОЗУ, поскольку к нему обращаются чаще, но в какой-то момент реже используемые данные придется загрузить в память, избавившись от рабочего множества (или других реже используемых данных). Таким образом, с диска идет постоянный отток туда-сюда: обращение к рабочему набору лишено предсказуемой производительности.

Отслеживание производительности

Производительность запросов часто важно отслеживать и поддерживать согласованной. Есть несколько способов отследить, есть ли у MongoDB проблемы с текущей загрузкой запросов.

ЦП может быть ограничен скоростью ввода/вывода (на это указывает высокое время ожидания ввода/вывода). Подсистема хранения WiredTiger является многопоточной и может использовать дополнительные ядра ЦП. Это видно по более высокому уровню использования метрик ЦП по сравнению с более старой подсистемой хранения MMAP. Однако если пользовательское или системное время приближается к 100 % (или 100 %, умноженное на количество имеющихся у вас процессоров), наиболее распространенная причина – отсутствие индекса в часто используемом запросе. Рекомендуется отслеживать использование процессора (особенно после развертывания новой версии приложения), чтобы убедиться, что все ваши запросы работают должным образом.

Обратите внимание на прекрасный график, показанный на рис. 22.7: если число отказов страниц небольшое, время ожидания ввода/вывода может быть меньше иных показателей ЦП. Только когда другие показатели начнут расти, плохие индексы могут стать источником неприятностей.

Похожая метрика – постановка в очередь: сколько запросов ожидают обработки со стороны MongoDB. Запрос считается поставленным в очередь, когда он ожидает блокировки, которая необходима ему, чтобы выполнить операцию чтения или записи. На рис. 22.8 показан график очередей операций чтения и записи с течением времени. Ни одна из очередей не является предпочтительной (в основном это пустой график), но беспокоиться

не о чем. В загруженной системе операции обычно приходится немного подождать, пока правильная блокировка не будет доступна.

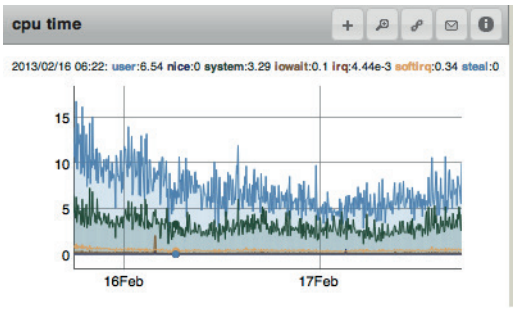

Рис. 22.7. Процессор с минимальным временем ожидания ввода/вывода: верхняя линия – пользовательская, нижняя – системная; остальные характеристики очень близки к 0 %

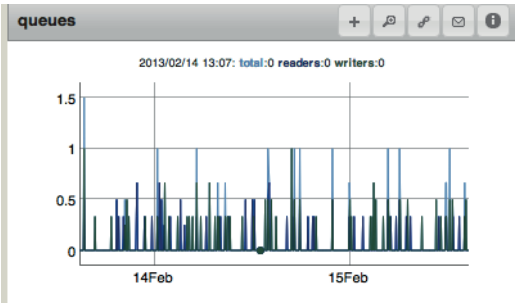

Рис. 22.8. Очереди операций чтения и записи с течением времени

Подсистема хранения WiredTiger обеспечивает параллелизм на уровне документов, что позволяет выполнять несколько одновременных операций записи в одну коллекцию. Это значительно улучшило производительность параллельных операций. Система тикетов контролирует количество используемых потоков, чтобы избежать зависания: она выдает тикеты для операций чтения и записи (по умолчанию 128), после чего новые операции чтения или записи помещаются в очередь. Поля `wiredTiger.concurrentTransactions.read.available` и `wiredTiger.concurrentTransactions.write.available` в команде `serverStatus` могут использоваться, чтобы отслеживать, когда количество доступных тикетов достигает нуля, указывая на то, что соответствующие операции теперь стоят в очереди.

Можно увидеть, накапливаются ли запросы, посмотрев на количество запросов в очереди.

Как правило, размер очереди должен быть низким. Большая и вездесущая очередь указывает на то, что *mongod* не справляется со своей нагрузкой. Нужно уменьшить нагрузку на этот сервер как можно скорее.

Отслеживание свободного пространства

Еще одна метрика, которая является базовой, но важной для мониторинга, – это использование диска. Иногда пользователи ждут, пока на их диске не останется свободного места, прежде чем думать о том, как справиться с этим. Контролируя использование своего диска и отслеживая свободное место на диске, вы можете предсказать, как долго ваш текущий диск будет работать надлежащим образом, и заранее планировать, что делать, если что-то произойдет.

Когда у вас заканчивается свободное место на диске, есть несколько вариантов:

- если вы используете шардинг, добавьте еще один шард;
- если у вас есть неиспользуемые индексы, удалите их. Их можно определить с помощью агрегации `$indexStats` для конкретной коллекции;
- если вы не запустили операцию уплотнения, сделайте это на вторичном узле, чтобы увидеть, поможет ли она. Обычно это полезно только в тех случаях, когда большой объем данных или индексов был удален из коллекции и не будет заменен;
- завершите работу каждого члена набора реплик (по одному) и скопируйте его данные на диск побольше, который затем можно будет смонтировать. Перезапустите члена и перейдите к следующему;
- замените членов вашего набора реплик на членов с диском побольше: удалите старого члена и добавьте нового и дайте ему возможность синхронизироваться с остальной частью набора. Повторите это для каждого члена набора;
- если вы используете опцию `directoryperdb` и у вас особенно быстро растущая база данных, переместите ее на свой собственный диск. Затем смонтируйте том в качестве каталога в своем каталоге данных. Таким образом, остальные ваши данные не нужно будет перемещать.

Независимо от выбранной вами техники, планируйте заранее, чтобы минимизировать влияние на ваше приложение. Вам нужно время для создания резервных копий, изменения каждого члена набора по очереди и копирования своих данных с места на место.

Мониторинг репликации

Отставание репликации и длина журнала операций являются важными показателями для отслеживания. Отставание – когда второстепенные узлы не могут идти в ногу с первичными. Оно рассчитывается путем вычитания

времени последней операции, примененной ко вторичному узлу, из времени последней операции для первичного узла. Например, если вторичный узел только что применил операцию с временной отметкой 3:26:00 вечера, а у первичного узла эта отметка составляет 3:29:45 вечера, вторичный узел отстает на 3 минуты и 45 секунд. Нужно, чтобы отставание было как можно ближе к 0, и обычно оно составляет порядка миллисекунд. Если вторичный узел идет в ногу с первичным, отставание репликации должно выглядеть примерно так, как показано на рис. 22.9: в основном на протяжении всего времени это 0.

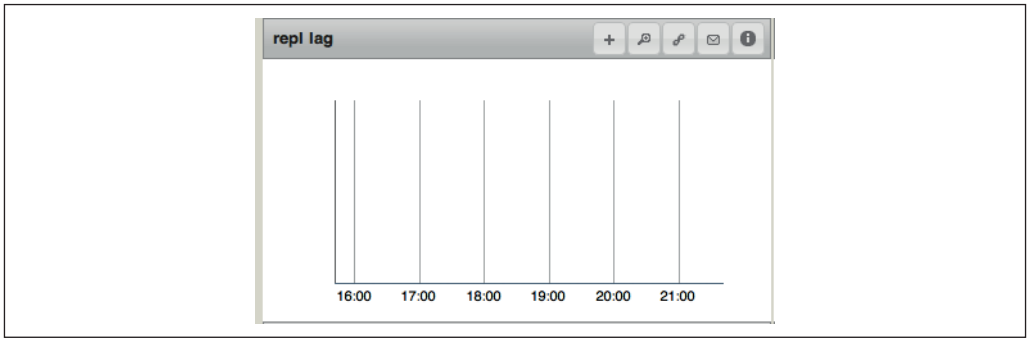

Рис. 22.9. Набор реплик без отставания; это то, что вы хотите увидеть

Если вторичный узел не может реплицировать операции записи так же быстро, насколько первичный может выполнить запись, вы увидите ненулевое отставание. Самый крайний случай – когда репликация застряла: вторичный узел не может применить больше операций по какой-то причине. В этот момент отставание будет увеличиваться по секунде в секунду, создавая крутой склон, показанный на рис. 22.10. Это может быть вызвано проблемами в сети или отсутствующим индексом `"_id"`, который требуется для каждой коллекции, чтобы репликация работала правильно.

Если в коллекции отсутствует индекс `"_id"`, выведите сервер из набора реплик, запустите его как автономный сервер и создайте индекс `"_id"`. Убедитесь, что вы создали индекс `"_id"` в качестве *уникального*.
После этого индекс `"_id"` нельзя удалить или изменить (только если удалить всю коллекцию).

Если система перегружена, вторичный узел может постепенно отставать. Какая-то репликация по-прежнему будет происходить, поэтому, скорее всего, вы не увидите на графике характерный наклон «по секунде в секунду». Тем не менее важно знать, могут ли вторичные узлы идти в ногу с пиковым трафиком или постепенно отстают.

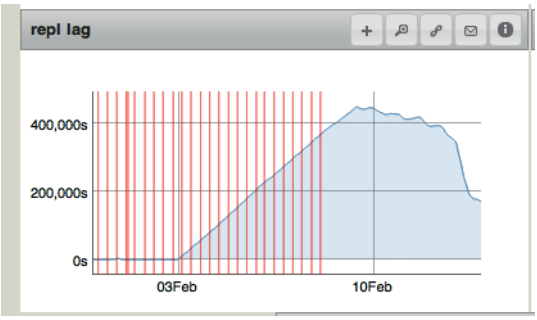

Рис. 22.10. Репликация застревает и незадолго до 10 февраля начинает восстанавливаться; вертикальные линии – перезапуски сервера

Первичные узлы не ограничивают операции записи, чтобы «помочь» вторичным узлам наверстать упущенное, поэтому последние часто отстают на перегруженных системах (особенно когда MongoDB имеет тенденцию устанавливать приоритеты для операций записи над операциями чтения, а это означает, что репликация на первичном узле может зависнуть).

Можно в некоторой степени принудительно регулировать первичный узел, используя параметр `"w"` с гарантией записи. Также можно попробовать удалить нагрузку со вторичного узла, направив все запросы, которые он обрабатывал, другому члену.

Если вы находитесь в крайне *недогруженной* системе, то можете встретить еще один интересный шаблон: внезапные скачки в отставании репликации, как показано на рис. 22.11. Показанные скачки на самом деле не запаздывают – они вызваны изменениями в выборке. *mongod* обрабатывает по одной операции записи каждые пару минут. Поскольку отставание измеряется как разница между временными метками на первичном и вторичном узлах, при измерении временной метки на вторичном узле непосредственно перед записью в первичный узел это выглядит так, как будто налицо отставание на несколько минут. Если увеличить скорость записи, эти скачки должны исчезнуть.

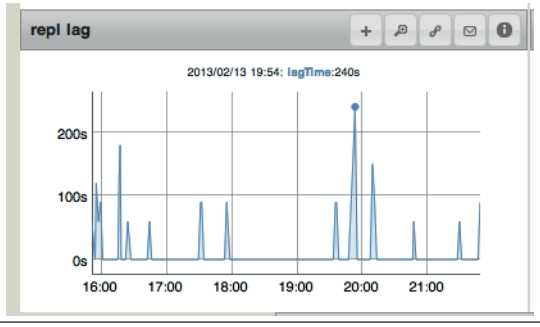

Рис. 22.11. Система с небольшим количеством операций записи может привести к появлению «фантомного» отставания

Другой важной метрикой репликации для отслеживания является длина журнала операций каждого члена. Каждый член, который может стать первичным, должен иметь журнал операций продолжительностью более одного дня. Если один член может быть источником синхронизации для другого члена, у него должен быть журнал операций, превышающий время, необходимое для завершения начальной синхронизации. На рис. 22.12 показано, как выглядит стандартный график длины такого журнала. У него отличная продолжительность: 1111 часов – это больше месяца! В целом журналы операции должны быть настолько длинными, насколько это позволяет место на вашем диске. Учитывая то, как они используются, они практически не занимают память, а длинный журнал операций может означать разницу между болезненным и легким опытом.

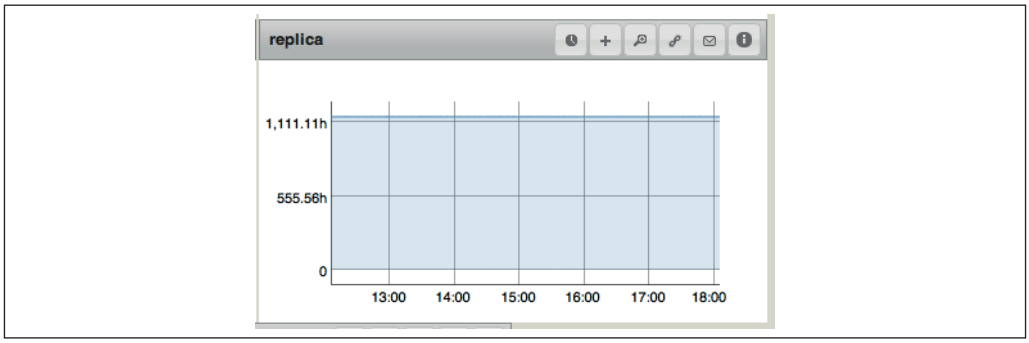

Рис. 22.12. Типичный график длины журнала операций

На рис. 22.13 показано немного необычное изменение, вызванное довольно коротким журналом операций и изменчивым трафиком. Работоспособность по-прежнему налицо, но журнал на этой машине, вероятно, слишком короткий (от 6 до 11 часов обслуживания). У администратора, наверное, появится желание сделать журнал длиннее, когда ему представится такая возможность.

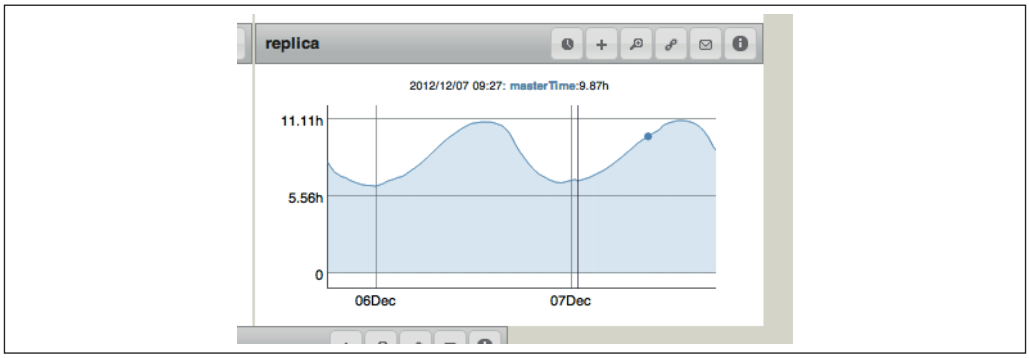

Рис. 22.13. График длины журнала операций приложения с ежедневными пиками трафика

Глава 23

Создание резервных копий

Важно регулярно делать резервные копии своей системы. Резервные копии являются хорошей защитой от большинства различных типов сбоев, и очень небольшое число этих типов нельзя решить путем восстановления из чистой резервной копии. В этой главе рассматриваются распространенные варианты создания резервных копий:

- резервное копирование на одном сервере, включая процедуру восстановления и снимки;
- особые принципы резервного копирования наборов реплик;
- создание резервной копии сегментированного кластера.

Резервные копии полезны только в том случае, если вы уверены в их развертывании в чрезвычайной ситуации. Таким образом, при использовании любого выбранного вами метода резервного копирования обязательно попрактикуйтесь, как создавать резервные копии и как восстанавливать их, пока не освоите процедуру восстановления.

Методы резервного копирования

В MongoDB существует несколько вариантов резервного копирования кластеров. MongoDB Atlas, официальный сервис облачных вычислений MongoDB, предоставляет как непрерывное резервное копирование, так и снимки облачных провайдеров. Непрерывное резервное копирование требует инкрементного резервного копирования данных в кластере, поэтому резервные копии обычно отстают от операционной системы всего на несколько секунд. Снимки облачного провайдера предоставляют локализованное хранилище резервных копий с использованием функционала снимков кластера (например, Amazon Web Services, Microsoft Azure или Google Cloud Platform). Лучшее решение для резервного копирования для большинства сценариев – непрерывное резервное копирование.

MongoDB также обеспечивает возможность резервного копирования с помощью Cloud Manager и Ops Manager. Cloud Manager – это служба резервного копирования, мониторинга и автоматизации для MongoDB.

Ops Manager – это локальное решение, функциональность которого аналогична Cloud Manager.

Для отдельных лиц и групп, управляющих кластерами MongoDB напрямую, существует несколько стратегий резервного копирования. Мы опишем эти стратегии в оставшейся части данной главы.

Резервное копирование сервера

Существует множество способов создания резервных копий. Независимо от метода, создание резервной копии может вызывать нагрузку на систему: обычно для этого требуется чтение всех ваших данных в памяти. Таким образом, резервное копирование, как правило, должно выполняться на вторичных узлах набора реплик (в отличие от первичных) или, для автономных серверов, в нерабочее время.

Приемы, описанные в этом разделе, применимы к любому типу *mongod*, будь то автономный сервер или член набора реплик, если не указано иное.

Снимок файловой системы

Снимки файловой системы используют инструменты системного уровня для создания копий устройства, на котором хранятся файлы данных MongoDB. Это быстрые методы, и они надежно работают, но требуют дополнительной настройки системы за пределами MongoDB.

В MongoDB версии 3.2 добавлена поддержка резервного копирования на уровне томов экземпляров MongoDB с использованием подсистемы хранения WiredTiger, когда файлы данных этих экземпляров и файлы журналов находятся на отдельных томах. Однако для создания согласованной резервной копии база данных должна быть заблокирована, и все операции записи в базу данных должны быть приостановлены во время этого процесса.

До появления версии 3.2 для создания резервных копий на уровне томов экземпляров MongoDB с использованием WiredTiger требовалось, чтобы файлы данных и журнал находились на одном томе.

Снимки работают путем создания указателей между текущими данными и специальным томом снимка. Эти указатели теоретически эквивалентны «жестким ссылкам». Поскольку рабочие данные расходятся со снимком, в этом процессе используется стратегия копирования при записи. В результате снимок хранит только измененные данные.

После создания снимка вы монтируете образ снимка в свою файловую систему и копируете данные из снимка. Полученная резервная копия содержит полную копию всех данных.

База данных должна быть действительной на момент создания снимка. Это означает, что все операции записи, принятые базой данных, должны быть полностью записаны на диск: либо в журнал, либо в файлы данных.

Если во время резервного копирования появятся операции записи, которых нет на диске, резервное копирование не будет отражать эти изменения.

Для подсистемы хранения WiredTiger файлы данных отражают согласованное состояние на момент последней контрольной точки. Контрольные точки происходят каждую минуту.

Снимки создают образ всего диска или тома. Если вам не нужно выполнять резервное копирование всей системы, рассмотрите возможность изоляции файлов данных MongoDB, журнала (если это применимо) и конфигурирования на одном логическом диске, который не содержит никаких других данных.

Кроме того, храните все файлы данных MongoDB на выделенном устройстве, чтобы вы могли делать резервные копии, не дублируя посторонние данные.

Убедитесь, что вы копируете данные из снимков в другие системы. Это гарантирует, что данные защищены от сбоев сайта.

Если в вашем экземпляре *mongod* включено ведение журналов, вы можете использовать любой вид файловой системы или инструмент для создания снимков на уровне томов или блоков для создания резервных копий.

Если вы управляете собственной инфраструктурой в системе на базе Linux, настройте свою систему с помощью Linux Logical Volume Manager (LVM), системы управления томами с данными для Linux, чтобы предоставить пакеты дисков и возможность создания снимков. LVM позволяет гибко комбинировать и разделять разделы физического диска, обеспечивая динамическое изменение размера файловых систем. Вы также можете использовать настройки на основе LVM в облачной/виртуализированной среде.

При начальной настройке LVM сначала мы назначаем разделы диска физическим томам (`pvcreate`), затем один или несколько из них назначаются группе томов (`vgcreate`), а потом мы создаем логические тома (`lvcreate`), ссылаясь на группы томов. Мы можем построить файловую систему на логическом томе (`mkfs`), которую после ее создания можно будет смонтировать для использования (`mount`).

Резервное копирование снимков и процедура восстановления

В этом разделе представлен обзор простого процесса резервного копирования с использованием LVM в системе Linux. Хотя утилиты, команды и пути могут (немного) отличаться от того, что есть в вашей системе, приведенные ниже шаги обеспечивают общий обзор операции резервного копирования.

Используйте только следующую процедуру в качестве руководства для системы резервного копирования и инфраструктуры.

Производственные системы резервного копирования должны учитывать ряд специфичных для приложения требований и факторов, которые являются уникальными для конкретных сред.

Чтобы создать снимок с помощью LVM, выполните команду от имени пользователя root в следующем формате:

```
# lvcreate --size 100M --snapshot --name mdb-snap01 /dev/vg0/mongodb
```

Используя эту команду, мы создаем снимок LVM (с помощью параметра `--snapshot`) с именем `mdbsnap01` тома `mongodb` в группе томов `vg0`, которая будет расположена в каталоге `/dev/vg0/mdb-snap01`. Расположение и пути к вашим системам, группам томов и устройствам могут незначительно отличаться в зависимости от конфигурации LVM вашей операционной системы.

Снимок имеет ограничение в 100 МБ из-за параметра `--size 100M`. Этот размер отражает не общий объем данных на диске, а скорее количество различий между текущим состоянием */dev/vg0/ mongodb* и снимком *(/dev/vg0/mdb-snap01)*.

Снимок будет существовать, когда команда вернется. Вы можете в любое время восстановить данные непосредственно из снимка или создать новый логический том и восстановить его из снимка в альтернативное изображение.

Хотя снимки хорошо подходят для быстрого создания высококачественных резервных копий, они не идеальны в качестве формата для хранения данных резервных копий. Снимки обычно зависят от той же инфраструктуры хранения, что и исходные образы дисков. Поэтому очень важно, чтобы вы заархивировали эти снимки и сохранили их в другом месте.

После создания снимка смонтируйте его и скопируйте данные в отдельное хранилище. Либо возьмите копию образа снимка на уровне блока, например с помощью следующей процедуры:

```
# umount /dev/vg0/mdb-snap01

# dd if=/dev/vg0/mdb-snap01 | gzip > mdb-snap01.gz
```

Эта последовательность команд выполняет следующие действия:

- гарантирует, что устройство */dev/vg0/mdb-snap01* не смонтировано;
- выполняет копию всего образа снимка на уровне блока с помощью команды `dd` и сжимает результат в файл с расширением *.gz* в текущем рабочем каталоге.

Команда `dd` создаст большой файл с расширением *.gz* в вашем текущем рабочем каталоге. Убедитесь, что вы выполняете *эту команду* в файловой системе, в которой достаточно свободного места.

Для восстановления снимка, созданного с помощью LVM, выполните такую последовательность команд:

```
# lvcreate --size 1G --name mdb-new vg0

# gzip -d -c mdb-snap01.gz | dd of=/dev/vg0/mdb-new

# mount /dev/vg0/mdb-new /srv/mongodb
```

Эта последовательность делает следующее:

- создает новый логический том с именем *mdb-new* в группе томов */dev/vg0*. */dev/vg0/mdb-new* – путь к новому устройству. Вы можете использовать другое имя и изменить 1G на нужный размер тома;
- распаковывает и разархивирует файл *mdb-snap01.gz* в образ диска *mdb-new*;
- монтирует образ диска *mdb-new* в каталог */srv/mongodb*. Измените точку монтирования в соответствии с местоположением вашего файла данных MongoDB или другим необходимым местоположением.

У восстановленного снимка будет устаревший файл *mongod.lock*. Если вы не удалите этот файл из снимка, MongoDB может предположить, что он указывает на нечистое завершение работы. Если параметр `storage.journal.enabled` активирован и вы не используете функцию `db.fsyncLock()`, вам не нужно удалять файл *mongod.lock*. Если вы используете эту функцию, вам нужно будет снять блокировку.

Чтобы восстановить резервную копию без записи в сжатый файл с расширением *.gz*, используйте следующую последовательность команд:

```
# umount /dev/vg0/mdb-snap01

# lvcreate --size 1G --name mdb-new vg0

# dd if=/dev/vg0/mdb-snap01 of=/dev/vg0/mdb-new

# mount /dev/vg0/mdb-new /srv/mongodb
```

Вы можете создавать резервные копии вне системы, используя комбинированный процесс и SSH. Эта последовательность идентична процедурам, описанным ранее, за исключением того, что она архивирует и сжимает резервную копию в удаленной системе с использованием SSH:

```
umount /dev/vg0/mdb-snap01

dd if=/dev/vg0/mdb-snap01 | ssh username@example.com gzip > /opt/backup/
mdb-snap01.gz

lvcreate --size 1G --name mdb-new vg0

ssh username@example.com gzip -d -c /opt/backup/mdb-snap01.gz | dd
of=/dev/vg0/mdb-new

mount /dev/vg0/mdb-new /srv/mongodb
```

Начиная с MongoDB версии 3.2 для резервного копирования на уровне томов экземпляров MongoDB с использованием WiredTiger файлы данных и журнал больше не должны размещаться на одном томе. Однако база данных должна быть заблокирована, и все операции записи в базу данных должны быть приостановлены во время процесса резервного копирования, чтобы обеспечить согласованность копирования.

Если ваш экземпляр *mongod* работает без ведения журнала или файлы журнала находятся на отдельном томе, вы должны сбросить все записи на диск и заблокировать базу данных, чтобы предотвратить запись во время процесса резервного копирования. Если у вас есть конфигурация набора реплик, используйте для резервного копирования вторичный узел, который не получает операции чтения (то есть скрытого члена).

Для этого выполните метод `db.fsyncLock()` в оболочке *mongo*:

```
> db.fsyncLock();
```

Затем выполните операцию резервного копирования, описанную ранее.

После разблокируйте базу данных, введя следующую команду в оболочке *mongo*:

```
> db.fsyncUnlock();
```

Этот процесс более подробно описан в следующем разделе.

Копирование файлов данных

Еще один способ создания резервных копий на одном сервере – сделать копию всего, что есть в каталоге данных. Поскольку вы не можете копировать все файлы одновременно без поддержки файловой системы, вы

должны предотвратить изменение файлов данных во время выполнения копии. Это можно сделать с помощью команды `fsyncLock`:

```
> db.fsyncLock()
```

Эта команда блокирует базу данных от любых дальнейших операций записи, а затем сбрасывает все грязные данные на диск (`fsync`), гарантируя, что файлы в каталоге данных имеют самую последнюю согласованную информацию и не изменяются.

Как только эта команда будет выполнена, *mongod* поставит в очередь все входящие операции записи. Он не будет обрабатывать дальнейшие операции записи, пока не последует разблокировка. Обратите внимание, что эта команда останавливает операции записи во *все* базы данных (не только в ту, к которой подключена *db*).

Как только команда `fsyncLock` вернется, скопируйте все файлы в вашем каталоге данных в папку в место резервного копирования. В Linux это можно сделать с помощью такой команды:

```
$ cp -R /data/db/* /mnt/external-drive/backup
```

Убедитесь, что вы копируете абсолютно все файлы и папки из каталога данных в место резервного копирования. Исключение файлов или каталогов может сделать резервную копию непригодной или поврежденной.

Как только вы закончите копировать данные, разблокируйте базу данных, чтобы она снова могла принимать операции записи:

```
> db.fsyncUnlock()
```

Ваша база данных опять начнет обрабатывать записи в обычном режиме.

Обратите внимание, что существуют некоторые проблемы блокировки с аутентификацией и командой `fsyncLock`. Если вы используете аутентификацию, не закрывайте оболочку между вызовами `fsyncLock` и `fsyncUnlock`. Если вы отключитесь, возможно, вы не сможете восстановить соединение, и вам придется перезапустить `mongod`. Параметр `fsyncLock` не сохраняется в промежутке между перезапусками; `mongod` всегда будет запускаться разблокированным.

В качестве альтернативы `fsyncLock` можно отключить *mongod*, скопировать файлы и затем снова запустить его. При выключении *mongod* все изменения фактически сбрасываются на диск, и во время резервного копирования новые операции записи не появляются.

Для восстановления файлов из копии каталога данных убедитесь, что *mongod* не запущен и что каталог данных, в который вы хотите восстановить файлы, пуст. Скопируйте резервные копии файлов данных в каталог данных и запустите *mongod*. Например, следующая команда восстановит

файлы, зарезервированные с помощью команды, показанной ранее:

```
$ cp -R /mnt/external-drive/backup/* /data/db/
$ mongod -f mongod.conf
```

Несмотря на предупреждения о частичных копиях каталога данных, вы можете использовать этот метод для резервного копирования отдельных баз данных, если знаете, что копировать и где это находится, с помощью параметра `--directoryperdb`. Чтобы создать резервную копию отдельной базы данных (например, *myDB*), которая доступна только в том случае, если вы используете параметр `--directoryperdb`, скопируйте весь каталог *myDB*. Частичные копии каталога данных возможны только при использовании параметра `-directoryperdb`.

Вы можете восстановить определенные базы данных, скопировав в каталог данных лишь файлы с правильным именем базы данных. Вы должны начать с чистого отключения, чтобы выполнить восстановление поэтапно, как есть. Если у вас произошел сбой или полное отключение, не пытайтесь восстановить отдельную базу данных из резервной копии: замените весь каталог и запустите *mongod*, чтобы разрешить воспроизведение файлов журнала.

Никогда не используйте команду `fsyncLock` в связке с *mongodump* (описано далее). В зависимости от того, что еще делает ваша база данных, *mongodump* может зависнуть навсегда, если база данных заблокирована.

Использование mongodump

Последний способ создания резервной копии на одном сервере – использование *mongodump*. Эта утилита упоминается последней, потому что у нее есть некоторые недостатки. Она медленнее работает (как при получении резервной копии, так и при восстановлении из нее) и имеет ряд проблем, связанных с наборами реплик, которые обсуждаются в разделе «Особые концепции для наборов реплик». Однако у нее также есть и некоторые преимущества: это хороший способ для резервного копирования отдельных баз данных, коллекций и даже подмножеств коллекций.

mongodump имеет множество параметров, которые можно увидеть, выполнив команду `mongodump --help`. Здесь мы сосредоточимся на наиболее полезных, чтобы использовать их для резервного копирования.

Для резервного копирования всех баз данных просто запустите *mongodump*. Если вы используете *mongodump* на той же машине, что и *mongod*, можете просто указать порт, на котором работает *mongod*:

```
$ mongodump -p 31000
```

mongodump создаст в текущем каталоге каталог *dump*, в котором содержится дамп всех ваших данных. Этот каталог организован в папки и подпапки по базам данных и коллекциям. Фактические данные хранятся в файлах с расширением *.bson*, которые просто содержат каждый документ в коллекции в формате BSON и объединены вместе. Вы можете просматривать файлы *.bson* с помощью утилиты *bsondump*, которая поставляется с MongoDB.

Вам даже не нужен работающий сервер, чтобы использовать *mongodump*. Вы можете использовать параметр `--dbpath`, чтобы указать каталог данных, а *mongodump* будет использовать файлы данных, чтобы скопировать данные:

```
$ mongodump --dbpath /data/db
```

Не нужно использовать этот параметр, если *mongod* работает.

Одна из проблем, связанных с *mongodump*, заключается в том, что это не мгновенное резервное копирование: система может принимать операции записи во время резервного копирования. Таким образом, вы можете столкнуться с ситуацией, когда пользователь A начинает резервное копирование, в результате чего *mongodump* выгружает базу данных A, но пока это происходит, пользователь B удаляет базу A. Однако *mongodump* уже выгрузила ее, поэтому вы получите снимок данных, который не соответствует состоянию на исходном сервере.

Чтобы избежать этого, если вы используете *mongod* с параметром `--replSet`, вы можете использовать параметр *mongodump* `--oplog`. Так вы будете отслеживать все операции, которые происходят на сервере во время выполнения выгрузки, поэтому эти операции можно воспроизвести при восстановлении резервной копии. Это даст вам согласованный снимок данных на определенный момент времени с исходного сервера.

Если вы передадите *mongodump* строку соединения с набором реплик (например, `"setName/seed1, seed2, seed3"`), она автоматически выберет первичный узел, из которого будет выполняться выгрузка. Если вы хотите использовать вторичный узел, можно указать предпочтение чтения. Предпочтение чтения может быть задано `--uri connection string`, `uri readPreferenceTags` или с помощью параметра командной строки `--readPreference`. Для получения более подробной информации о различных настройках и опциях, пожалуйста, *см.* документацию по *mongodump* на странице https://oreil.ly/GH3-O.

Для восстановления из резервной копии *mongodump* используйте утилиту *mongorestore*:

```
$ mongorestore -p 31000 --oplogReplay dump/
```

Если вы применяли опцию `--oplog` для выгрузки базы данных, то должны использовать параметр `--oplogReplay` с *mongorestore*, чтобы получить снимок на определенный момент времени.

Если вы заменяете данные на работающем сервере, у вас, вероятно, появится желание (или не появится) использовать параметр `--drop`, который удаляет коллекцию перед ее восстановлением.

Поведение утилит *mongodump* и *mongorestore* с течением времени изменилось. Чтобы предотвратить проблемы совместимости, попробуйте использовать одну и ту же версию обеих утилит (их версии можно увидеть с помощью команд `mongodump --version` и `mongorestore --version`).

Начиная с версии MongoDB 4.2 и выше нельзя использовать *mongodump* или *mongorestore* в качестве стратегии для резервного копирования разделенного кластера. Эти утилиты не поддерживают гарантии атомарности транзакции в шардах.

Перемещение коллекций и баз данных с помощью mongodump и mongorestore

Вы можете восстановить файлы в совершенно другую базу данных и коллекцию, а не в ту, которую выгрузили. Это может быть полезно, если в разных средах используются разные имена баз данных (скажем, *dev* и *prod*), но одни и те же имена коллекций.

Чтобы восстановить файл с расширением *.bson* в определенную базу данных и коллекцию, укажите цели в командной строке:

```
$ mongorestore --db newDb --collection someOtherColl dump/oldDB/oldColl.bson
```

Также возможно использовать эти утилиты с SSH для выполнения миграции данных без дискового ввода/вывода, используя функцию архива. Это объединяет три этапа в одну операцию. Ранее вам приходилось выполнять резервное копирование на диск, а затем копировать эти файлы на целевой сервер и запускать *mongorestore* на этом сервере, чтобы восстановить резервные копии:

```
$ ssh eoin@proxy.server.com mongodump --host source.server.com\ --archive
| ssh eoin@target.server.com mongorestore -archive
```

Сжатие можно сочетать с функцией архивирования этих инструментов для дальнейшего уменьшения размера информации, отправляемой при выполнении миграции данных. Вот тот же пример миграции данных по протоколу SSH с использованием функций архивирования и сжатия:

```
$ ssh eoin@proxy.server.com mongodump --host source.server.com\ --archive
--gzip | ssh eoin@target.server.com mongorestore --archive -gzip
```

Административные сложности с уникальными индексами

Если у вас есть уникальный индекс (не `"_id"`) для любой из ваших коллекций, вам следует рассмотреть возможность использования другого типа резервного копирования, отличного от утилит *mongodump* и *mongorestore*. Уникальные индексы требуют, чтобы данные не изменялись таким образом, чтобы это нарушало ограничение уникального индекса во время копирования. Самый безопасный способ гарантировать это – выбрать метод, который «замораживает» данные, а затем сделать резервную копию, как описано в любом из двух предыдущих разделов.

Если вы решили использовать утилиты *mongodump* и *mongorestore*, вам может потребоваться предварительно обработать свои данные при восстановлении из резервной копии.

Особые факторы при копировании наборов реплик

Основной дополнительный фактор при резервном копировании набора реплик состоит в том, что помимо данных необходимо также фиксировать состояние набора реплик, чтобы обеспечить точный снимок вашего развертывания.

Как правило, нужно делать резервные копии из вторичного узла: это позволяет не загружать первичный узел, и вы можете заблокировать вторичный узел, не влияя на свое приложение (если ваше приложение не отправляет запросы на чтение). Вы можете использовать любой из трех описанных выше методов для резервного копирования члена набора реплик, но рекомендуется сделать снимок файловой системы или копию файла данных. Любой из этих методов может быть применен к вторичным узлам набора реплик без изменений.

mongodump не так проста в использовании, когда репликация активирована. Во-первых, если вы используете эту утилиту, то должны сделать резервные копии, используя параметр `--oplog`, чтобы получить снимок; в противном случае состояние резервной копии не будет соответствовать состоянию других членов кластера. Также нужно создать журнал операций при восстановлении из резервной копии *mongodump*, или восстановленный член не будет знать, где он был синхронизирован.

Чтобы восстановить член набора реплик из резервной копии *mongodump*, запустите члена целевого набора реплик в качестве автономного сервера с пустым каталогом данных и запустите на нем утилиту *mongorestore* (как описано в предыдущем разделе) с параметром `--oplogReplay`. Теперь у него

должна быть полная копия данных, но ему по-прежнему нужен журнал операций. Создайте его, используя команду `createCollection`:

```
> use local
> db.createCollection("oplog.rs", {"capped" : true, "size" : 10000000})
```

Укажите размер коллекции в байтах. *См.* раздел «Изменение размера журнала операций» для получения рекомендаций по изменению его размера.

Теперь вам нужно заполнить журнал. Самый простой способ сделать это – восстановить файл резервной копии *oplog.bson* из выгрузки в коллекцию *local.oplog.rs*:

```
$ mongorestore -d local -c oplog.rs dump/oplog.bson
```

Обратите внимание, что это не выгрузка самого журнала (*dump/local/oplog.rs.bson*), а скорее операций журнала, которые произошли во время выгрузки. После завершения работы *mongorestore* вы можете перезапустить этот сервер в качестве члена набора реплик.

Особые факторы при копировании разделенного кластера

Основной дополнительный фактор при резервном копировании разделенного кластера с использованием подходов, описанных в этой главе, заключается в том, что резервное копирование фрагментов можно выполнять только тогда, когда они активны, а в случае с разделенными кластерами невозможно «идеально» выполнить резервное копирование, пока они активны: нельзя получить снимок всего состояния кластера в определенный момент времени. Тем не менее это ограничение, как правило, обходит тот факт, что по мере увеличения кластера становится все менее и менее вероятным, что вам когда-нибудь придется восстанавливать все это из резервной копии. Таким образом, при работе с разделенным кластером мы сосредоточиваемся на резервном копировании фрагментов: конфигурационных серверов и наборов реплик по отдельности. Если вам нужна возможность резервного копирования всего кластера на определенный момент времени или вы предпочитаете автоматизированное решение, можете воспользоваться облачным менеджером MongoDB или функцией резервного копирования Atlas.

Отключите балансировщик перед выполнением любой из этих операций в разделенном кластере (резервное копирование или восстановление). Нельзя получить согласованный снимок с летающими вокруг чанками. *См.* раздел «Балансировка данных» для получения инструкций по включению и выключению балансировщика.

Резервное копирование и восстановление всего кластера

Когда кластер очень маленький или находится в разработке, вы, возможно, захотите выгрузить и восстановить его целиком. Это можно сделать, выключив балансировщик и затем запустив утилиту *mongodump* через *mongos*, в результате чего будет создана резервная копия всех шардов на любом компьютере, где работает *mongodump*.

Чтобы восстановить файлы из этого типа резервной копии, запустите *mongorestore*, подключенную к *mongos*.

Кроме того, после выключения балансировщика вы можете создавать резервные копии файловых систем или каталогов данных каждого шарда и серверов конфигурации. Однако вы неизбежно будете получать копии каждого из них в несколько разное время, что может быть или не быть проблемой. Кроме того, как только вы включите балансировщик и произойдет миграция, часть данных, которые вы сохранили с одного из шардов, исчезнет оттуда.

Резервное копирование и восстановление одного шарда

Чаще всего вам нужно будет восстановить только один шард в кластере. Если вы не слишком разборчивы, можно сделать восстановление из резервной копии этого шарда, используя один из только что описанных методов с применением одного сервера.

Однако есть одна важная проблема, о которой следует знать. Предположим, вы сделали резервную копию своего кластера в понедельник. В четверг ваш диск тает, и вы должны восстановить данные из резервной копии. В последующие дни новые чанки могли переместиться в этот шард.

В вашей резервной копии шарда с понедельника этих новых чанков не будет. Возможно, вы сможете использовать резервную копию сервера конфигурации, чтобы выяснить, где в понедельник находились исчезающие чанки, но это гораздо сложнее, чем просто восстановить шард. В большинстве случаев восстановление шарда и потеря данных в этих чанках является предпочтительным маршрутом.

Вы можете подключиться к шарду напрямую для восстановления из резервной копии (вместо того чтобы использовать *mongos*).

Глава 24

Развертывание MongoDB

В этой главе приведены рекомендации по настройке сервера для запуска в производство. В частности, здесь рассказывается о:

- том, какое оборудование приобретать и как его настроить;
- применении виртуального окружения;
- важных настройках ядра и дискового ввода/вывода;
- настройках сети: кому куда нужно подключаться.

Проектирование системы

Как правило, вы хотите оптимизировать безопасность данных и максимально быстрый доступ, который вы можете себе позволить. В этом разделе обсуждается наиболее подходящий способ достичь этих целей при выборе дисков, конфигурации RAID, процессоров и других аппаратных и низкоуровневых программных компонентов.

Выбор носителя для хранения

В порядке предпочтения мы хотели бы хранить и извлекать данные из:

1) ОЗУ;
2) твердотельного накопителя;
3) вращающегося диска.

К сожалению, большинство людей имеют ограниченный бюджет или достаточно данных, поэтому хранить все в оперативной памяти нецелесообразно, а твердотельные накопители слишком дороги. Таким образом, типичное развертывание – это небольшой объем оперативной памяти (относительно общего объема данных) и много места на вращающемся диске. Если вы относите себя к этому лагерю, важно, чтобы ваш рабочий набор был меньше оперативной памяти, и вы должны быть готовы к масштабированию, если рабочий набор станет больше.

Если вы можете потратить сколько вам хочется на аппаратное обеспечение, приобретите много оперативной памяти и/или твердотельные накопители.

Чтение данных из ОЗУ занимает несколько наносекунд (скажем, 100). И наоборот, чтение с диска занимает несколько миллисекунд (скажем, 10). Возможно, разницу между этими двумя числами трудно себе представить, поэтому давайте увеличим их до более подходящих цифр: если доступ к ОЗУ занял 1 секунду, обращение к диску заняло бы больше одного дня!

100 наносекунд × 10 000 000 = 1 секунда;

10 миллисекунд × 10 000 000 = 1,16 дня.

Это очень сложные вычисления (ваш диск может быть немного быстрее или ОЗУ немного медленнее), но величина этой разницы не сильно меняется. Таким образом, мы хотим обращаться к диску как можно реже.

Рекомендуемые уровни спецификации RAID

RAID – это аппаратное или программное обеспечение, которое позволяет обрабатывать несколько дисков, как если бы они были одним диском. Эту технологию можно использовать для обеспечения надежности и производительности или того и другого. Набор дисков, использующих RAID, называется массивом RAID (звучит несколько чересчур, поскольку RAID – это аббревиатура, образованная от англ. Redundant Array of Independent Disks – избыточный массив независимых дисков).

Существует несколько способов настройки RAID в зависимости от функций, которые вы ищете, – обычно это сочетание скорости и отказоустойчивости. Вот самые распространенные уровни спецификации:

RAID0

Чередование дисков для повышения производительности. Каждый диск содержит часть данных, аналогично шардингу в MongoDB. Поскольку базовых дисков несколько, на диск одновременно может быть записано большое количество данных. Это улучшает пропускную способность при записи. Однако в случае сбоя диска и потери данных его копии не создаются. Это также может вызвать замедление операций чтения, поскольку некоторые тома данных могут работать медленнее, по сравнению с другими.

RAID1

Зеркалирование для повышения надежности. Идентичная копия данных записывается в каждый элемент массива. У этого уровня более низкая производительность по сравнению с RAID0, поскольку один элемент с медленным диском может замедлить все операции записи. Тем не менее, если диск выйдет из строя, у вас все равно будет копия данных на другом элементе массива.

RAID5

Чередование дисков, а также хранение дополнительной части данных о других данных, которые были сохранены, чтобы предотвратить потерю данных при сбое сервера. По сути, *RAID5* может справиться с одним диском, который выходит из строя, и скрыть этот сбой от пользователя. Тем не менее этот уровень медленнее, чем любой из других перечисленных здесь, потому что он должен вычислять эту дополнительную часть информации всякий раз, когда данные пишутся. В случае с MongoDB это особенно дорого, поскольку типичная рабочая нагрузка делает много небольших операций записи.

RAID10

Сочетание уровней RAID0 и RAID1: данные чередуются для скорости и зеркалируются для надежности.

Мы рекомендуем использовать уровень RAID10: он безопаснее RAID0 и может сгладить проблемы с производительностью, которые могут возникнуть при использовании уровня RAID1. Тем не менее некоторые полагают, что RAID1 поверх наборов реплик избыточен, и выбирают RAID0. Это вопрос личных предпочтений: насколько вы готовы рисковать ради производительности?

Не используйте RAID5: он очень и очень медленный.

Центральный процессор

Традиционно у MongoDB не было проблем при использовании ЦП, но с применением подсистемы хранения WiredTiger это уже не так. Подсистема хранения WiredTiger является многопоточной и может использовать дополнительные ядра процессора. Поэтому вы должны найти баланс для своих инвестиций между памятью и процессором.

Выбирая между скоростью и количеством ядер, выбирайте скорость. MongoDB лучше использует больше циклов на одном процессоре, чем увеличенное распараллеливание.

Операционная система

64-битная Linux – операционная система, на которой работает MongoDB. Если возможно, воспользуйтесь ею. CentOS и Red Hat Enterprise Linux, вероятно, являются наиболее популярными вариантами, но любой вариант должен работать (Ubuntu и Amazon Linux также распространены). Обязательно используйте самую последнюю стабильную версию операционной системы, потому что старые пакеты с ошибками или ядра иногда могут вызывать проблемы.

64-битная версия Windows также хорошо поддерживается.

Другие разновидности Unix не так хорошо поддерживаются: будьте осторожны, если вы используете Solaris или один из вариантов BSD. Сборки для этих систем, по крайней мере исторически, имели много проблем. MongoDB явно прекратил поддержку Solaris в августе 2017 года, отметив отсутствие признания этой системы у пользователей.

Одно важное замечание касательно кросс-совместимости: MongoDB использует один и тот же проводной протокол и одинаково размещает файлы данных во всех системах, поэтому вы можете развертывать их в комбинации операционных систем. Например, у вас может быть процесс *mongos*, работающий в Windows, и *mongod*'ы, которые являются его шардами, работающими в Linux. Вы также можете копировать файлы данных из Windows в Linux или наоборот без проблем совместимости.

Начиная с версии 3.4 MongoDB больше не поддерживает 32-битные платформы с архитектурой x86. Не запускайте какой бы то ни было сервер MongoDB на 32-битном компьютере.

MongoDB работает с архитектурами с прямым порядком байтов и одной архитектурой с обратным порядком байтов: zSeries от IBM. Большинство драйверов поддерживают системы с прямым и обратным порядками байтов, поэтому можно запускать клиентов на любом из них. Однако сервер обычно будет работать на машине с прямым порядком байтов.

Объем подкачки

Нужно выделить небольшой объем подкачки на тот случай, если будут достигнуты ограничения на объем памяти, чтобы ядро не остановило работу MongoDB. Обычно она не использует пространство подкачки, но в экстремальных случаях его может использовать подсистема хранения WiredTiger. Если это произойдет, вам следует подумать об увеличении объема памяти вашего компьютера или пересмотре рабочей нагрузки, чтобы избежать этой проблемной ситуации ради сохранения производительности и стабильности.

Большая часть памяти, используемая MongoDB, «скользкая»: она будет сброшена на диск и заменена другой памятью, как только система запросит место для чего-то еще. Поэтому никогда не нужно писать данные базы данных в пространство подкачки: сначала они будут сброшены на диск.

Однако иногда MongoDB будет использовать подкачку для операций, требующих упорядочения данных: либо для построения индексов, либо для сортировки. Она пытается не использовать слишком много памяти для этих типов операций, но, выполняя многие из них одновременно, вы можете осуществить подкачку принудительно.

Если вашему приложению удается заставить MongoDB использовать пространство подкачки, вам следует заняться перепроектированием приложения или снижением нагрузки на сервер подкачки.

Файловая система

В Linux рекомендуется использовать только файловую систему XFS для ваших томов данных с подсистемой хранения WiredTiger. Можно использовать с WiredTiger файловую систему ext4, но имейте в виду, что тут существуют известные проблемы с производительностью (в частности, она может зависнуть на контрольных точках WiredTiger).

Для Windows хорошо подходит либо NTFS, либо FAT.

Не используйте сетевое хранилище файлов (NFS), смонтированное напрямую для хранилища MongoDB. Некоторые клиентские версии врут по поводу очистки, случайным образом повторно монтируют и очищают страничный кеш и не поддерживают эксклюзивную блокировку файлов. Использование NFS может привести к повреждению журнала, и его следует избегать любой ценой.

Виртуализация

Виртуализация – отличный способ получить дешевое оборудование и быстро расширяться. Тем не менее тут есть и некоторые недостатки – особенно непредсказуемая сеть и дисковый ввод/вывод. В этом разделе рассматриваются проблемы, связанные с виртуализацией.

Избыточное выделение памяти

Параметр ядра Linux `memory overcommit` контролирует, что происходит, когда процессы запрашивают слишком много памяти у операционной системы. В зависимости от того, как оно установлено, ядро может отдавать память процессам, даже если эта память фактически не доступна (в надежде, что она станет доступной к тому времени, когда она понадобится процессу). Это называется *избыточным выделением*: ядро обещает память, которой на самом деле нет. Данная настройка ядра операционной системы не очень хорошо работает с MongoDB.

Возможные значения для `vm.overcommit_memory`: 0 (ядро догадывается, сколько нужно выделить); 1 (выделение памяти всегда происходит успешно) или 2 (не выделяем больше виртуального адресного пространства, чем пространство подкачки плюс часть коэффициента избыточного выделения).

Значение 2 сложное, но это лучший доступный вариант. Чтобы установить его, выполните команду

```
$ echo 2 > /proc/sys/vm/overcommit_memory
```

После изменения этого параметра операционной системы перезагружать MongoDB не нужно.

Таинственная память

Иногда уровень виртуализации неправильно обрабатывает выделение памяти.

Таким образом, у вас может быть виртуальная машина, которая утверждает, что у нее доступно 100 ГБ оперативной памяти, но позволяет вам получить доступ только к 60 ГБ. И наоборот, мы видели, как люди, у которых должно было быть 20 ГБ памяти, в конечном итоге смогли разместить в ОЗУ целый набор данных размером 100 ГБ!

Если предположить, что вам не повезло, вы ничего не сможете сделать. Если ваша операционная система настроена на чтение должным образом и ваша виртуальная машина просто не использует всю необходимую память, вам, возможно, придется просто переключаться между виртуальными машинами.

Обработка проблем ввода/вывода сетевого диска

Одна из самых больших проблем, связанных с использованием виртуализированного оборудования, состоит в том, что обычно вы используете диск совместно с другими арендаторами, что усугубляет упомянутую ранее медлительность диска, потому что все борются за дисковый ввод/вывод. Таким образом, виртуализированные диски могут обладать очень непредсказуемой производительностью: они могут нормально работать, когда ваши соседи не заняты, и внезапно их работа замедляется, если кто-то еще начинает «бомбить» диск.

Другая проблема заключается в том, что это хранилище часто физически не подключено к машине, на которой работает MongoDB, поэтому даже если у вас есть весь диск, ввод/вывод будет медленнее, чем с локальным диском. Существует также маловероятный, но возможный сценарий потери сервером MongoDB сетевого подключения к вашим данным.

У Amazon есть, пожалуй, наиболее широко используемое сетевое хранилище блоков, которое называется Elastic Block Store (EBS). Тома EBS могут быть подключены к инстансам Elastic Compute Cloud (EC2), что позволяет вам сразу же предоставить машине практически любой объем диска. Если вы используете EC2, вам также следует задействовать расширенную сеть AWS, если она доступна для типа инстанса, а также отключить динамическое масштабирование напряжения и частоты (DVFS) и режимы энергосбережения ЦП плюс гиперпоточность. С другой стороны, EBS делает резервное копирование очень простым (сделайте снимок со вторичного узла, смонтируйте накопитель EBS на другом инстансе и запустите *mongod*). С другой стороны, вы можете столкнуться с переменной производительностью.

Если вам требуется более предсказуемая производительность, есть несколько вариантов. Одним из них является размещение MongoDB на ваших собственных серверах – таким образом, вы знаете, что никто ничего не тормозит. Однако для многих это не вариант, поэтому лучше всего получить инстанс в облаке, который гарантирует определенное количество операций ввода/вывода в секунду (IOPS). *См.* http://docs.mongodb.org, где приводятся актуальные рекомендации.

Если вы не можете использовать ни один из этих вариантов и вам требуется больше дискового ввода/вывода, чем может выдержать перегруженный том EBS, есть способ обойти это. По сути, вы можете следить за объемом, который использует MongoDB. Если и когда этот том замедляется, немедленно остановите этот инстанс и используйте новый с другим объемом данных.

Есть статистические данные, за которыми нужно следить:

- скачки при использовании ввода/вывода (в Cloud Manager и Atlas используется термин «IO wait») по понятным причинам;
- скачки показателей отказов страниц. Обратите внимание, что изменения в поведении приложения могут также вызвать изменения рабочего множества: вы должны отключить этот сценарий перед развертыванием новых версий своего приложения;
- количество потерянных TCP-пакетов растет (Amazon особенно резок в этом отношении: когда производительность начинает падать, он сбрасывает TCP-пакеты повсюду);
- скачки очередей операций чтения и записи MongoDB (это можно увидеть в Cloud Manager и Atlas или в столбце `qr/qw` утилиты *mongostat*).

Если ваша нагрузка меняется в течение дня или недели, убедитесь, что ваш сценарий учитывает это: вы не хотите, чтобы какое-то непонятное задание cron остановило все ваши инстансы из-за необычайно тяжелой утренней суеты в понедельник.

Этот курьез полагается на то, что вы недавно сделали резервные копии или сравнительно быстро синхронизируете наборы данных. Если каждый инстанс у вас содержит терабайты данных, можно использовать альтернативный подход. Кроме того, *вероятно*, только это и сработает: если ваш новый том тоже будут «бомбить», он будет таким же медленным, как и старый.

Использование несетевых дисков

В этом разделе используется терминология Amazon. Тем не менее она может применяться и к другим поставщикам.

Эфемерные диски – это фактические диски, подключенные к физической машине, на которой работает ваша виртуальная машина. У них нет такого количества проблем, как у сетевого хранилища. Локальные диски по-прежнему могут быть перегружены другими пользователями на том же сервере, но при наличии более крупного сервера вы можете быть уверены, что не используете диски совместно со слишком большим количеством людей. Даже если это менее крупный по размерам инстанс, зачастую эфемерный диск будет давать лучшую производительность, чем сетевой, пока другие арендаторы не выполняют тонны операций ввода/вывода в секунду.

Недостаток кроется в названии: эти диски эфемерны. Если ваш инстанс EC2 выйдет из строя, нет гарантии, что вы попадете в тот же ящик при перезапуске инстанса, и тогда ваши данные исчезнут.

Таким образом, эфемерные диски следует использовать с осторожностью. Вы должны убедиться, что не храните важные или нереплицированные данные на этих дисках. В частности, не размещайте на этих временных дисках журнал или свою базу данных в сетевом хранилище. В целом воспринимайте эфемерные диски как медленный кеш, а не как быстрый диск и используйте их соответственно.

Конфигурирование настроек системы

Существует несколько системных настроек, которые могут помочь MongoDB работать более плавно, в основном они касаются доступа к диску и памяти. В этом разделе описывается каждый из этих параметров и способы их настройки.

Отключение архитектуры неравномерного доступа к памяти

Когда у компьютеров был один ЦП, все ОЗУ в основном были одинаковыми с точки зрения времени доступа. Поскольку у компьютеров стало появляться больше процессоров, инженеры поняли, что наличие всей памяти на одинаковом расстоянии от каждого процессора (как показано на рис. 24.1) было менее эффективным, чем наличие у каждого процессора некой памяти, которая особенно близка к нему и к которой этот конкретный процессор может быстро получить доступ (рис. 24.2). Такая архитектура, где у каждого процессора имеется своя «локальная» память, называется *архитектурой неравномерного доступа к памяти* (NUMA).

Во многих приложениях архитектура неравномерного доступа к памяти работает хорошо: часто процессорам нужны разные данные, потому что они запускают разные программы. Тем не менее для баз данных в целом и для MongoDB в частности это ужасно, потому что базы данных имеют иные шаблоны доступа к памяти, в отличие от других типов приложений.

MongoDB использует огромный объем памяти и должна иметь возможность доступа к памяти, которая является «локальной» для других процессоров. Однако стандартные настройки архитектуры неравномерного доступа к памяти во многих системах затрудняют это.

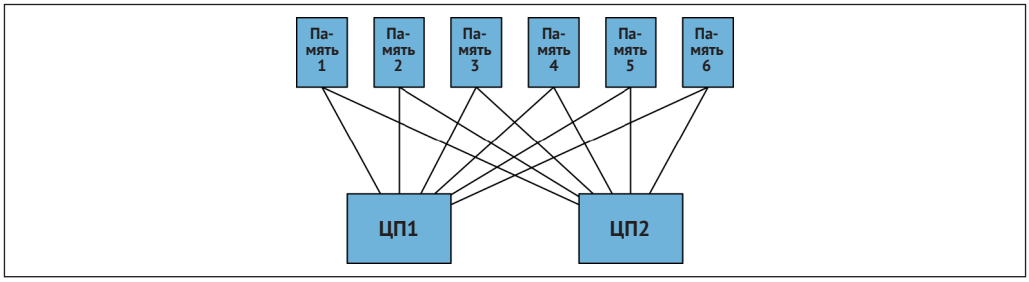

Рис. 24.1. Архитектура с равномерным доступом к памяти: у каждого процессора одинаковый доступ к памяти

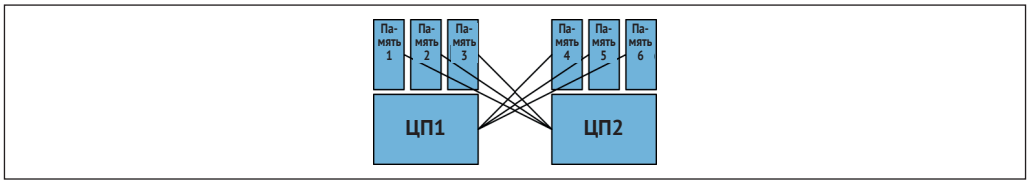

Рис. 24.2. Архитектура с неравномерным доступом к памяти: определенная память подключена к ЦП, что обеспечивает более быстрый доступ ЦП к этой памяти; процессоры по-прежнему могут получать доступ к памяти других процессоров, но это дороже, чем доступ к их собственной

Процессоры предпочитают использовать связанную с ними память, а процессы, как правило, предпочитают один процессор другим. Это означает, что память часто заполняется неравномерно, потенциально давая вам один процессор, использующий 100 % своей локальной памяти, а другие процессоры используют только часть памяти, как показано на рис. 24.3.

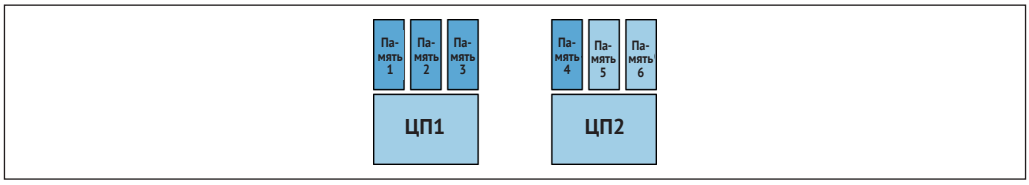

Рис. 24.3. Пример использования памяти в системе с архитектурой неравномерного доступа к памяти

В сценарии, показанном на рис. 24.3, предположим, что ЦП1 нужны некие данные, которых еще нет в памяти. Он должен использовать свою локальную память для данных, у которых еще нет «дома», но его локальная память заполнена. Таким образом, он должен убрать какие-то данные из своей локальной памяти, чтобы освободить место для новых данных,

даже если в памяти, подключенной к ЦП2, остается достаточно места! Этот процесс приводит к тому, что MongoDB работает намного медленнее, чем ожидалось, поскольку у нее имеется только часть доступной памяти, от той, что должна быть. MongoDB в большей степени предпочитает полуэффективный доступ к большему количеству данных по сравнению с чрезвычайно эффективным доступом к меньшему количеству данных.

При запуске серверов и клиентов MongoDB на оборудовании с архитектурой неравномерного доступа к памяти необходимо настроить политику чередования памяти таким образом, чтобы не хост работал в стиле NUMA. MongoDB проверяет настройки NUMA во время запуска при развертывании на компьютерах с Linux и Windows. Если конфигурация NUMA может снизить производительность, MongoDB выводит предупреждение.

В Windows чередование памяти должно быть включено в BIOS компьютера. Обратитесь к документации своей системы для получения подробной информации.

При запуске MongoDB в Linux вы должны отключить восстановление зоны в настройках *sysctl*, используя одну из следующих команд:

```
echo 0 | sudo tee /proc/sys/vm/zone_reclaim_mode

    sudo sysctl -w vm.zone_reclaim_mode=0
```

Затем вы должны использовать утилиту *numactl* для запуска своих экземпляров *mongod*, включая конфигурационные серверы, экземпляры *mongos* и всех клиентов. Если у вас нет команды `numactl`, обратитесь к документации по вашей операционной системе, чтобы установить пакет *numactl*.

Следующая команда демонстрирует, как запустить экземпляр MongoDB с помощью *numactl*:

```
numactl --interleave=all <путь> <параметры>
```

<путь> – это путь к программе, которую вы запускаете, а *<параметры>* – это любые необязательные аргументы для передачи программе.

Чтобы полностью отключить поведение NUMA, вы должны выполнить обе операции. Для получения дополнительной информации обратитесь к документации (https://oreil.ly/cm-_D).

Упреждающее чтение

Упреждающее чтение – это оптимизация, при которой операционная система считывает с диска больше данных, чем было запрошено. Это полезно, потому что большинство рабочих нагрузок, которые обрабатывают компьютеры, являются последовательными: если вы загружаете первые 20 МБ видео, вам, вероятно, понадобятся следующие несколько мегабайтов. Таким образом, система будет читать с диска больше данных, чем вы

на самом деле запрашиваете и храните их в памяти на случай, если они вам понадобятся в ближайшее время.

Для подсистемы хранения WiredTiger вы должны установить упреждающее чтение на 8 до 32 независимо от типа носителя (вращающийся диск, твердотельный накопитель и т. д.). Более высокая цифра выгодна для последовательных операций ввода/вывода, но, поскольку шаблоны доступа к диску в MongoDB обычно являются случайными, более высокое значение обеспечивает ограниченное преимущество и даже может привести к снижению производительности. Для большинства рабочих нагрузок упреждающее чтение от 8 до 32 обеспечивает оптимальную производительность MongoDB.

В целом нужно устанавливать значение упреждающего чтения в этом диапазоне, если только тестирование не покажет, что более высокое значение измеримо, повторяемо и надежно выгодно. Служба поддержки MongoDB Professional может дать советы и руководство касательно ненулевых конфигураций упреждающего чтения.

Отключение THP

THP (Transparent Huge Pages) вызывает похожие проблемы, связанные с высокими показателями упреждающего чтения. Не используйте эту функцию, только если:

- все ваши данные не помещаются в память;
- у вас нет планов, чтобы оно когда-либо выходило за пределы памяти.

MongoDB нужно подгрузить множество маленьких кусочков памяти, поэтому использование THP может привести к увеличению дискового ввода/вывода.

Системы перемещают данные с диска в память и обратно по странице. Страницы, как правило, составляют пару килобайт (по умолчанию x86 – 4096 байт). Если у машины много гигабайтов памяти, отслеживание каждой из этих (относительно крошечных) страниц может быть медленнее, чем просто отслеживание нескольких страниц с большей гранулярностью. THP – это решение, которое позволяет вам иметь страницы размером до 256 МБ (на архитектурах IA-64). Однако его использование означает, что вы храните мегабайты данных из одного раздела диска в памяти. Если ваши данные не помещаются в ОЗУ, то при загрузке более крупных фрагментов с диска ваша память просто быстро заполнится данными, которые необходимо будет снова выгрузить. Кроме того, сброс любых изменений на диск будет выполняться медленнее, поскольку диск должен записывать мегабайты «грязных» данных, а не несколько килобайтов.

THP был фактически разработан для поддержки баз данных, поэтому это может удивить опытных администраторов баз данных. Однако MongoDB имеет тенденцию выполнять намного менее последовательный доступ к диску по сравнению с реляционными базами данных.

В Windows они называются «Большие страницы» (Large Pages), а не «Огромные страницы» (Huge Pages). В некоторых версиях Windows эта функция включена по умолчанию, а в некоторых нет, поэтому проверьте и убедитесь, что она выключена.

Выбор алгоритма планирования

Контроллер диска получает запросы от операционной системы и обрабатывает их в порядке, определяемом алгоритмом планирования. Иногда изменение этого алгоритма может улучшить производительность диска. Для другого оборудования и рабочих нагрузок это может не иметь значения. Лучший способ решить, какой алгоритм использовать, – это протестировать их самим на своей рабочей нагрузке. Deadline и CFS, как правило, являются хорошим выбором.

Есть несколько ситуаций, когда планировщик noop (сокращение от «no-op») является лучшим вариантом. Если вы находитесь в виртуализированной среде, используйте его. Этот планировщик в основном передает операции через базовый контроллер диска как можно быстрее. Он быстрее всех делает это и позволяет реальному дисковому контроллеру справиться с любым переупорядочением, которое должно произойти.

Точно так же планировщик noop обычно является лучшим выбором для твердотельных накопителей. У твердотельных накопителей нет таких же проблем с локальностью, как у вращающихся дисков.

Наконец, если вы применяете контроллер RAID с кешированием, используйте noop. Кеш ведет себя как твердотельный накопитель и будет эффективно распространять операции записи на диск.

Если вы находитесь на физическом сервере, который не виртуализирован, операционная система должна использовать планировщик deadline. Он ограничивает максимальное время задержки на запрос и поддерживает разумную пропускную способность диска, которая лучше всего подходит для приложений баз данных с интенсивным использованием диска.

Вы можете изменить алгоритм планирования, установив параметр `--elevator` в своей конфигурации загрузки.

Параметр называется *elevator* (лифт), потому что планировщик ведет себя подобно лифту, который подбирает людей (запросы ввода/вывода) с разных этажей (процессы/время) и высаживает там, где им нужно, можно сказать, оптимальным способом.

Часто все алгоритмы работают довольно хорошо; большой разницы между ними не наблюдается.

Отключаем отслеживание времени доступа

По умолчанию система отслеживает время последнего доступа к файлам. Поскольку файлы данных, используемые MongoDB, очень загружены, вы можете повысить производительность, отключив это отслеживание. В Linux это можно сделать, изменив параметр `atime` на `noatime` в */etc/fstab*:

```
/dev/sda7 /data xfsf rw,noatime 1 2
```

Нужно повторно смонтировать устройство, чтобы изменения вступили в силу.

`atime` более важен для старых ядер (например, ext3); более новые по умолчанию используют `relatime`, который обновляется менее агрессивно. Также имейте в виду, что параметр `noatime` может повлиять на другие программы, использующие раздел, такие как *mutt* или инструменты резервного копирования.

Аналогично, в Windows вы должны установить опцию `disablelastaccess`. Чтобы отключить запись времени последнего доступа, выполните это:

```
C:\> fsutil behavior set disablelastaccess 1
```

Нужно перезагрузиться, чтобы настройки вступили в силу. Установка этого параметра может повлиять на удаленную подсистему хранения, но вам, вероятно, не следует использовать службу, которая в любом случае автоматически перемещает ваши данные на другие диски.

Изменение ограничений

Есть два ограничения, которые MongoDB имеет тенденцию преодолевать: количество потоков, которые процессу разрешено порождать, и количество дескрипторов файлов, которые процессу разрешено открывать. Оба они обычно должны быть неограниченными.

Всякий раз, когда сервер MongoDB принимает соединение, он порождает поток для обработки любой активности этого соединения. Поэтому если у вас есть 3000 подключений к базе данных, у базы данных будет 3000 ра-

ботающих потоков (плюс несколько других потоков для не связанных с клиентом задач). В зависимости от конфигурации вашего сервера приложений ваш клиент может порождать от десятка до тысяч соединений с MongoDB.

Если ваш клиент будет динамически создавать больше дочерних процессов при увеличении трафика (большинство серверов приложений будут делать это), важно убедиться, что эти процессы не настолько многочисленны, чтобы иметь возможность максимально ограничить пределы MongoDB. Например, если у вас есть 20 серверов приложений, каждому из которых разрешено порождать 100 дочерних процессов, а каждый дочерний процесс может порождать 10 потоков, которые все подключаются к MongoDB, это может привести к появлению 20 × 100 × 10 = 20 000 соединений при пиковом трафике. MongoDB, вероятно, будет не очень рада появлению десятков тысяч потоков и, если у вас не хватает потоков на процесс, просто начнет отказывать в новых соединениях.

Другим ограничением для изменения является количество файловых дескрипторов, которое MongoDB разрешено открывать. Каждое входящее и исходящее соединение использует файловый дескриптор, поэтому только что упомянутый шквал клиентских соединений породит 20 000 открытых файловых дескрипторов.

mongos, в частности, имеет тенденцию создавать соединения со множеством шардов. Когда клиент подключается к процессу *mongos* и делает запрос, тот открывает соединения со всеми шардами, необходимыми для выполнения этого запроса. Таким образом, если у кластера 100 шардов, а клиент подключается к *mongos* и пытается запросить все его данные, *mongos* должен открыть 100 соединений: по одному на каждый шард, что быстро может привести к взрывному числу соединений, как можно себе представить из предыдущего примера. Предположим, что произвольно сконфигурированный сервер приложений установил сотню подключений к процессу *mongos*. Это можно транслировать на 100 входящих соединений × 100 шардов = 10 000 соединений с шардами! (Это предполагает нецелевой запрос для каждого соединения, что не очень хорошо, поэтому это несколько экстремальный пример.)

Таким образом, нужно внести несколько поправок. Многие целенаправленно конфигурируют процессы *mongos*, чтобы разрешить только определенное количество входящих соединений, используя опцию `maxConns`. Это хороший способ убедиться, что ваш клиент ведет себя хорошо.

Также следует увеличить ограничение на количество дескрипторов файлов, поскольку значение по умолчанию (обычно 1024) просто слишком мало. Установите максимальное количество файловых дескрипторов на неограниченное (https://oreil.ly/oTGLL) или, если вы нервничаете по этому поводу, 20 000. Каждая система имеет свой способ изменения этих ограни-

чений, но в целом убедитесь, что вы изменили и жесткое ограничение, и мягкое. Жесткое ограничение применяется ядром и может быть изменено только администратором, тогда как мягкое ограничение настраивается пользователем.

Если максимальное количество соединений остается на уровне 1024, Cloud Manager предупредит вас об этом, показывая хост, подсвеченный желтым цветом в списке хостов. Если проблема связана с низкими лимитами, то на вкладке **Last ping** (Последний пинг) должно отобразиться сообщение, подобное тому, что показано на рис. 24.4.

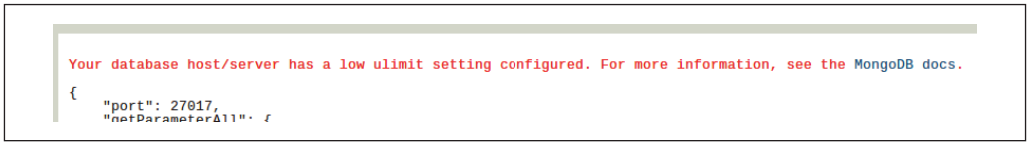

Рис. 24.4. Предупреждение о низком лимите (дескрипторы файлов) в Cloud Manager

Даже если вы не используете шарды и у вас есть приложение, которое применяет только небольшое количество соединений, рекомендуется увеличить жесткие и мягкие лимиты как минимум до 4096. Это помешает MongoDB предупредить вас и даст вам немного пространства, на всякий случай.

Конфигурирование сети

В этом разделе описывается, какие серверы к каким другим серверам должны быть подключены. Часто по соображениям сетевой безопасности (и чувствительности) вы, возможно, захотите ограничить подключение серверов MongoDB. Обратите внимание, что многосерверные развертывания в MongoDB должны обрабатывать сети, где произошел сбой, или те, что недоступны, но это не рекомендуется в качестве общепринятой стратегии развертывания.

В случае с автономным сервером клиенты должны иметь возможность подключаться к *mongod*.

Члены набора реплик должны иметь возможность устанавливать соединения с любым другим членом. Клиенты должны иметь возможность подключаться ко всем членам, которые не являются скрытыми и не являются арбитрами. В зависимости от конфигурации сети члены также могут пытаться подключаться друг к другу, поэтому вы должны разрешить серверам *mongod* создавать соединения друг с другом.

С шардингом все немного сложнее. Есть четыре компонента: серверы *mongos*, шарды, конфигурационные серверы и клиенты. Связность можно обобщить в следующих трех пунктах:

- клиент должен иметь возможность подключиться к процессу *mongos;*
- процесс *mongos* должен иметь возможность подключаться к шардам и конфигурационным серверам;
- шард должен иметь возможность подключаться к другим шардам и конфигурационным серверам.

Полная схема подключений описана в табл. 24.1.

Таблица 24.1. Связность в шардинге

Связность к типу сервера	От типа сервера *mongos*	Шард	Конфигурационный сервер	Клиент
mongos	Не требуется	Не требуется	Не требуется	Требуется
Шард	Требуется	Требуется	Не требуется	Не рекомендуется
Конфигурационный сервер	Требуется	Требуется	Не требуется	Не рекомендуется
Клиент	Не требуется	Не требуется	Не требуется	Не относится к MongoDB

В таблице есть три возможных значения.

«Требуется» означает, что связность между этими двумя компонентами необходима для шардинга, чтобы все работало как было задумано. MongoDB попытается изящно ухудшить качество, если потеряет эти соединения из-за проблем с сетью, но вам не следует специально настраивать ее таким образом.

«Не требуется» означает, что эти два элемента никогда не взаимодействуют в указанном направлении, поэтому соединения не требуется.

«Не рекомендуется» означает, что эти два элемента никогда не должны взаимодействовать, но из-за ошибки пользователя такое возможно. Например, рекомендуется, чтобы клиенты подключались только к процессам *mongos*, а не к шардам, чтобы клиенты случайно не делали запросы напрямую к шардам. Точно так же клиенты не должны иметь прямого доступа к конфигурационным серверам, чтобы они не могли случайным образом изменить данные конфигурации.

Обратите внимание, что процессы mongos и шарды взаимодействуют с конфигурационными серверами, но эти серверы не устанавливают соединения ни с кем, даже друг с другом.

Шарды должны обмениваться данными во время миграции: они соединяются друг с другом напрямую для передачи данных.

Как упоминалось ранее, члены набора реплик, которые образуют шарды, должны иметь возможность подключаться друг к другу.

Наводим порядок в системе

В этом разделе рассматриваются некоторые распространенные проблемы, о которых вам следует знать перед развертыванием.

Синхронизация часов

В целом безопаснее всего, чтобы часы ваших систем находились в пределах секунды друг от друга. Наборы реплик должны быть в состоянии справиться практически с любой рассинхронизацией часов. Шардинг может справиться с рассинхронизацией (если она превысит несколько минут, вы увидите предупреждения в журналах), но лучше минимизировать риски. Синхронизация часов также облегчает понимание того, что происходит из журналов.

Можно синхронизировать часы, используя команду *w32tm* в Windows и демон *ntp* в Linux.

OOM Killer

В редких случаях MongoDB выделяет достаточно памяти, чтобы она стала целью компонента ядра Linux – OOM Killer. Особенно часто это происходит во время построения индекса, поскольку это один из немногих случаев, когда резидентная память MongoDB должна создавать нагрузку на систему.

Если ваш процесс MongoDB внезапно останавливается, при этом никаких сообщений об ошибках или завершении работы в журналах нет, проверьте в каталоге */var/lo/messages* (или там, где ваше ядро регистрирует такие вещи), чтобы узнать, есть ли там какие-либо сообщения о прекращении работы *mongod*.

Если ядро остановило работу MongoDB из-за чрезмерного использования памяти, вы должны увидеть в журнале ядра что-то вроде этого:

```
kernel: Killed process 2771 (mongod)
kernel: init invoked oom-killer: gfp_mask=0x201d2, order=0, oomkilladj=0
```

Если вы выполняли запуск с журналированием, можете просто перезапустить *mongod* на этом этапе. Если нет, восстановите данные из резервной копии или выполните повторную синхронизацию данных из реплики.

OOM Killer особенно нервничает, если у вас нет области подкачки и у вас заканчивается нехватка памяти, поэтому хороший способ предотвратить его переполнение – настроить скромный объем подкачки. Как упоминалось ранее, MongoDB никогда не должна использовать его, но это делает OOM Killer счастливым.

Если OOM Killer останавливает процесс *mongos*, можете просто перезапустить его.

Отключите периодические задачи

Убедитесь, что у вас нет никаких заданий cron, антивирусных сканеров или демонов, которые могли бы периодически появляться и воровать ресурсы. Один из виновников – автоматическое обновление менеджеров пакетов. Эти программы оживут, съедят тонну оперативной памяти и процессора, а затем исчезнут. Вряд ли вам понравится, если такое произойдет на своем производственном сервере.

Приложение A

Установка MongoDB

Двоичные файлы MongoDB доступны для операционных систем Linux, macOS, Windows и Solaris. Это означает, что для большинства платформ вы можете скачать архив со страницы центра загрузки MongoDB (https://www.mongodb.com/download-center), накачать его и запустить двоичный файл.

Серверу MongoDB требуется каталог, в который он может записывать файлы базы данных, и порт, на котором он может слушать соединения. Этот раздел охватывает всю установку для двух вариантов систем: Windows и всех остальных (Linux/Unix/macOS).

Когда мы говорим об «установке MongoDB», обычно речь идет о настройке *mongod*, основного сервера базы данных. *mongod* можно использовать как автономный сервер или как член набора реплик. В большинстве случаев это будет процесс MongoDB, который вы используете.

Выбор версии

MongoDB использует довольно простую схему управления версиями: версии, где после точки идет четное число, – это стабильные версии, а версии, где после точки идет нечетное число, являются версиями, которые находятся в стадии разработки. Например, все, что начинается с 4.2, является стабильной версией, допустим 4.2.0, 4.2.1 и 4.2.8. Все, что начинается с 4.3, является версией в стадии разработки, например 4.3.0, 4.3.2 или 4.3.12. Возьмем в качестве примера версию 4.2/4.3, чтобы продемонстрировать, как работает временная шкала управления версиями.

1. Выпущена версия MongoDB 4.2.0. Это основная версия, у которой будет обширный журнал изменений проекта.
2. После того как разработчики приступают к работе над этапами для версии 4.4 (следующей основной стабильной версии), они выпускают версию 4.3.0. Это новая ветка, которая довольно похожа на 4.2.0, но, вероятно, у нее имеется одна или несколько дополнительных функций и, возможно, какие-то ошибки.
3. Поскольку разработчики продолжают добавлять функции, они выпустят версии 4.3.1, 4.3.2 и т. д. Эти версии не должны использоваться в производственных системах.

4. Некоторые незначительные исправления могут быть бэкпортированы в ветку 4.2, что приведет к появлению выпусков 4.2.1, 4.2.2 и т. д. Разработчики консервативны в отношении того, что бэкпортируется; в стабильный выпуск добавлено несколько новых функций. Обычно портируются только исправления ошибок.
5. После того как все основные этапы для 4.4.0 достигнуты, версия 4.3.7 (или любая последняя версия в стадии разработки) превратится в 4.4.0-rc0.
6. После тщательного тестирования версии 4.4.0-rc0 обычно есть пара незначительных ошибок, которые необходимо исправить. Разработчики исправляют эти ошибки и выпускают версию 4.4.0-rc1.
7. Разработчики повторяют предыдущий этап до тех пор, пока не будут обнаружены новые ошибки, а затем 4.4.0-rc2 (или любая последняя версия) переименовывается в 4.4.0.
8. Разработчики начинают заново с первого шага, увеличивая все версии на 0.2.

Можно увидеть, насколько вы близки к производственной версии, просмотрев дорожную карту главного сервера на трекере ошибок MongoDB (http://jira.mongodb.org).

Если вы работаете в производственной системе, нужно использовать стабильную версию. Если вы планируете использовать версию в стадии разработки в производственной системе, сначала спросите об этом в списке рассылки или с помощью IRC, чтобы получить совет от разработчиков.

Если вы только начинаете разработку проекта, лучше использовать версию в стадии разработки. К тому времени, когда вы развернете ее в производственной системе, вероятно, появится стабильная версия с функциями, которые вы используете (MongoDB пытается придерживаться регулярного цикла стабильных версий каждые 12 месяцев). Однако нужно сопоставить это с возможностью того, что вы можете столкнуться с ошибками сервера, а это может обескуражить нового пользователя.

Установка в Windows

Чтобы установить MongoDB в Windows, загрузите файл Windows.*msi* со страницы центра загрузки MongoDB (https://oreil.ly/nZZd0). Воспользуйтесь рекомендациями, приведенными в предыдущем разделе, чтобы выбрать правильную версию MongoDB. Когда вы нажмете на ссылку, загрузится файл с расширением *.msi*. Дважды щелкните мышью на значок файла, чтобы запустить программу установки.

Теперь вам нужно создать каталог, в который MongoDB сможет записывать файлы базы данных. По умолчанию MongoDB пытается использовать

каталог *\data\db* на текущем диске в качестве каталога данных (например, если вы запускаете сервер *mongod* на диске C: в Windows, она будет использовать *C:\Program Files\MongoDB\Server\&<VERSION>\data*). Он будет создан установщиком автоматически. Если вы решили использовать каталог, отличный от *\data\db*, вам нужно будет указать путь при запуске MongoDB. Об этом будет написано далее.

Теперь, когда у вас есть каталог данных, откройте командную строку (*cmd.exe*). Перейдите в каталог, в который вы распаковали двоичные файлы MongoDB, и выполните следующее:

```
$ C:\Program Files\MongoDB\Server\&<VERSION>\bin\mongod.exe
```

Если вы выбрали каталог, отличный от *C:\Program Files\MongoDB\Server\&<VERSION>\data*, вам нужно указать его здесь с помощью аргумента `--dbpath`:

```
$ C:\Program Files\MongoDB\Server\&<VERSION>\bin\mongod.exe \
    --dbpath C:\Documents and Settings\Username\My Documents\db
```

См. главу 21, где приводятся более распространенные параметры, или выполните команду `mongod.exe --help`, чтобы увидеть все параметры.

Установка в качестве службы

MongoDB также можно установить в качестве службы в Windows. Для этого просто запустите ее с полным путем, избегайте пробелов и используйте параметр `--install`. Например:

```
$ C:\Program Files\MongoDB\Server\4.2.0\bin\mongod.exe \
    --dbpath "\"C:\Documents and Settings\Username\My Documents\db\"" \
    --install
```

Затем ее можно запускать и останавливать из панели управления.

Установка в POSIX (Linux и Mac OS X)

Выберите версию MongoDB, основываясь на рекомендациях, приведенных в разделе «Выбор версии». Перейдите в Центр загрузки MongoDB (https://oreil.ly/XEScg) и выберите правильную версию для своей ОС.

Если вы работаете на Mac и у вас установлена MacOS Catalina версии 10.15 и выше, вы должны использовать каталог */System/Volumes/Data/db* вместо */data/db*. Эта версия внесла изменение, которое делает корневую папку доступной только для чтения и сбрасывается при перезагрузке, что приведет к потере вашей папки данных в MongoDB.

Вы должны создать каталог для базы данных, чтобы поместить туда свои файлы. По умолчанию база данных будет использовать каталог */data/db*, хотя вы можете указать любой другой каталог. Если вы создаете каталог по умолчанию, убедитесь, что у него есть правильные права на запись. Вы можете создать каталог и установить права, выполнив следующие команды:

```
$ mkdir -p /data/db
$ chown -R $USER:$USER /data/db
```

При необходимости команда `mkdir -p` создает каталог и все его родительские каталоги (то есть если каталога */data* не существует, она создает каталог */data*, а затем каталог */data/db*).

Команда `chown` меняет владельца */data/db*, чтобы ваш пользователь мог вести в него запись. Конечно, вы также можете просто создать каталог в своей домашней папке и указать, что MongoDB должна использовать его при запуске базы данных, чтобы избежать любых проблем, связанных с правами.

Распакуйте файл с расширением *.tar.gz*, загруженный из центра загрузки MongoDB:

```
$ tar zxf mongodb-linux-x86_64-enterprise-rhel62-4.2.0.tgz
$ cd mongodb-linux-x86_64-enterprise-rhel62-4.2.0
```

Теперь вы можете запустить базу данных:

```
$ bin / mongod
```

Или, если вы хотите использовать альтернативный путь к базе данных, укажите его с параметром `--dbpath`:

```
$ bin / mongod --dbpath ~ / db
```

Можно выполнить команду `mongod.exe --help`, чтобы увидеть все возможные параметры.

Установка из диспетчера пакетов

Также есть большое количество диспетчеров пакетов, которые можно применять для установки MongoDB. Если вы предпочитаете использовать один из них, существуют официальные пакеты для Red Hat, Debian и Ubuntu, а также неофициальные пакеты для многих других систем. Если вы используете неофициальную версию, убедитесь, что она устанавливает относительно новую версию.

Для macOS есть неофициальные пакеты для Homebrew и MacPorts. Чтобы использовать тэп (репозиторий) Homebrew (https://oreil.ly/9xoTe), сначала установите его, а затем требуемую версию MongoDB через Homebrew.

В следующем примере показано, как установить последнюю производственную версию MongoDB Community Edition. Можно добавить специализированный тэп в сеанс терминала MacOS:

```
$ brew tap mongodb/brew
```

Затем установите последнюю доступную производственную версию MongoDB Community Server (включая все инструменты командной строки):

```
$ brew install mongodb-community
```

Если вы выбираете версию для MacPorts, имейте в виду: для компиляции всех библиотек Boost требуется несколько часов, что является необходимым условием MongoDB. Начните загрузку и оставьте ее на ночь.

Независимо от того, какой диспетчер пакетов вы используете, рекомендуется выяснить, куда он помещает файлы журналов MongoDB, прежде чем возникнет проблема и вам нужно будет их найти. Важно убедиться, что они сохраняются должным образом до того, как возникнут какие-либо проблемы.

Приложение B

Внутреннее устройство MongoDB

Нет необходимости разбираться во внутреннем устройстве MongoDB, чтобы эффективно использовать ее, но оно может представлять интерес для разработчиков, которые хотят работать над инструментами, вносить свой вклад или просто понимать, что происходит «под капотом». В этом приложении приводятся некоторые основы. Исходный код MongoDB доступен по адресу https://github.com/mongodb/mongo.

BSON

Документы в MongoDB представляют собой абстрактную концепцию – конкретное представление документа зависит от используемого драйвера или языка. Поскольку документы широко используются для обмена данными в MongoDB, также должно быть представление документов, которое используется всеми драйверами, инструментами и процессами в экосистеме MongoDB. Это представление называется Binary JSON или BSON (никто не знает, куда исчезла буква J).

BSON – легковесный двоичный формат, способный представлять любой документ MongoDB в виде строки байтов. База данных понимает BSON, и это формат, в котором документы сохраняются на диск.

Когда драйверу дается документ для вставки, использования в качестве запроса и т. д., он будет кодировать этот документ в формат BSON перед отправкой его на сервер. Аналогично, документы, возвращаемые клиенту с сервера, отправляются в виде строк BSON. Эти данные BSON декодируются драйвером в его собственное представление документа, а затем возвращаются клиенту.

Формат BSON имеет три основные цели:

Эффективность

BSON разработан для эффективного представления данных без использования большого дополнительного пространства. В худшем случае BSON немного менее эффективен, чем JSON, а в лучшем слу-

чае (например, при хранении двоичных данных или больших чисел) он намного эффективнее.

Проходимость

В некоторых случаях BSON жертвует эффективностью пространства, чтобы облегчить прохождение формата. Например, строковые значения имеют префикс длины, а не используют терминатор для обозначения конца строки. Это полезно, когда серверу MongoDB необходимо проанализировать документы.

Производительность

Наконец, BSON разработан для быстрого кодирования и декодирования. Он использует представления в стиле C для типов, с которыми можно быстро работать в большинстве языков программирования.

Точную спецификацию BSON *см.* по адресу http://www.bsonspec.org.

Проводной протокол

Драйверы обращаются к серверу MongoDB с помощью облегченного проводного протокола TCP/IP. Протокол задокументирован на сайте документации MongoDB (https://oreil.ly/rVJAr), но в основном он состоит из тонкой оболочки вокруг данных BSON. Например, сообщение вставки состоит из 20 байт данных заголовка (которые включают в себя код, дающий серверу указание выполнить вставку и длину сообщения, имя коллекции, куда нужно вставить документы, и список документов BSON для вставки).

Файлы данных

Внутри каталога данных MongoDB – по умолчанию это */data/db/* – для каждой коллекции и каждого индекса будет храниться отдельный файл. Имена файлов не соответствуют именам коллекций или индексов, но вы можете использовать функцию `stats` в оболочке *mongo*, чтобы идентифицировать связанный файл для конкретной коллекции. В поле `"wiredTiger.uri"` будет содержаться имя файла, которое нужно искать в каталоге данных MongoDB.

Использование функции `stats` в базе данных *sample_mflix* коллекции *movies* в качестве результата в поле `"wiredTiger.uri"` дает «collection-14--2146526997547809066»:

```
>db.movies.stats()
{
    "ns" : "sample_mflix.movies",
    "size" : 65782298,
```

```
    "count" : 45993,
    "avgObjSize" : 1430,
    "storageSize" : 45445120,
    "capped" : false,
    "wiredTiger" : {
        "metadata" : {
            "formatVersion" : 1
    },
    "creationString" : "access_pattern_hint=none,allocation_size=4KB,\
app_metadata=(formatVersion=1),assert=(commit_timestamp=none,\
read_timestamp=none),block_allocation=best,\
block_compressor=snappy,cache_resident=false,checksum=on,\
colgroups=,collator=,columns=,dictionary=0,\
encryption=(keyid=,name=),exclusive=false,extractor=,format=btree,\
huffman_key=,huffman_value=,ignore_in_memory_cache_size=false,\
immutable=false,internal_item_max=0,internal_key_max=0,\
internal_key_truncate=true,internal_page_max=4KB,key_format=q,\
key_gap=10,leaf_item_max=0,leaf_key_max=0,leaf_page_max=32KB,\
leaf_value_max=64MB,log=(enabled=true),lsm=(auto_throttle=true,\
bloom=true,bloom_bit_count=16,bloom_config=,bloom_hash_count=8,\
bloom_oldest=false,chunk_count_limit=0,chunk_max=5GB,\
chunk_size=10MB,merge_custom=(prefix=,start_generation=0,suffix=),\
merge_max=15,merge_min=0),memory_page_image_max=0,\
memory_page_max=10m,os_cache_dirty_max=0,os_cache_max=0,\
prefix_compression=false,prefix_compression_min=4,source=,\
split_deepen_min_child=0,split_deepen_per_child=0,split_pct=90,\
type=file,value_format=u",
    "type" : "file",
    "uri" : "statistics:table:collection-14--2146526997547809066",
...
}
```

Детали файла можно проверить в каталоге данных MongoDB:

```
ls -alh collection-14--2146526997547809066.wt
-rw------- 1 braz staff 43M 28 Sep 23:33 collection-14--2146526997547809066.wt
```

Можно использовать фреймворк агрегации, чтобы найти URI для каждого индекса в определенной коллекции:

```
db.movies.aggregate([{
$collStats:{storageStats:{}}}]).next().storageStats.indexDetails
{
    "_id_" : {
    "metadata" : {
        "formatVersion" : 8,
```

```
            "infoObj" : "{ \"v\" : 2, \"key\" : { \"_id\" : 1 },\
            \"name\" : \"_id_\", \"ns\" : \"sample_mflix.movies\" }"
        },
        "creationString" : "access_pattern_hint=none,allocation_size=4KB,\
    app_metadata=(formatVersion=8,infoObj={ \"v\" : 2, \"key\" : \
    { \"_id\" : 1 },\"name\" : \"_id_\", \"ns\" : \"sample_mflix.movies\" }),\
    assert=(commit_timestamp=none,read_timestamp=none),block_allocation=best,\
    block_compressor=,cache_resident=false,checksum=on,colgroups=,collator=,\
    columns=,dictionary=0,encryption=(keyid=,name=),exclusive=false,extractor=,\
    format=btree,huffman_key=,huffman_value=,ignore_in_memory_cache_size=false,\
    immutable=false,internal_item_max=0,internal_key_max=0,\
    internal_key_truncate=true,internal_page_max=16k,key_format=u,key_gap=10,\
    leaf_item_max=0,leaf_key_max=0,leaf_page_max=16k,leaf_value_max=0,\
    log=(enabled=true),lsm=(auto_throttle=true,bloom=true,bloom_bit_count=16,\
    bloom_config=,bloom_hash_count=8,bloom_oldest=false,chunk_count_limit=0,\
    chunk_max=5GB,chunk_size=10MB,merge_custom=(prefix=,start_generation=0,\
    suffix=),merge_max=15,merge_min=0),memory_page_image_max=0,\
    memory_page_max=5MB,os_cache_dirty_max=0,os_cache_max=0,\
    prefix_compression=true,prefix_compression_min=4,source=,\
    split_deepen_min_child=0,split_deepen_per_child=0,split_pct=90,type=file,\
    value_format=u",
        "type" : "file",
        "uri" : "statistics:table:index-17--2146526997547809066",
    ...
        "$**_text" : {
    ...
        "uri" : "statistics:table:index-29--2146526997547809066",
    ...
        "genres_1_imdb.rating_1_metacritic_1" : {
    ...
        "uri" : "statistics:table:index-30--2146526997547809066",
    ...
    }
```

WiredTiger хранит каждую коллекцию или индекс в одном произвольно большом файле. Единственными ограничениями, которые влияют на потенциальный максимальный размер этого файла, являются ограничения размера файловой системы.

WiredTiger записывает новую копию полного документа всякий раз, когда этот документ обновляется. Старая копия на диске помечается для повторного использования и в конечном итоге будет перезаписана в будущем, обычно во время следующей контрольной точки. Таким образом перерабатывается пространство, используемое в файле WiredTiger. Можно выполнить команду `compact`, чтобы переместить данные в этом файле в на-

чало, оставляя пустое место в конце. С регулярными интервалами WiredTiger удаляет это избыточное пустое пространство путем усечения файла. В конце процесса сжатия избыточное пространство возвращается в файловую систему.

Пространства имен

Каждая база данных организована в *пространства имен*, которые отображаются в файлы WiredTiger. Эта абстракция отделяет внутренние детали подсистемы хранения от уровня запросов MongoDB.

Подсистема хранения WiredTiger

Подсистема хранения по умолчанию для MongoDB – это WiredTiger. Когда сервер запускается, он открывает файлы данных и начинает процессы копирования контрольных точек и журналирования. Он работает в сочетании с операционной системой, чья ответственность сосредоточена на подкачке и удалении данных, а также сбросе данных на диск. Эта подсистема хранения имеет несколько важных свойств:

- сжатие включено по умолчанию для коллекций и индексов. По умолчанию используется алгоритм сжатия snappy от компании Google. Другие варианты включают в себя Zstandard (zstd) и zlib от Facebook или отсутствие сжатия вообще. Это сводит к минимуму использование хранилища в базе данных за счет дополнительных требований к процессору;
- параллелизм на уровне документов позволяет одновременно обновлять различные документы от нескольких клиентов в коллекции. WiredTiger использует MultiVersion Concurrency Control (MVCC) для изоляции операций чтения и записи, чтобы клиенты могли видеть согласованное представление данных на момент времени в начале операции;
- при копировании контрольных точек создается согласованный снимок данных на момент времени. Оно происходит каждые 60 секунд и включает в себя запись всех данных в снимке на диск и обновление связанных метаданных;
- журналирование и копирование контрольных точек гарантируют, что не будет никакого момента времени, когда данные могут быть потеряны в случае сбоя процесса *mongod*. WiredTiger использует журнал с упреждающей записью, в котором хранятся изменения до того, как они будут применены.

Об авторах

Шеннон Брэдшоу – вице-президент по образованию в MongoDB. Он руководит командами MongoDB Documentation и MongoDB University. Эти команды разрабатывают и поддерживают большинство учебных ресурсов по MongoDB, используемых сообществом. Шеннон имеет докторскую степень в области компьютерных наук, которую он получил в Северо-Западном университете. До работы в MongoDB Шеннон был профессором информатики, специализирующимся на информационных системах и взаимодействии человека и информации.

Йон Брэзил – старший инженер по учебным программам в MongoDB. Он работает над онлайн- и инструктируемыми учебными продуктами, предоставляемыми через Университет MongoDB, и ранее занимал различные должности в организации поддержки технических служб в MongoDB. Эойн имеет докторскую степень и степень магистра компьютерных наук от Университета Лимерика и степень магистра по коммерциализации технологий от Национального университета Ирландии в Голуэе. До прихода в MongoDB он руководил командами в сфере мобильных услуг и высокопроизводительных вычислений в секторе научных исследований.

Кристина Ходоров – инженер-программист, пять лет работавшая над ядром MongoDB. Она руководила разработкой набора реплик MongoDB, а также занималась написанием драйверов для PHP и Perl. Выступала с докладами о MongoDB на встречах и конференциях по всему миру и ведет блог на технические темы: http://www.kchodorow.com. В настоящее время работает в Google.

Об изображении на обложке

Животное, изображенное на обложке книги, – *мангустовый лемур*, член весьма разнообразной группы приматов, эндемичных для Мадагаскара. Предполагается, что предки этих лемуров случайно попали на Мадагаскар из Африки (это путешествие длиной не менее 350 миль) на плоту около 65 млн лет назад. Свободные от конкуренции с другими африканскими видами (такими как обезьяны и белки), они приспособились заполнять самые разнообразные экологические ниши, и на сегодняшний день известно почти 100 видов лемуров. Своим названием, которое происходит от слова «лемур» (злой дух в римской мифологии), они обязаны загадочным звукам, который они издают, ночной активности и горящим глазам. Малагасийская культура также связывает лемуров со сверхъестественным, считая их то душами предков, то источником табу, то духами, жаждущими мести. Некоторые деревни идентифицируют определенный вид лемура как предка своей группы.

Мангустовые лемуры (*Eulemur mongoz*) – животные среднего размера длиной примерно от 12 до 18 дюймов и весом от 3 до 4 фунтов. Густой хвост добавляет дополнительные 16–25 дюймов. У самок и молодых лемуров белая борода, а у самцов рыжая борода и щеки. Мангустовые лемуры питаются фруктами и цветами и выполняют роль опылителей для некоторых растений; особенно они любят нектар хлопкового дерева. Они также могут есть листья и насекомых.

Мангустовые лемуры обитают в сухих лесах на северо-западе Мадагаскара. Один из двух видов лемуров обнаружен за пределами Мадагаскара, лемуры, которые живут на Коморских островах (куда, как предполагается, они были ввезены людьми). Они обладают необычным качеством: эти животные ведут катемеральный образ жизни (совмещают дневную и ночную активность), меняя свое поведение в соответствии с влажным и сухим сезонами. Мангустовые лемуры находятся под угрозой потери среды обитания и классифицируются как уязвимый вид.

Многие животные, изображенные на обложках книг издательства O'Reilly, находятся под угрозой исчезновения; все они важны для мира.

Иллюстрация на обложке выполнена Карен Монтгомери на основе черно-белой гравюры из многотомника *The Royal Natural History* Ричарда Лидеккера.

Предметный указатель

Книги издательства «ДМК ПРЕСС» можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
Тел.: **+7(499) 782-38-89**. Электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать
адрес (полностью), по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Шеннон Брэдшоу, Йон Брэзил, Кристина Ходоров

MongoDB: полное руководство

Мощная и масштабируемая система управления базами данных

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Беликов Д. А.*
Редактор *Соломонов Д. В.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100$^{1}/_{16}$.
Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 43,88. Тираж 200 экз.
Отпечатано в ООО «Принт-М»
142300, Московская обл., Чехов, ул. Полиграфистов, 1

Веб-сайт издательства: **www.dmkpress.com**