Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Информационные технологии»

НЕРЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ
для студентов специальностей 1-40 05 01
«Информационные системы и технологии
(по направлениям)» и 1-40 80 04
«Информатика и технологии программирования»
дневной и заочной форм обучения

Электронный аналог печатного издания

Гомель 2022

УДК 004.65(075.8)
ББК 32.972.134я73
Н54

Рекомендовано к изданию научно-методическим советом факультета автоматизированных и информационных систем ГГТУ им. П. О. Сухого (*протокол № 10 от 25.06.2021 г.*)

Учебно-методическое пособие выполнено в ходе реализации проекта MaCICT(Modernisation of Master Curriculum in Information Computer Technologies) в рамках Erasmus + Программы Европейского Союза.

Составители: *О. Д. Асенчик, В. И. Токочаков*

Рецензент: доц. каф. «Информатика» ГГТУ им. П. О. Сухого
канд. техн. наук, доц. *Д. В. Прокопенко*

Н54 **Нереляционные** базы данных : учеб.-метод. пособие для студентов спецальностей 1-40 05 01 «Информационные системы и технологии (по направлениям)» и 1-40 80 04 «Информатика и технологии программирования» днев. и заоч. форм обучения / сост.: О. Д. Асенчик, В. И. Токочаков. – Гомель : ГГТУ им. П. О. Сухого, 2022. – 256 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

ISBN 978-985-535-495-7.

Содержит семь основополагающих тем, позволяющих студентам овладеть навыками разработки и использования объектных и других нереляционных баз данных.

Для студентов специальностей 1-40 05 01 «Информационные системы и технологии (по направлениям)» и 1-40 80 04 «Информатика и технологии программирования» дневной и заочной форм обучения.

УДК 004.65(075.8)
ББК 32.972.134я73

ISBN 978-985-535-495-7

Оглавление

Глава 1. ОБЩАЯ ХАРАКТЕРИСТИКА БАЗ ДАННЫХ *NoSQL*

Базы данных *NoSQL* также называют нереляционными, или «не *SQL*», чтобы подчеркнуть тот факт, что они отличаются от реляционных баз данных (*SQL*) со строками и таблицами, но также позволяют обрабатывать огромные объемы быстро меняющихся неструктурированных данных.

Технологии *NoSQL* появились еще в 1960-х годах и носили различные названия, однако всплеск их популярности связан с тем, что картина данных меняется, а разработчикам необходимо адаптироваться к огромным объемам и массивам данных, поступающим из облака, мобильных устройств, социальных сетей и систем обработки больших данных.

Новые данные и их типы – от вирусных твитов знаменитостей до жизненно важных сведений в электронных медицинских записях – создаются с головокружительной скоростью. Базы данных *NoSQL* призваны помочь разработчикам быстро создавать системы баз данных для хранения этих данных, а также сделать их доступными для поиска, консолидации и анализа.

Базы данных *NoSQL* помогают ИТ-специалистам и разработчикам справляться с новыми задачами в условиях растущего разнообразия моделей и типов данных. Кроме того, они обеспечивают эффективную обработку непредсказуемых данных, часто с невероятно высокими скоростями. Они также обеспечивают удобный перенос баз данных в облако для имеющихся рабочих нагрузок *NoSQL*.

Преимущества баз данных *NoSQL*

Гибкая разработка: благодаря возможности реагировать на незапланированные ситуации базы данных, например, в форме значимого изменения структуры, базы данных *NoSQL* подходят для интенсивных циклов выпуска программного обеспечения, а также для быстрой и гибкой разработки приложений.

Гибкая обработка данных: позволяет менять схемы и запросы в соответствии с требованиями к данным. Хранение данных в виде агрегатов помогает быстро вносить итеративные улучшения, не проектируя схему заранее.

Масштабируемость: благодаря возможности горизонтального масштабирования, позволяющего сократить количество дорогостоящих серверов без необходимости в обновлении. Возможно масштаби-

рование для обработки больших объемов данных или хранение единой крупной базы данных в легко распределяемых кластерах серверов.

***Разнообразие типов моделей данных**:* большинство нереляционных баз данных высокой производительности также могут обрабатывать данные со сложной структурой. При этом они не привязаны к их фиксированным моделям как реляционные базы данных.

Наиболее распространенные следующие типы баз данных *NoSQL*:

- хранилища типа «ключ–значение» – пары ключей и значений формируются с помощью хэш-таблицы. Базы данных типа «ключ–значение» лучше всего подходят для тех случаев, когда ключ известен, а связанное с ним значение – нет;
- базы данных документов – это расширенная версия баз данных типа «ключ–значение», в которой целые документы упорядочиваются по группам, называемым коллекциями. Они поддерживают вложенные пары «ключ–значение» и запросы к любым атрибутам документов;
- столбчатые базы данных – базы данных на основе столбцов, широких столбцов или семейств столбцов обеспечивают эффективное хранение данных и отправку запросов к строкам разреженных данных. Их также удобно использовать для запросов к определенным столбцам базы данных;
- графовые базы данных позволяют использовать модель на основе узлов и ребер, представляющих взаимосвязь данных (например, отношения между людьми в социальной сети). Кроме того, они упрощают хранение и навигацию по сложным отношениям.

Выбор типа используемой базы данных применительно к конкретной задаче – не всегда тривиален. Нельзя утверждать, что нереляционные базы данных всегда предпочтительнее реляционных.

В табл. 1.1 даны рекомендации от *Microsoft*, которые позволяют облегчить выбор между реляционными и нереляционными базами данных применительно к определенному классу задач, решаемому приложением, использующим их.

Таблица 1.1

Рекомендации, которые позволяют облегчить выбор между реляционными и нереляционными базами

Параметр	***NoSQL* (нереляционные базы данных)**	***SQL* (реляционные базы данных)**
ПРЕДНАЗНАЧЕНИЕ	Обработка крупных, не связанных между собой, неопределенных или быстро меняющихся данных. Не зависящие от схемы данные или схема, задаваемая приложением. Приложения, для которых производительность и доступность важнее, чем строгая согласованность. Непрерывно работающие приложения, которые обслуживают пользователей по всему миру	Обработка реляционных данных с логичными и обособленными требованиями, которые можно определить заранее. Схема, которую требуется хранить, поддерживая синхронизацию между приложением и базой данных. Устаревшие системы, предназначенные для реляционных структур. Приложения, требующие использования сложных запросов или многострочных транзакций
СЦЕНАРИИ ИСПОЛЬЗОВАНИЯ	Мобильные приложения. Аналитика в реальном времени. Управление содержимым. Персонализация. Приложения Интернета вещей. Перенос баз данных	Бухгалтерские, финансовые и банковские системы. Системы управления запасами. Системы управления транзакциями
МАСШТАБИРОВАНИЕ	Горизонтальное масштабирование данных путем сегментирования между серверами	Вертикальное масштабирование данных путем повышения серверной нагрузки
МОДЕЛЬ ДАННЫХ	Типы баз данных: ключ–значение, базы данных документов, столбчатые и графовые базы данных. Хранение данных в зависимости от типа базы данных	Тип базы данных: таблицы строк, сгруппированные по связям. Использование языка SQL. Данные хранятся в виде строк таблиц. Связанные данные хранятся отдельно и объединяются для сложных запросов

Microsoft рекомендует следующий алгоритм оценки баз данных *NoSQL*:

1. Выбор модели данных.

Многие базы данных *NoSQL* рассчитаны на работу с агрегатами – коллекциями данных, с которыми пользователи взаимодействуют как с единицами. Благодаря этому они намного лучше сочетаются с современными объектно-ориентированными языками программирования. Чтобы подобрать базу данных *NoSQL*, рекомендуем сначала выбрать модель данных. После этого следует рассмотреть поддерживающие ее базы данных *NoSQL*, а также языки программирования и пакеты *SDK*, поддерживаемые каждой базой данных.

2. Анализ требований к согласованности данных.

Большинство баз данных *NoSQL* поддерживают два крайних уровня согласованности: строгую (вы получаете наиболее актуальные данные, но возможна задержка) и итоговую (быстрый отклик, но данные могут быть устаревшими). Многие базы данных *NoSQL* поддерживают и другие, промежуточные уровни согласованности. Выберите базу данных *NoSQL*, которая обеспечивает максимальную гибкость и возможности управления моделями согласованности в соответствии с требованиями задачи.

3. Анализ возможности переноса баз данных в облачное хранилище.

Благодаря распределенному характеру и горизонтальной масштабируемости базы данных *NoSQL* хорошо подходят для облачных вычислений, а также можно найти множество систем баз данных *NoSQL*, предназначенных для работы в локальных или гибридных средах.

Оценивая различные варианты облачного хранилища, нужно учитывать перечисленные ниже особенности.

Поддерживаемые модели данных. Поддерживает ли поставщик облачных служб все модели данных, которые могут вам потребоваться?

Развертывание и эксплуатация. Насколько легко развернуть базу данных и реплицировать ее в другие регионы при необходимости?

Географическое присутствие. Где находятся центры обработки данных поставщика облачных служб? Удастся ли разместить данные в нужных расположениях? Как вы будете обеспечивать соответствие международным нормам конфиденциальности данных, например регламенту *GDPR* в Европейском Союзе?

Удобство репликации. Как происходит репликация базы данных в другой географический регион?

Масштабируемость. Позволяют ли ресурсы базы данных *NoSQL* обеспечить надлежащую производительность, а также масштабирование для роста? Сможете ли вы выполнять вертикальное или горизонтальное масштабирование?

Уровень доступности. Что произойдет в случае непредвиденного сбоя? Встроены ли в службу возможности по обеспечению высокого уровня доступности и аварийного восстановления?

Уровни обслуживания. Какие уровни доступности и гарантии по задержкам предоставляются?

Экосистема. Насколько тесно база данных интегрирована с остальными компонентами облачной платформы, и можно ли быстро совмещать ее с другими компонентами для создания новых решений?

Применения систем баз данных *NoSQL*

Рассмотрим более подробно области применения систем баз данных *NoSQL*.

Модели данных на основе *NoSQL* отлично подойдут для компаний, создающих мобильные, игровые и веб-приложения, а также приложения Интернета вещей, которым необходимы гибкие, масштабируемые, высокопроизводительные и функциональные базы данных для эффективного выполнения различных задач – игр, электронной коммерции, аналитики больших данных, работы в Интернете в реальном времени и т. д.

Использование *NoSQL* позволит привлекать больше пользователей со всего мира благодаря повышенной производительности приложений с высокой доступностью и аварийным восстановлением, а также дает возможность осуществлять взаимодействие с пользователями в реальном времени.

Персонализация и предоставление рекомендации в реальном времени, удобство работы пользователей позволят осуществить базы данных *NoSQL*.

Поддержка разнообразных рабочих нагрузок Интернета вещей – современное и востребованное направление. Мгновенное и эластичное масштабирование обеспечивает непрерывный прием данных с большим количеством операций записи, а также повышенную производительность запросов для приложений Интернета вещей.

Благодаря гибким схемам и иерархическим данным базы данных *NoSQL* отлично подходят для хранения каталогов продукции, содержащих изделия с различными атрибутами.

Новый контент для привлечения пользователей игровых приложений – персонализация решений с добавлением такого содержимого, как игровая статистика, интеграция с социальными сетями и списки лидеров. Низкая задержка и эластичная масштабируемость позволяют адаптироваться к пикам трафика во время запуска игр и проведения турниров.

Создание бессерверных приложений – это удобное масштабирование приема данных, пропускной способности и объемов данных с мгновенной доступностью, автоматическим индексированием, стабильной скоростью приема и высокой производительностью запросов.

Углубленный анализ больших данных. Реализуйте машинное обучение в нужном масштабе для быстро меняющихся данных большого объема и получите больше пользы от аналитики.

Перенос имеющихся рабочих нагрузок *NoSQL* в облако позволяет затрачивать меньше времени на управление локальной базой данных. При этом можно и далее использовать имеющиеся инструменты, драйверы, библиотеки и пакеты *SDK*.

Глава 2. НЕРЕЛЯЦИОННЫЕ МОДЕЛИ ДАННЫХ

Нереляционная база данных – это база данных, в которой не используется табличная схема строк и столбцов. В этих базах данных применяется модель хранения, оптимизированная под конкретные требования типа хранимых данных. Например, данные могут храниться как простые пары «ключ–значение», документы *JSON* или граф, состоящий из ребер и вершин.

Все эти хранилища не используют реляционную модель. Кроме того, они, как правило, поддерживают определенные типы данных. Процесс запроса данных также специфический. Например, хранилища данных временных рядов рассчитаны на запросы к последовательностям данных, упорядоченных по времени, а хранилища данных графов – на анализ взвешенных связей между сущностями. Ни один из форматов не подходит в полной мере при выполнении задач управления данными о транзакциях.

Термин *NoSQL* применяется к хранилищам данных, которые не используют язык запросов *SQL*, а запрашивают данные с помощью других языков и конструкций. На практике *NoSQL* означает «нереляционная база данных», даже несмотря на то, что многие из этих баз данных поддерживают запросы, совместимые с *SQL*. Однако базовая стратегия выполнения запросов *SQL* обычно значительно отличается от применяемой в системе управления реляционной базы данных (реляционной СУБД).

Ниже описаны основные категории нереляционных баз данных, или баз данных *NoSQL*.

Хранилища данных документов

Хранилище данных документов управляет набором значений именованных строковых полей и данных объекта в сущности, которая называется *документом*. Обычно данные в этих хранилищах содержатся в виде документов *JSON*. Каждое значение поля может представлять собой скалярный элемент, например, число, или сложный объект, например, список или коллекцию типа «родитель–потомок». Данные в полях документа можно закодировать разными способами, например, в формате *XML*, *YAML*, *JSON*, *BSON*, или хранить в виде обычного текста. Поля документов доступны системе управления хранилищем, что позволяет приложению выполнять запросы и применять фильтры, основанные на значениях этих полей.

Как правило, документ содержит все данные одной сущности. Такие элементы представляют собой сущность, специфичны в применении. Например, сущность может содержать сведения о клиенте, заказе или и те и другие. Один документ может содержать сведения, которые в реляционной СУБД обычно распределяются по нескольким реляционным таблицам. Хранилище документов не обязывает использовать одинаковую структуру для всех документов. Поддержка свободной формы обеспечивает большую гибкость. Например, приложения могут хранить в документах разные данные в соответствии с текущими требованиями компании (рис. 2.1).

Key	Document
1001	{ "CustomerID": 99, "OrderItems": [{ "ProductID": 2010, "Quantity": 2, "Cost": 520 }, { "ProductID": 4365, "Quantity": 1, "Cost": 18 }], "OrderDate": "04/01/2017" }
1002	{ "CustomerID": 220, "OrderItems": [{ "ProductID": 1285, "Quantity": 1, "Cost": 120 }], "OrderDate": "05/08/2017" }

Рис. 2.1. Пример хранилища данных

Приложение может получать документы по ключу. Это уникальный идентификатор документа. Часто к нему применяется хэширование для равномерного распределения данных. Некоторые базы данных документов создают ключ автоматически; другие – позволяют указать, какой атрибут документа следует использовать в качестве ключа. Приложение также может запрашивать документы по значениям одного или нескольких полей. Некоторые базы данных документов поддерживают индексирование, что ускоряет их поиск по одному или нескольким индексированным полям.

Многие базы данных документов поддерживают обновления «на месте», то есть позволяют приложению изменять значения от-

дельных полей без перезаписи всего документа. Чтение и запись операций на нескольких полях в одном документе, как правило, являются атомными.

Столбчатые хранилища данных

Столбчатое хранилище данных, или хранилище семейств столбцов, упорядочивает данные по столбцам и строкам. Столбчатое хранилище данных в простейшей форме почти неотличимо от реляционной базы данных, по крайней мере, организационно. Преимущество столбчатого хранилища заключается в способности денормализованно структурировать разреженные данные, что связано со столбцово-ориентированным методом хранения.

Столбчатое хранилище данных можно представить в виде набора табличных данных со строками и столбцами, в которых столбцы разделяются на определенные группы или семейства столбцов. Каждое семейство столбцов включает набор логически связанных столбцов, которые обычно извлекаются или управляются в качестве единого целого. Другие данные, которые используются в других процессах, хранятся отдельно, в других семействах столбцов. В семейство столбцов можно динамически добавить новые, а строки могут быть разреженными (то есть строки не обязаны иметь значение для каждого столбца).

На рис. 2.2 представлен пример таблицы с двумя семействами столбцов: *Identity* и *Contact Info*. Данные одной сущности имеют одинаковые ключи строк во всех семействах столбцов. Такая структура, в которой строки любого объекта в семействе столбцов могут динамически изменяться, определяет преимущество этой категории хранилищ. Семейства столбцов очень хорошо подходят для хранения данных с различными схемами.

В отличие от хранилища пар «ключ–значение» и баз данных документов, большинство столбчатых баз данных упорядочивают хранимые данные с помощью самих значений ключей, а не хэш-кодов от них. Ключ строки рассматривается как первичный индекс и обеспечивает доступ на основе определенного ключа или их диапазона. Некоторые реализации позволяют создавать вторичные индексы по определенным столбцам в их семействе. Вторичные индексы позволяют получать данные по значениям столбцов, а не ключам строки.

CustomerID	Column Family: Identity
001	**First name:** Mu Bae **Last name:** Min
002	**First name:** Francisco **Last name:** Vila Nova **Suffix:** Jr.
003	**First name:** Lena **Last name:** Adamcyz **Title:** Dr.

CustomerID	Column Family: Contact Info
001	**Phone number:** 555-0100 **Email:** someone@example.com
002	**Email:** vilanova@contoso.com
003	**Phone number:** 555-0120

Рис. 2.2. Пример таблицы с двумя семействами столбцов

Все столбцы одного семейства хранятся на диске в одном файле. Каждый файл содержит определенное число строк. При использовании больших наборов данных этот подход позволяет повысить производительность за счет снижения объема данных, которые необходимо считывать с диска, когда отправляется запрос на получение нескольких столбцов за раз.

Операции чтения и записи строки обычно являются атомными в пределах одного семейства столбцов, хотя некоторые реализации обеспечивают атомарность по всему ряду, охватывающую несколько семейств столбцов.

Хранилище пар «ключ–значение»

Хранилище пар «ключ–значение» представляет собой большую хэш-таблицу. Каждое значение сопоставляется с уникальным ключом, и хранилище ключей использует его для хранения данных, применяя к нему некоторую функцию хэширования. Выбор функции хэширования должен обеспечить равномерное распределение хэшированных ключей по хранилищу данных.

Большинство хранилищ пар «ключ–значение» поддерживают только самые простые операции запроса, вставки и удаления. Чтобы частично или полностью изменить значение, приложение всегда перезаписывает существующее значение целиком. В большинстве реализаций атомарной операцией считается чтение или запись одного значения. Запись больших значений занимает относительно долгое время.

Приложение может хранить в наборе значений произвольные данные, но некоторые хранилища пар «ключ–значение» накладывают ограничения на максимальный размер значений. Программное обеспечение хранилища ничего не знает о значениях, которые в нем хранятся. Все сведения о схеме поддерживаются и применяются на уров-

не приложения. Эти значения являются большими двоичными объектами, которые хранилище извлекает и сохраняет по соответствующему ключу (рис. 2.3).

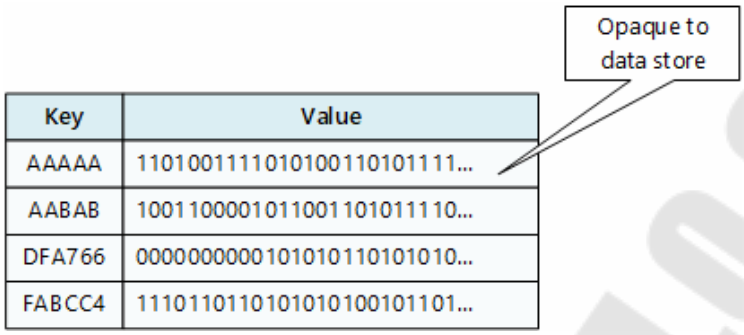

Рис. 2.3. Пример таблицы с двумя семействами столбцов

Хранилища пар «ключ–значение» рассчитаны на приложения, выполняющие простые операции поиска на основе значения ключа или диапазона ключей, однако эти хранилища не очень подходят для систем, которым нужно запрашивать данные из нескольких таблиц хранилищ пар «ключ–значение», например, присоединенные данные в нескольких таблицах.

Кроме того, хранилища пар «ключ–значение» неудобны в сценариях, где могут выполняться запросы или фильтрация по значению, а не только по ключам. Например, с реляционной базой данных можно найти запись, используя оговорку *WHERE* для фильтрации не ключевых столбцов, однако магазины ключей/значений обычно не имеют этого типа возможности поиска или если имеют возможность делать это, все значения сканируются медленно.

Одно хранилище пар «ключ–значение» очень легко масштабируется, поскольку позволяет удобно распределить данные среди нескольких узлов на разных компьютерах.

Хранилища данных графов

Хранилища данных графов управляют сведениями двух типов: узлами и ребрами. Узлы в этом случае представляют сущности, а ребра определяют связи между ними. Узлы и грани имеют свойства, которые предоставляют сведения о конкретном узле или грани, примерно как столбцы в реляционной таблице. Грани могут иметь направление, указывающее на характер связи.

Хранилища данных графов позволяют приложениям эффективно выполнять запросы, которые проходят через сеть узлов и ребер, а также позволяют анализировать связи между сущностями.

На рис. 2.4 представлены данные персонала организации, структурированные в виде графа. Сущностями здесь являются сотрудники и отделы, а грани определяют отношения подчинения и отдел, в котором работает каждый сотрудник. На этом графе грани имеют строгое направление, которое на схеме обозначено стрелками.

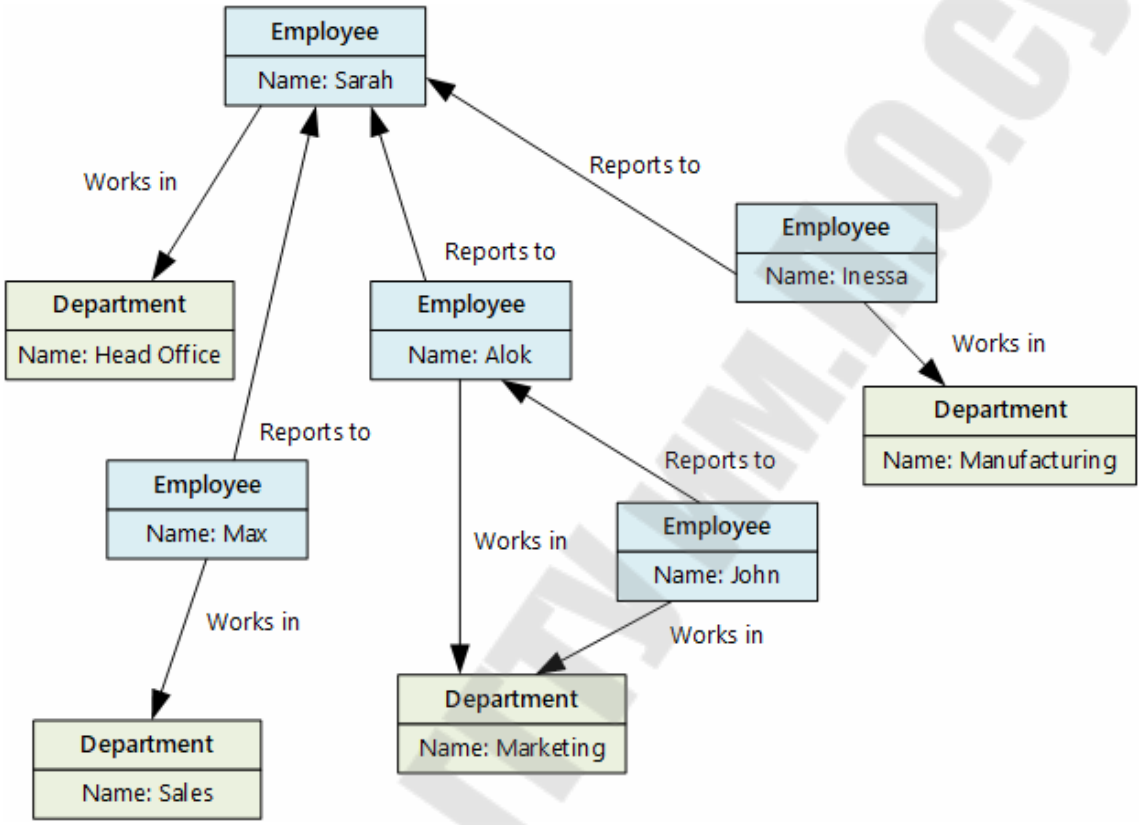

Рис. 2.4. Граф данных персонала организации

Такая структура позволяет легко выполнять такие запросы, как «найти всех сотрудников, которые прямо или косвенно подчиняются Светлане» или «найти всех, кто работает в одном отделе с Дмитрием». В больших графиках с большим количеством сущностей и взаимосвязей можно быстро выполнять сложные анализы. Многие базы данных графов предоставляют язык запросов, который можно использовать для эффективного обхода сети связей.

Хранилища данных временных рядов

Данными временных рядов называются наборы значений, которые упорядочены по времени. Соответственно, хранилища данных временных рядов оптимизированы для хранения данных именно такого типа. Хранилища данных временных рядов должны поддерживать очень большое число операций записи, так как обычно в них в

режиме реального времени собирается большой объем данных из большого количества источников (рис. 2.5). Эти хранилища также хорошо подходят для данных телеметрии, например, для сбора данных от датчиков Интернета вещей или счетчиков в приложениях или системах. Обновления в таких базах данных выполняются редко, а удаление чаще всего является массовой операцией.

timestamp	deviceid	value
2017-01-05T08:00:00.123	1	90.0
2017-01-05T08:00:01.225	2	75.0
2017-01-05T08:01:01.525	2	78.0

Рис. 2.5. Пример временного ряда

Размер отдельных записей в базе данных временных рядов обычно невелик, но этих записей очень много, а значит общий размер данных быстро увеличивается. Хранилища данных временных рядов также обрабатывают данные, полученные вне очереди или несвоевременно, автоматически индексируют точки данных и оптимизируют запросы, полученные в течение определенного промежутка времени. Эта возможность позволяет быстро выполнять запросы к миллионам точек данных и нескольким потокам данных, что, в свою очередь, обеспечивает поддержку визуализации временных рядов (стандартный способ потребления данных временных рядов).

Хранилища данных объектов

Хранилища данных объектов оптимизированы для хранения и извлечения больших двоичных объектов, например, изображений, текстовых файлов, видео- и аудиопотоков, объектов данных и документов приложений большого размера, образы дисков виртуальных машин (рис. 2.6). Объект состоит из сохраненных данных, метаданных и уникального идентификатора доступа к объекту. Хранилища объектов поддерживают отдельные большие файлы, а также позволяют управлять всеми файлами за счет внушительного общего объема хранилища.

path	blob	metadata
/delays/2017/06/01/flights.cscv	0XAABBCCDDEEF...	{created: 2017-06-02}
/delays/2017/06/02/flights.cscv	0XAADDCCDDEEF...	{created: 2017-06-03}
/delays/2017/06/03/flights.cscv	0XAEBBDEDDEEF...	{created: 2017-06-03}

Рис. 2.6. Пример хранилища данных объектов

Некоторые хранилища данных объектов реплицируют определенный большой двоичный объект между несколькими узлами кластера, что обеспечивает быстрое параллельное чтение. Это, в свою очередь, позволяет реализовать масштабируемую архитектуру запроса данных, хранящихся в больших файлах, так как несколько процессов, обычно выполняющихся на разных серверах, могут одновременно запрашивать большие файлы данных.

Часто хранилища данных объектов используют как сетевые общие папки. Доступ к файлам, хранящимся в этих папках, можно получить через компьютерную сеть с использованием стандартных сетевых протоколов, например, *SMB*. При наличии надлежащих механизмов контроля безопасности и одновременного контроля доступа обмен данными таким образом может позволить распределенным службам обеспечить высокомасштабируемый доступ к данным для базовых операций низкого уровня, таких как простое чтение и запись запросов.

Хранилища данных внешних индексов

Хранилища данных внешних индексов позволяют искать информацию, содержащуюся в других хранилищах и службах. Внешний индекс выступает в роли вторичного индекса любого хранилища данных. Кроме того, с его помощью можно индексировать большие объемы данных и предоставлять доступ к этим индексам почти в реальном времени.

Например, в файловой системе могут храниться текстовые файлы. По пути файл можно найти быстро, но поиск на основе содержимого выполняется медленно, так как сканируются все файлы. Внешний индекс позволяет создавать вторичные индексы, а затем быстро

искать путь к файлам, соответствующим заданным условиям. Рассмотрим еще один пример использования внешнего индекса. Предположим, что хранилища пар «ключ–значение» поддерживают индексирование только по ключу (рис. 2.7). Вы можете создать вторичный индекс на основе значений данных и быстро найти ключ, определяющий каждый соответствующий элемент.

id	search-document
233358	{"name": "Pacific Crest National Scenic Trail", "county": "San Diego", "elevation":1294, "location": {"type": "Point", "coordinates": [–120.802102,49.00021]}}
801970	{"name": "Lewis and Clark National Historic Trail", "county": "Richland", "elevation":584, "location": {"type": "Point", "coordinates": [–104.8546903,48.1264084]}}
1144102	{"name": "Intake Trail", "county": "Umatilla", "elevation":1076, "location": {"type": "Point", "coordinates": [–118.0468873,45.9981939]}}

Рис. 2.7. Пример хранилища внешних индексов

Индексы создаются в процессе индексирования, который может выполняться по модели извлечения, то есть по требованию хранилища данных или по модели передачи, то есть по команде из кода приложения. В некоторых системах поддерживаются многомерные индексы и полнотекстовый поиск по большим объемам текстовых данных.

Внешние индексные хранилища данных часто используются для поддержки полного поиска текста и веб-страниц. В этих случаях поддерживается точный или нечеткий поиск. Нечеткий поиск находит документы, которые соответствуют набору условий, и вычисляет для них коэффициент совпадения с этим набором. Некоторые внешние индексы также поддерживают лингвистический анализ, который возвращает соответствия с учетом синонимов, категорий (например, при поиске по запросу «собаки» соответствием считается «питомцы») и морфологии (например, при поиске по запросу «бег» соответствием считается «бегущий»).

Стандартные требования

Часто архитектура нереляционных хранилищ данных отличается от архитектуры реляционных баз данных. В частности, они отличаются отсутствием фиксированной схемы, а также не поддерживают транзакции или ограничивают их область. Из соображений масштабируемости они обычно не включают вторичные индексы.

В Приложении 1 (табл. П.1.1 и П.1.2) приведено сравнение требований каждого нереляционного хранилища данных по одинаковому набору параметров.

Глава 3. СИСТЕМА УПРАВЛЕНИЯ БАЗАМИ ДАННЫХ *MongoDB*

3.1. Общая характеристика *MongoDB* и начало работы

MongoDB (от *humongous*) – кроссплатформенная документо-ориентированная СУБД.

Классифицированная как база данных *NoSQL*, *MongoDB* отходит от традиционных основ реляционной структуры базы данных в пользу *JSON*-подобных документов с динамическими схемами (*MongoDB* называет этот формат *BSON*), что делает интеграцию данных в определенных видах приложений проще и быстрее. Программное обеспечение этой СУБД выпущено под комбинацией *GNU Affero General Public License* и лицензией *Apache*. Таким образом, *MongoDB* является бесплатным программным обеспечением с открытым исходным кодом.

MongoDB реализует новый подход к построению баз данных, где нет таблиц, схем, запросов *SQL*, внешних ключей и многих других вещей, которые присущи объектно-реляционным базам данных.

В отличие от реляционных баз данных *MongoDB* предлагает документо-ориентированную модель данных, благодаря чему *MongoDB* работает быстрее, обладает лучшей масштабируемостью, а также проста в использовании.

Ниже мы рассмотрим основные особенности этой СУБД.

Документоориентированность. Документоориентированная СУБД заменяет концепцию строки более гибкой моделью, документом. Позволяя использовать вложенные документы и массивы, такой подход дает возможность представлять сложные отношения с помощью одной записи. Это позволяет разработчикам, работающим с современными объектно ориентированными языками, при рассматрении своих данных.

Таким образом, если реляционные базы данных хранят строки, то *MongoDB* хранит документы. Документ можно представить как хранилище ключей и значений. Ключ представляет собой простую метку, с которой ассоциирован определенный фрагмент данных – значение.

Каждому ключу соответствует определенное значение. Но здесь также надо учитывать одну особенность: если в реляционных базах

есть четко очерченная структура, где есть поля, и, если какое-то поле не имеет значение, ему можно присвоить значение *NULL*. Если какому-то ключу в *MongoDB* не соотвествует значение, то этот ключ просто опускается в документе и не употребляется.

Если в традиционном мире *SQL* есть таблицы, то в мире *MongoDB* есть коллекции. И если в реляционных базах данных таблицы хранят однотипные, жестко структурированные объекты, то в коллекции могут содержать самые разные объекты, имеющие различную структуру и различный набор свойств.

При всех различиях есть одна особенность, которая сближает *MongoDB* и реляционные базы данных. В реляционных СУБД встречается такое понятие, как первичный ключ. Это понятие описывает некий столбец, который имеет уникальные значения. В *MongoDB* для каждого документа имеется уникальный идентификатор, который называется *_id*. И если не указать его значение, то *MongoDB* автоматически сгенерирует его.

Отсутствие жесткой схемы базы данных и в связи с этим потребности при малейшем изменении концепции хранения данных пересоздавать эту схему значительно облегчают работу с базами данных *MongoDB* и в дальнейшем с их масштабированием. Кроме того, экономится время разработчиков. Им больше не надо думать о пересоздании базы данных и тратить время на построение сложных запросов.

Специальные запросы. *MongoDB* поддерживает поиск по области, поиск запросов по диапазону, поиск регулярного выражения. Запросы могут возвращать определенные поля документов, а также включать в себя пользовательские функции *JavaScript*.

Индексация. Любые поля в документе *MongoDB* могут быть проиндексированы. Вторичные индексы также доступны. Индексы в *MongoDB* почти идентичны таковым в реляционных базах данных.

Репликации. Система хранения данных в *MongoDB* представляет набор реплик, который состоит из двух или более копий данных. В этом наборе есть основной узел, а также может быть набор вторичных узлов. Все вторичные узлы сохраняют целостность и автоматически обновляются вместе с обновлением главного узла. И если основной узел по каким-то причинам выходит из строя, то один из вторичных узлов становится главным.

Балансировка нагрузки. Система масштабируется горизонтально, используя технику сегментирования (англ. *sharding*) объектов баз данных – распределение их частей по различным узлам кластера.

Администратор выбирает ключ сегментирования, который определяет, по какому критерию данные будут разнесены по узлам (в зависимости от значений хэша ключа сегментирования). Благодаря тому, что каждый узел кластера может принимать запросы, обеспечивается балансировка нагрузки.

Система хранения файлов. *MongoDB* может использоваться в качестве файловой системы, называемой *GridFS*, с функциями балансировки нагрузки и репликации данных на нескольких машинах для хранения файлов.

Одной из проблем при работе с любыми системами баз данных является сохранение данных большого размера. Можно сохранять их данные в файлах, используя различные языки программирования. Некоторые СУБД предлагают специальные типы данных для хранения бинарных данных в базе данных (например, *BLOB* в *MySQL*). В отличие от реляционных СУБД *MongoDB* позволяет сохранять различные документы с разным набором данных, однако при этом размер документа ограничивается в 16 мегабайт. Но *MongoDB* предлагает специальную технологию, функцию *Grid File System* (*GridFS*), которая позволяет хранить данные по размеру больше 16 мегабайт.

Система *GridFS* состоит из двух коллекций. В первой коллекции, которая называется *files*, хранятся имена файлов, а также их метаданные, например, размер. В другой коллекции, которая называется *chunks*, в виде небольших сегментов хранятся данные файлов, обычно сегментами по 256 килобайт.

Для тестирования *GridFS* можно использовать специальную утилиту *mongofiles*, которая идет в пакете *mongodb*.

Агрегирование. *MongoDB* предоставляет три способа выполнения агрегации: конвейер агрегации, функция *MapReduce* и универсальные методы агрегации.

MapReduce может использоваться для пакетной обработки данных и операций агрегации. Но согласно документации *MongoDB*, конвейер агрегации обеспечивает лучшую производительность для большинства операций агрегации.

Инфраструктура агрегации позволяет пользователям получать результаты, для которых используется предложение *SQL GROUP BY*. Операторы агрегации могут быть соединены вместе для формирования конвейера – аналогично каналам *Unix*. Структура агрегации включает в себя оператор поиска *$lookup*, который может объединять документы из нескольких коллекций, а также объединять и статистические операторы, такие как стандартное отклонение.

Использование *JavaScript* на стороне сервера. *JavaScript* можно использовать в запросах, функциях агрегирования (например, *MapReduce*) и отправлять непосредственно в базу данных для выполнения.

Ограниченные коллекции. *MongoDB* поддерживает коллекции фиксированного размера, называемые ограниченными коллекциями. Этот тип коллекции поддерживает порядок вставки и после достижения указанного размера ведет себя как круговая очередь.

Транзакции. *MongoDB* поддерживает многодокументные транзакции с момента выпуска 4.0 в июне 2018 года.

У *MongoDB* есть официальные драйверы для основных языков программирования (*C*, *C++*, *C#*, *Erlang*, *Golang*, *Haskell*, *J#*, *Java*, *JavaScript*, *Lisp*, *Perl*, *PHP*, *Python*, *Ruby*, *Delphi*, *Scala*). Существует также большое количество неофициальных или поддерживаемых сообществом драйверов для других языков программирования и фреймворков.

Основным интерфейсом к базе данных была командная оболочка *mongo*. С версии *MongoDB* 3.2 в качестве графической оболочки поставляется *MongoDB Compass*. Существуют продукты и сторонние проекты, которые предлагают инструменты с *GUI* для администрирования и просмотра данных.

Считается, что базы данных *MongoDB* подходят для следующих применений: хранение и регистрация событий; системы управления документами и контентом; электронная коммерция; игры; данные мониторинга, датчиков; мобильные приложения; хранилище операционных данных веб-страниц (например, хранение комментариев, рейтингов, профилей пользователей, сеансы пользователей).

Для установки *MongoDB* необходимо загрузить один распространяемых пакетов с официального сайта *https://www.mongodb.com/download-center/community*. Этот сайт предоставляет пакеты дистрибутивов для различных платформ: *Windows*, *Linux*, *MacOS*, *Solaris*. Для каждой платформы доступно несколько дистрибутивов. Следует отметить, что есть два вида серверов – *Community* и *Enterprise*. В этом учебно-методическом пособии мы будем использовать версию *Community*.

На момент написания данной книги последней была версия 4.2.8. Использование конкретной версии может несколько отличаться от применения иных версий платформы *MongoDB*.

Для загрузки необходимого программного обеспечения сервера нужно выбрать нужную операционную систему и подходящий тип

пакета. Например, для операционной системы *Windows* можно выбрать тип пакета *ZIP*, то есть загрузить сервер в виде архива.

Следует отметить, что если до установки уже была установлена более ранняя версия *MongoDB*, то ее необходимо удалить, а также удалить все ранее созданные базы данных.

После загрузки архивного пакета распакуем его в папку *C*:*mongodb*.

Если после установки мы откроем папку *C*:*mongodb**bin*, то сможем найти там различные приложения. Рассмотрим их назначение:

- *bsondump*: считывает содержимое *BSON*-файлов и преобразует их в читабельный формат, например, в *JSON*;
- *mongo*: консольный клиент, который предоставляет интерфейс для взаимодействия с СУБД и ее базами данных;
- *mongod*: собственно сервер баз данных *MongoDB*, который обрабатывает запросы, управляет форматом данных и выполняет различные операции по управлению базами данных в фоновом режиме;
- *mongodump*: утилита создания бэкапа баз данных;
- *mongoexport*: утилита для экспорта данных в форматы *JSON*, *TSV* или *CSV*;
- *mongofiles*: утилита, позволяющая управлять файлами в системе *GridFS*;
- *mongoimport*: утилита, импортирующая данные в форматах *JSON*, *TSV* или *CSV* в базу данных *MongoDB*;
- *mongorestore*: позволяет записывать данные из дампа, созданного *mongodump*, в новую или существующую базу данных;
- *mongos*: служба маршрутизации *MongoDB*, которая помогает обрабатывать запросы и определять местоположение данных в кластере *MongoDB*;
- *mongstat*: представляет счетчики операций с базой данных;
- *mongotop*: предоставляет способ подсчета времени, затраченного на операции чтения–записи в базе данных.

После установки надо создать на жестком диске каталог, в котором будут находиться базы данных *MongoDB*.

В ОС *Windows* по умолчанию *MongoDB* хранит базы данных по пути *C:\data\db*, поэтому, если вы используете *Windows*, вам нужно создать соответствующий каталог. В ОС *Linux* и *MacOS* каталогом по умолчанию будет /*data*/*db*.

Если же возникла необходимость использовать какой-то другой путь к файлам, то его можно передать при запуске *MongoDB* во флаге *--dbpath*.

Итак, после создания каталога для хранения базы данных можно запустить сервер *MongoDB*. Сервер представляет приложение *mongod*, которое находится в папке *bin*. Для этого запустим командную строку (в *Windows*) или консоль (в *Linux*) и там введем соответствующие команды (рис. 3.1).

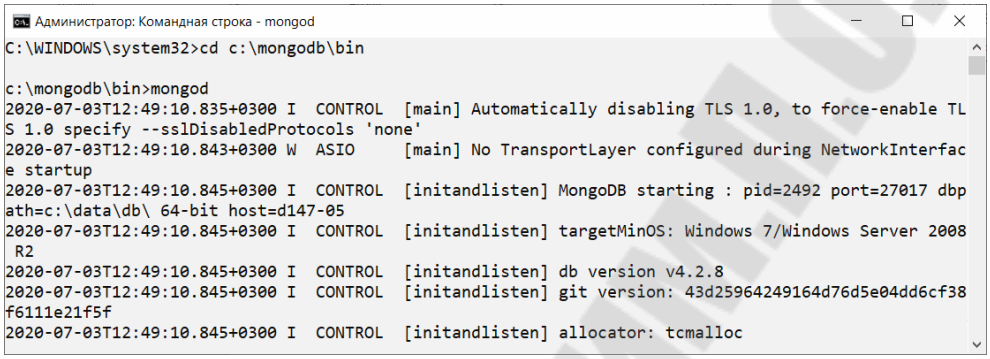

Рис. 3.1. Пример ввода командной строки

Командная строка отобразит нам ряд служебной информации, например, что сервер запускается на *localhost* на порту 27017.

После удачного запуска сервера мы сможем производить операции с СУБД через оболочку *mongo*. Эта оболочка представляет файл *mongo.exe*, который располагается в вышерассмотренной папке установки. Запустим этот файл (рис. 3.2).

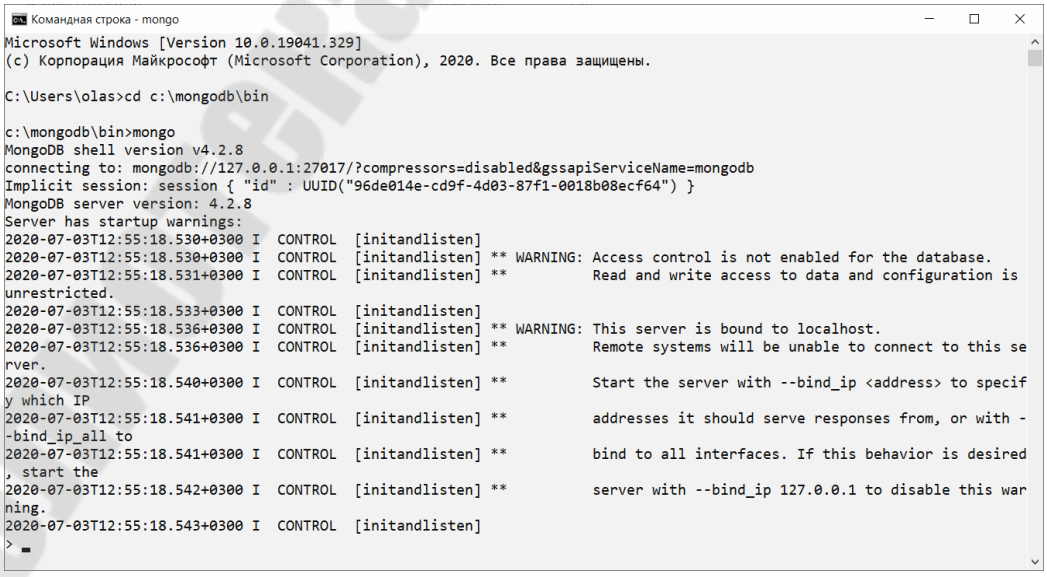

Рис. 3.2. Пример запуска оболочки *mongo*

Это консольная оболочка для взаимодействия с сервером, через которую можно управлять данными. Второй строкой эта оболочка говорит о подключении к серверу *mongod*.

Произведем какие-либо простейшие действия. Для этого введем в *mongo* последовательно следующие команды и после каждой команды нажмем на *Enter*:

```
use test
db.users.save( { name: "Tom" } )
db.users.find()
```

Первая команда *use test* устанавливает в качестве используемой базу данных *test*. Даже если такой базы данных нет, то она создается автоматически. И далее *db* будет представлять текущую базу данных, то есть базу данных *test*. После *db* идет *users* – это коллекция, в которую затем мы добавляем новый объект. Если в *SQL* нам надо создавать таблицы заранее, то коллекция *MongoDB* создает самостоятельно при их отсутствии.

С помощью метода *db.users.save*() в коллекцию *users* базы данных *test* добавляется объект { name: "*Tom*" }. Описание добавляемого объекта определяется в формате, с которым вы, возможно, знакомы, если имели дело с форматом *JSON*. В данном случае у объекта определен один ключ *name*, которому сопоставляется значение *Tom*. То есть мы добавляем пользователя с именем *Tom*.

Если объект был успешно добавлен, то консоль выведет результат в виде выражения *WriteResult*({ "*nInserted*" : 1 }).

А третья команда *db.users.find*() выводит на экран все объекты из базы данных *test* (рис. 3.3).

Рис. 3.3. Пример вывода на экран объектов из базы данных *test*

Из вывода вы можете увидеть, что к начальным значениям объекта было добавлено некоторое поле *ObjectId*. *MongoDB* в качестве уникальных идентификаторов документа использует поле *_id*, и в данном случае *ObjectId* как раз представляет значение для идентификатора *_id*.

В этом учебно-методическом пособии мы часто будем демонстрировать возможности СУБД, используя консоль *mongo*, добавляя и отображая объекты.

Для работы с *MongoDB* из приложений, написанных на конкретных языках программирования, потребуются специальные драйверы. На странице *https://docs.mongodb.com/ecosystem/drivers/* можно найти драйвера для таких языков программирования, как *C*, *C++*, *C#*, *Go*, *Java*, *JavaScript*, *Perl*, *PHP*, *Python*, *Ruby*, *Rust*, *Scala*, *Swift* и др.

3.2. Структурные единицы: базы данных, коллекции, документы, представления

Основными структурными единицами СУБД *MongoDB* являются одна или несколько баз данных, а также коллекции, документы и представления, расположенные в конкретной базе данных.

Ниже мы обсудим их назначение и способы манипуляции ими.

Если в реляционных базах данных содержимое составляют таблицы, то в *mongodb* база данных состоит из коллекций.

Модель устройства базы данных в *MongoDB* представлена на рис. 3.4.

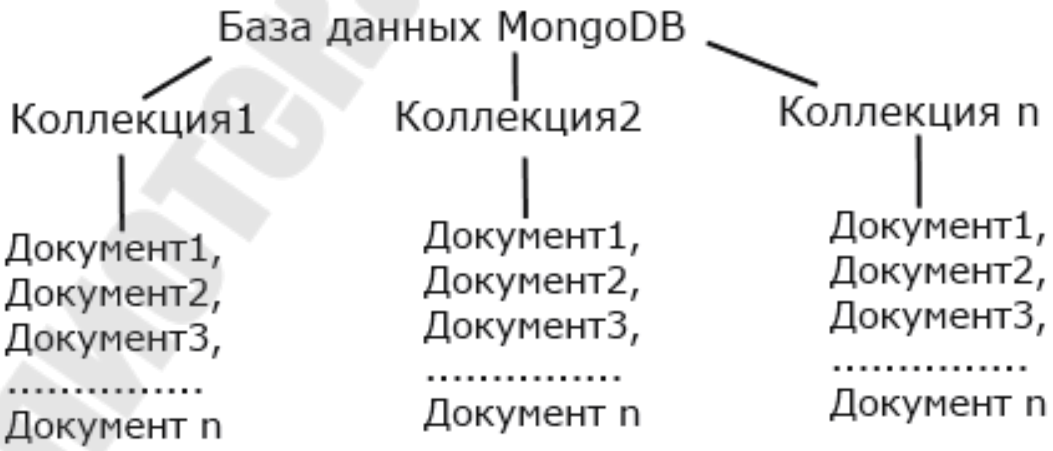

Рис. 3.4. Модель устройства базы данных в *MongoDB*

Таким образом, ***база данных*** представляет собой совокупность коллекций. Следует отметить, что в *MongoDB* применяется организация данных, не требующая описания схемы базы данных, предназначенной, чтобы связывать документы одной коллекции с документами, содержащими данные, другой коллекции через ключевые поля.

Чтобы выбрать для дальнейшего использования конкретную базу данных из всей совокупности баз данных, находящихся под управлением выбранного сервера СУБД, в оболочке *mongo* следует ввести оператор *use <db>*, как в следующем виде: *use myDB*.

Если база данных *myDB* не существует, *MongoDB* создает базу данных при первом сохранении данных для этой базы данных. Таким образом, вы можете переключиться на несуществующую базу данных и выполнить следующую операцию в оболочке *mongo*:

```
use myNewDB
db.myNewCollection1.insertOne( { x: 1 } )
```

Операция *insertOne*() создает базу данных *myNewDB* и коллекцию *myNewCollection*1, если они еще не существуют. Убедитесь, что имена базы данных и коллекции соответствуют ограничениям именования *MongoDB*.

MongoDB хранит документы в коллекциях. Коллекции аналогичны таблицам в реляционных базах данных (рис. 3.5).

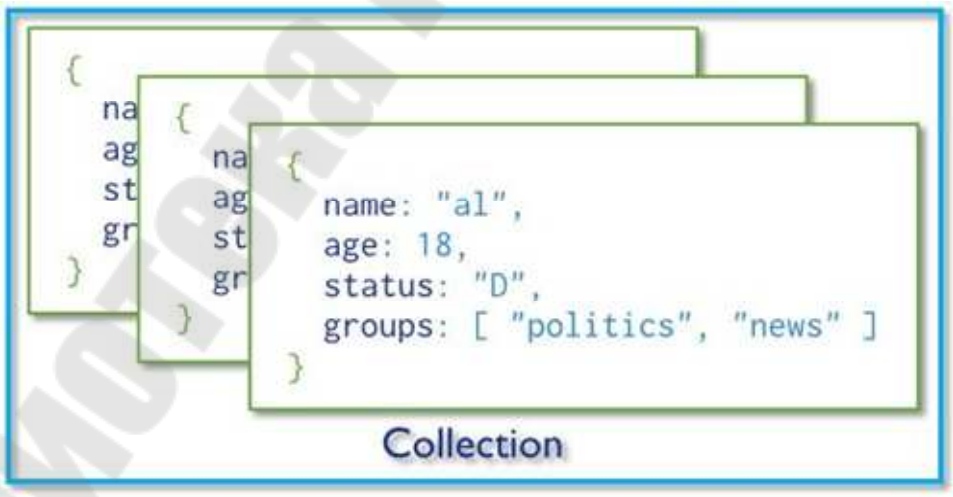

Рис. 3.5. Модель хранения документов в коллекциях

Коллекция представляет собой совокупность документов однотипной структуры.

Если коллекция не существует, *MongoDB* создает коллекцию при первом сохранении данных для нее:

*db.myNewCollection*2.*insertOne*({ x: 1 })
*db.myNewCollection*3.*createIndex*({ y: 1 })

Операции *insertOne*() и *createIndex*() неявно создают свою соответствующую коллекцию, если они еще не существуют.

MongoDB предоставляет метод *db.createCollection*() для явного создания коллекции с различными параметрами, такими как установка максимального размера или правила проверки документации.

Каждая коллекция имеет свое уникальное имя *UUID* (*universally unique identifier*) – произвольный идентификатор, состоящий из не более чем 128 различных алфавитно-цифровых символов и знака подчеркивания. *UUID* коллекции остается одинаковым для всех членов набора реплик и сегментов в сегментированном кластере.

Чтобы получить значение *UUID* для коллекции, следует выполнить команду *listCollections* или метод *db.getCollectionInfos*(). Эти методы возвращают массив документов, содержащих информацию о коллекции.

В процессе работы, возможно, потребуется изменить название коллекции. Например, если при первом добавлении данных в ее названии была опечатка. И чтобы не удалять и затем пересоздавать коллекцию, следует использовать функцию *renameCollection*:

db.users.renameCollection ("новое_название")

И если переименование пройдет удачно, то консоль отобразит:

{"*ok*" : 1}

Когда мы отправляем запрос к базе данных на выборку, то *MongoDB* возвращает нам документы в том порядке, как правило, в котором они были добавлены. Однако такой порядок не всегда гарантируется, так как данные могут быть удалены, перемещены, изменены. Поэтому в *MongoDB* существует понятие ограниченной коллекции.

***Ограниченная коллекция* (*capped collection*)** – коллекция, имеющая фиксированный размер и гарантирующая расположение документов в том же порядке, в котором они добавлялись. Когда в кол-

лекции уже нет места, наиболее старые документы удаляются, и в конец добавляются новые данные.

В отличие от обычных коллекций, ограниченные мы можем задать явным образом. Например, создадим ограниченную коллекцию с названием *profile* и зададим для нее размер в 9500 байт:

db.createCollection("*profile*", {*capped*:*true*, *size*:9500})

И после удачного создания коллекции консоль выведет:

{"*ok*":1}

Можно также ограничить количество документов в коллекции, указав его в параметре *max*:

db.createCollection("*profile*", {*capped*:*true*, *size*:9500, max: 150})

Однако при таком способе создания коллекции следует учитывать, что если все место под коллекцию заполнено (например, выделенные нами 9500 байтов), а количество документов еще не достигло максимума, в данном случае 150, то в этом случае при добавлении нового документа самый старый документ будет удаляться, а на его место будет вставляться новый документ.

При обновлении документов в таких коллекциях надо помнить, что документы не должны расти в размерах, иначе обновление не удастся произвести.

Нельзя также удалять документы из подобных коллекций, можно только удалить всю коллекцию.

В отличие от реляционных баз данных *MongoDB* не использует табличное устройство с четко заданным количеством столбцов и типов данных. *MongoDB* является документоориентированной системой, в которой центральным понятием является документ.

Документ – объект заданной структуры, хранящий некоторую информацию. *MongoDB* хранит записи данных как документы *BSON*. *BSON* – это двоичное представление документов *JSON*, которое содержит больше типов данных, чем *JSON*. В некотором смысле документ подобен строкам в реляционных СУБД, где строки хранят информацию об отдельном элементе. Например, типичный документ имеет вид:

```
{
   "name": "Bill",
   "surname": "Gates",
   "age": "48",
   "company":
     {
      "name" : "microsoft",
      "year" : "1974",
      "price" : "300000"
      }
}
```

В общем случае документы *MongoDB* состоят из пар «поле–значение» и имеют следующую структуру:

```
{
  field1: value1,
  field2: value2,
  field3: value3,
  ...
  fieldN: valueN
}
```

Значением поля может быть любой из типов данных *BSON*, включая другие документы, массивы и массивы документов. Например, следующий документ содержит значения разных типов:

```
var mydoc = {
            _id: ObjectId("5099803df3f4948bd2f98391"),
            name: { first: "Alan", last: "Turing" },
            birth: new Date('Jun 23, 1912'),
            death: new Date('Jun 07, 1954'),
            contribs: [ "Turing machine", "Turing test", "Turingery" ],
            views : NumberLong(1250000)
          }
```

Вышеуказанные поля имеют следующие типы данных:

- *_id* содержит *ObjectId*;
- *Name* содержит внедренный документ, который содержит поля *first* и *last*;
- рождение и смерть содержат значения типа *Date*;

• *Contribs* содержит массив строк;

• *View* содержит значение типа *NumberLong*.

Имена полей документов являются строками. Документы имеют следующие ограничения на имена полей:

• имя поля *_id* зарезервировано для использования в качестве первичного ключа; его значение должно быть уникальным в коллекции, неизменным; может быть любого типа, кроме массива;

• имена полей не могут содержать нулевой символ;

• имена полей верхнего уровня не могут начинаться со знака доллара ($).

Некоторые документы, созданные внутренними процессами *MongoDB*, могут иметь дубликаты полей, но ни один процесс никогда не добавит дубликаты полей в существующий пользовательский документ.

MongoDB использует точечную нотацию для доступа к элементам массива и для доступа к полям внедренного документа.

Чтобы указать или получить доступ к элементу массива по позиции индекса, начинающегося с нуля, объедините имя массива с точкой индекса (.) и позицией, начинающейся с нуля. В этом случае необходимо использовать кавычки:

"*<array>.<index>*"

Например, в виде массива из трех элементов задано следующее поле в документе:

```
{
  ...
  contribs: [ "Turing machine", "Turing test", "Turingery" ],
  ...
}
```

Чтобы указать третий элемент в массиве *contribs*, следует использовать точечную запись вида *contribs*.2.

Чтобы указать или получить доступ к полю внедренного документа с точечной нотацией, объедините имя внедренного документа с точкой (.) и именем поля и заключите в кавычки:

"*<embedded document>.<field>*"

Например, в виде внедренного документа задано следующее поле:

```
{
  ...
  name: { first: "Alan", last: "Turing" },
  contact: { phone: { type: "cell", number: "111-222-3333" } },
  ...
}
```

Чтобы получить доступ к полю с именем *last*, в поле имени можно использовать точечную запись вида *name.last*, а чтобы указать номер в документе телефона в поле контакта – точечную запись вида *contact.phone.number*.

Документы имеют следующие атрибуты: размер, порядок полей и поле *_id*. Приведем их более подробное описание.

Максимальный размер документа составляет 16 мегабайт. Максимальный размер документа гарантирует, что один документ не может использовать чрезмерное количество ОЗУ или во время передачи чрезмерное количество полос пропускания. Для хранения документов, размер которых превышает максимальный, *MongoDB* предоставляет *GridFS API* в соответствии с возможностями драйвера для конкретного языка программирования.

Порядок полей документа сохраняется неизменным после операций записи, за исключением следующих случаев:

- поле *_id* всегда является первым полем в документе;
- обновления, которые включают переименование имен полей, могут привести к переупорядочению полей в документе.

Поле _id – обязательное уникальное поле, которое действует как первичный ключ. Если во вставленном документе пропущено поле *_id*, драйвер *MongoDB* автоматически создает *ObjectId* для поля *_id*. Это также относится к документам, вставленным через операции обновления.

Поле *_id* имеет следующее поведение и ограничения:

- по умолчанию *MongoDB* создает уникальный индекс в поле *_id* при создании коллекции;
- поле *_id* всегда является первым полем в документах. Если сервер получает документ, у которого сначала нет поля *_id*, то сервер переместит поле в начало;
- поле *_id* может содержать значения любого типа данных *BSON*, кроме массива.

Ниже приведены общие рекомендации для выбора значений для

поля *_id*:

• используйте *ObjectId*;

• используйте естественный уникальный идентификатор, если он доступен, это сэкономит место и позволит избежать дополнительного индекса;

• генерируйте автоинкрементное число;

• создайте *UUID* в коде приложения. Для более эффективного хранения значений *UUID* в коллекции и в индексе *_id* сохраните *UUID* в качестве значения типа *BINData BSON*.

***Представление* (*View*)** – это запрашиваемый объект, содержимое которого определяется результатами агрегации в других коллекциях или представлениях.

MongoDB не сохраняет результаты выполнения представления на диск. Содержимое представления вычисляется по требованию, когда клиент запрашивает представление. Таким образом, операции записи для представлений не поддерживаются, то есть представления – это объекты только для чтения.

В общем случае, как и у других СУБД, *MongoDB* может требовать, чтобы клиенты имели разрешение запрашивать представление.

Примерами представлений являются:

• представление для сбора данных о сотрудниках, чтобы исключить любую личную информацию;

• представление для сбора собранных данных датчика, чтобы добавить вычисляемые поля и метрики;

• представление, объединяющее две коллекции, содержащие данные о запасах и истории заказов, соответственно.

Когда клиенты запрашивают представление, *MongoDB* добавляет клиентский запрос в базовый конвейер и возвращает клиенту результаты этого комбинированного конвейера. *MongoDB* может применять агрегационные конвейерные оптимизации к комбинированному конвейеру.

Чтобы создать или определить представление, можно использовать разные подходы.

Использование метода *db.createCollection*():

```
db.createCollection(
  "<viewName>",
  {
    "viewOn" : "<source>",
```

```
   "pipeline" : [<pipeline>],
   "collation" : { <collation> }
  }
)
```

Использование метода *db.createView*():

```
db.createView(
  "<viewName>",
  "<source>",
  [<pipeline>],
  {
    "collation" : { <collation> }
  }
)
```

Следует отметить, что представления должны создаваться в той же базе данных, что и исходная коллекция.

Динамическая схема

В общем случае коллекции имеют динамические схемы. Это означает, что документы в одной коллекции, в свою очередь, могут иметь разную схему (структуру). Например, оба приведенных ниже документа могут храниться в одной коллекции:

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

Это фактически приводит к тому, что тот или иной документ можно поместить в любую коллекцию. В связи с этим являются закономерными вопросы: зачем вообще нужны отдельные коллекции, если нет необходимости в одинаковых схемах для разных видов документов, зачем использовать дополнительные коллекции?

Можно использовать одну динамическую коллекцию в базе данных, однако так поступать нецелесообразно по целому ряду причин, связанных с использованием данных коллекций для разработки графического интерфейса приложения базы данных, ее администрирования и скорости выполнения типовых операций [9].

Хранение разных видов документов в одной коллекции может стать существенной проблемой для разработчиков и администрато-

ров. 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 проще и 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 сложнее искать эти три значения в одной коллекции, чем иметь три отдельные коллекции и запрашивать правильную.

Группировка документов одного и того же вида в одной коллекции допускает локальность хранения данных на диске. 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.

При создании коллекции мы начинаем навязывать ее документам некую структуру при создании индексов. Это особенно уместно в случае с уникальными индексами. Помещая в одну коллекцию только документы одного типа, можно более эффективно индексировать свои коллекции.

3.3. Типы данных. *BSON*

***JSON* (*JavaScript Object Notation*)** – текстовый формат обмена данными, основанный на *JavaScript*. Несмотря на происхождение от *JavaScript*, формат является независимым от языка и может использоваться практически с любым языком программирования. Для многих языков программирования существует готовый код для создания и обработки данных в формате *JSON*.

За счет своей лаконичности по сравнению с *XML* формат *JSON* может быть более подходящим для сериализации сложных структур. Применяется в веб-приложениях как для обмена данными между браузером и сервером (*AJAX*), так и для обмена между серверами.

Поскольку формат *JSON* является подмножеством синтаксиса языка *JavaScript*, то он может быть десериализован встроенной функцией *eval*().

JSON-текст представляет собой (в закодированном виде) одну из двух структур:

1) набор пар «ключ–значение». В различных языках это реализовано как запись, структура, словарь, хеш-таблица, список с ключом или ассоциативный массив. Ключом может быть только регистрозависимая строка, значением – любая форма;

2) упорядоченный набор значений. Во многих языках это реализовано как массив, вектор, список или последовательность.

Структуры данных, используемые *JSON*, поддерживаются любым современным языком программирования, что позволяет применять *JSON* для обмена данными между различными языками программирования и программными системами.

В качестве значений в *JSON* могут быть использованы:

• запись – это неупорядоченное множество пар «ключ–значение», заключенное в фигурные скобки «{ }». Ключ описывается строкой, между ним и значением стоит символ «:». Пары «ключ–значение» отделяются друг от друга запятыми;

• массив (одномерный) – это упорядоченное множество значений. Массив заключается в квадратные скобки «[ ]». Значения разделяются запятыми. Массив может быть пустым, то есть не содержать ни одного значения;

• число (целое или вещественное);

• литералы *true* (логическое значение «истина»), *false* (логическое значение «ложь») и *null*;

• строка – это упорядоченное множество из нуля или более символов юникода, заключенное в двойные кавычки. Символы могут быть указаны с использованием *escape*-последовательностей, начинающихся с обратной косой черты «\» (поддерживаются варианты \', \", \\, \/, *t*, *n*, *r*, *f* и *b*), или могут быть записаны шестнадцатеричным кодом в кодировке *Unicode* в виде *uFFFF*.

Строка очень похожа на литерал одноименного типа данных в языке *JavaScript*. Число также очень похоже на *JavaScript*-число, за исключением того, что в этом случае используется только десятичный формат (с точкой в качестве разделителя). Пробелы могут быть вставлены между любыми двумя синтаксическими элементами.

Следующий пример показывает *JSON*-представление данных об объекте, описывающем конкретного человека. В данных присутствуют строковые поля имени и фамилии, информация об адресе и массив, содержащий список телефонов. Как видно из примера, значение может представлять собой вложенную структуру.

```
{
  "firstName": "Иван",
  "lastName": "Воробей",
  "address": {
     "streetAddress": "Речицкое ш., 101, кв.101",
     "city": "Гомель",
     "postalCode": 101101
  },
  "phoneNumbers": [
     "212 123-1234",
     "212 123-4567"
  ]
}
```

Следует обратить внимание на пару "*postalCode*": 101101. В качестве значений в *JSON* могут быть использованы как число, так и строка. Поэтому значение "*postalCode*" в виде "101101" является строкой, а "*postalCode*": 101101 – это уже числовое значение.

Однако есть несколько проблем, которые делают *JSON* не совсем подходящим форматом для использования внутри базы данных:

- *JSON* – это текстовый формат, а анализ текста происходит относительно медленно;
- читаемый формат *JSON* далек от оптимальной модели экономичного хранения с точки зрения занимаемого места в базе данных;
- *JSON* поддерживает только ограниченное количество основных типов данных.

Чтобы использовать формат *JSON* в *MongoDB* сделать его высокопроизводительным и универсальным, разработчиками был использован родственный формат *BSON* для преодоления разрыва. *BSON* использует двоичное представление для хранения данных в формате *JSON*, оптимизированное по скорости, занимаемому месту и гибкости.

BSON, сокращение от ***Binary JSON***, представляет собой двоичную кодировку сериализации документов. Подобно *JSON*, *BSON* поддерживает *JSON*-подобных встраивание документов и массивов в другие документы и массивы. Этот формат также содержит расширения, которые позволяют представлять типы данных, не являющимися частью спецификации *JSON*. Например, *BSON* имеет тип *Date* и тип *BinData*.

Официальная спецификация *BSON* находится по адресу *bsonspec.org*.

Каждый тип *BSON* имеет как целочисленные, так и строковые идентификаторы, как указано в табл. 3.1.

Таблица 3.1

Таблица идентификаторов типа *BSON*

Тип	Номер	Псевдоним	Описание
Double	1	*double*	Числовой тип данных для хранения чисел с плавающей точкой
String	2	*string*	Строковый тип данных
Object	3	*object*	Строковый тип данных, используется для хранения внедренного документа
Array	4	*array*	Тип данных для хранения массивов элементов
Binary data	5	*binData*	Тип для хранения данных в бинарном формате
Undefined	6	*undefined*	Устаревший
ObjectId	7	*objectId*	Тип данных для хранения *_id* документа
Boolean	8	*bool*	Логический тип данных, хранящий логические значения *TRUE* или *FALSE*, например, {*"married": FALSE*}
Date	9	*date*	Хранит дату в формате времени *Unix*
Null	10	*null*	Тип данных для хранения значения *Null*
Regular Expression	11	*regex*	Применяется для хранения регулярных выражений
DBPointer	12	*dbPointer*	Устаревший
JavaScript	13	*javascript*	Тип данных для хранения кода *javascript*
Symbol	14	*symbol*	Устаревший. Тип данных, идентичный строковому. Используется преимущественно для тех языков, в которых есть специальные символы
JavaScript (with scope)	15	*javascriptWithScope*	Тип данных позволяет нам хранить данные *JavaScript* с областью действия

Окончание табл. 3.1

Тип	Номер	Псевдоним	Описание
32-bit integer	16	*int*	Используется для хранения целочисленных значений, например, {*"age": 29*}
Timestamp	17	*timestamp*	Применяется для хранения времени
64-bit integer	18	*long*	Используется для хранения длинных целочисленных значений, например, {*"age": 29*}
Decimal128	19	*decimal*	Начиная с версии 3.4, 128-битное десятичное представление для хранения действительно больших (или действительно маленьких) чисел, когда важно точно округлить десятичные дроби
Min key	-1	*minKey*	Используются для сравнения значений с наименьшим элементов *BSON*
Max key	127	*maxKey*	Используются для сравнения значений с наибольшим элементов *BSON*

Можно использовать эти значения с оператором $*type* для запроса документов по их типу *BSON*. Оператор агрегации $*type* возвращает тип выражения оператора, используя одну из перечисленных строк типа *BSON*.

Ниже приведем описание специфики некоторых типов *BSON*.

ObjectId – уникальные, быстро генерируемые и упорядоченные значения. Значения полей типа *ObjectId* имеют длину 12 байтов и состоят из нескольких сегментов:

- 4-байтового значения, представляющего секунды с начала эпохи *Unix*;
- 5-байтового случайного числа, уникального для машины и процесса;
- 3-байтового счетчика, начинающегося со случайного значения.

Таким образом, первые 9 байт гарантируют уникальность среди других машин, на которых могут быть реплики базы данных. А следующие 3 байта гарантируют уникальность в течение одной секунды для одного процесса. Такая модель построения идентификатора гарантирует с высокой долей вероятности, что процесс будет иметь уникальное значение, так как она позволяет создавать до 16 777 216 уникальных объектов *ObjectId* в секунду для одного процесса.

В *MongoDB* каждый документ, хранящийся в коллекции, требует уникального поля *_id*, которое действует как первичный ключ. Если во вставленном документе пропущено поле *_id*, драйвер *MongoDB* автоматически создает *ObjectId* для поля *_id*.

Клиенты *MongoDB* должны добавить поле *_id* с уникальным *ObjectId*. Использование *ObjectIds* для поля *_id* обеспечивает следующие дополнительные преимущества:

- в оболочке *mongo* вы можете получить *доступ к времени создания ObjectId*, используя метод *ObjectId.getTimestamp* ();
- *сортировка по полю _id*, в котором хранятся значения *ObjectId*, примерно эквивалентна сортировке по времени создания.

Следует отметить, что, хотя значения *ObjectId* должны увеличиваться со временем, они не обязательно являются монотонными, так как содержат только одну секунду временного разрешения. Поэтому значения *ObjectId*, созданные за одну и ту же секунду, не имеют гарантированного упорядочения и генерируются клиентами, которые могут иметь разные системные часы.

Строки *BSON* записываются в кодировке *UTF*-8. Обычно драйверы для каждого языка программирования преобразуются из строкового формата языка в *UTF*-8 при сериализации и десериализации *BSON*. Это позволяет с легкостью хранить большинство международных символов в строках *BSON*. Кроме того, запросы регулярных выражений *MongoDB* $*regex* поддерживают *UTF*-8 в строке регулярных выражений.

Timestamp – специальный тип метки времени для внутреннего использования, который прямо не связан с обычным типом *Date*. Этот тип внутренней метки времени является 64-битным значением, где:

- старшие 32 бита – это значение *time_t* (секунды с начала эпохи *Unix*);
- младшие 32 бита представляют собой порядковый номер приращения для операций в течение данной секунды.

В то время как формат *BSON* имеет младший порядок и потому сначала сохраняет младшие значащие биты, экземпляр *mongod* всегда сравнивает значение *time_t* перед порядковым значением на всех платформах, независимо от порядка байтов.

В одном экземпляре *mongod* значения меток времени всегда уникальны.

При вставке документа, содержащего поля верхнего уровня с пустыми значениями временной метки, *MongoDB* заменяет пустые значения

текущим значением временной метки, за исключением следующего. Если само поле *_id* содержит пустое значение метки времени, оно всегда будет вставлено как есть и не будет заменено.

Пусть происходит вставка документ с пустым значением метки времени:

db.test.insertOne({ *ts*: *new Timestamp*() });

Запуск операции поиска *db.test.find*() вернул бы документ, который выглядит следующим образом:

{ "*_id*" : *ObjectId*("542*c*2*b*97*bac*0595474108*b*48"), "*ts*" : *Timestamp*(1412180887, 1) }

Сервер заменил пустое значение временной метки для *ts* значением временной метки во время вставки.

Тип *Timestamp* предназначен для внутреннего использования *MongoDB*. В большинстве случаев при разработке приложений следует использовать тип *Date*.

Date – это 64-разрядное целое число, представляющее количество миллисекунд с начала эпохи *Unix* (1 января 1970 г.). Это приводит к широкому диапазону дат: приблизительно 290 миллионов лет в прошлом и будущем. Официальная спецификация *BSON* ссылается на тип даты *BSON* как дату и время *UTC*. Поля типа *Date* имеют знак. Отрицательные значения представляют даты до 1970 г.

Ниже приведем ряд примеров использования данных этого типа.

Создание переменной типа *Date*, используя конструктор *new Date*()в оболочке *mongo*:

*var mydate*1 = *new Date*()

Создание переменной типа *Date*, используя конструктор *ISODate*() в оболочке *mongo*:

*var mydate*2 = *ISODate*()

Преобразование значение даты в строку:

mydate1.toString()

Получение значения месяца из значения *Date*:

mydate1.getMonth()

Следует отметить, что номера месяцев с отсчитываются от нуля, так что январь – это месяц 0.

Object – тип данных, который используется для хранения внедренного документа. Серия вложенных документов, особенно в формате пары «ключ–значение», называется ***внедренным документом***.

Как видно из приведенного ниже примера, мы вставляем документ с именем *local* в базу данных *test* (команда *db.test.insert*) в качества объекта типа *Object*:

var local = {*author*: "*Digamber*", *score*: 5.5, *publication*: "*positronx*"}
db.test.insert({*system*: "*MacOS*", *diskspace*: "10*GB*", *server*: *local*})

Array – тип данных, позволяющий хранить массив. Причем мы можем хранить несколько значений или список в одном элементе. В следующем примере показано как хранить значения в массиве:

use test
*var array*1 = [*'USA'*, *'France'*, *'UK'*]
*var array*2 = [*'USA'*, *'France'*, *'UK'*, 515, 615, 2.15]
*var array*3 = [*'USA'*, *'France'*, *'UK'*, 515, *new Date*()]
db.test.insert({*data*1: *array*1, *data*2: *array*2, *data*3: *array*3})

3.4. Добавление документов

Операции создания или вставки добавляют новые документы в коллекцию. Если коллекция в момент выполнения операции не существует, ядро СУБД перед операциями операции вставки создаст ее.

MongoDB предоставляет следующие методы для вставки документов в коллекцию:

db.collection.insertOne()
db.collection.insertMany()

Все операции вставки нацелены на одну коллекцию. Все операции записи в *MongoDB* являются атомарными на уровне одного документа.

Операция *InsertOne*() вставляет один документ в выбранную коллекцию. В следующем примере новый документ вставляется в коллекцию *catalogue*. Если в документе не указано поле *_id* оно добавляется со значением *ObjectId* в новый документ:

```
db.catalogue.insertOne(
{
item: "canvas",
qty: 100,
tags: ["cotton"],
size: { h: 28, w: 35.5, uom: "cm" }
})
```

Чтобы получить только что вставленный документ, можно выполнить следующий запрос к коллекции:

```
db.catalogue.find( { item: "canvas" } )
```

Если нужно вставить несколько документов в коллекцию, можно использовать метод *insertMany*(). Этот метод позволяет передавать массив документов в базу данных, что более эффективно, потому что код не будет «бегать» в базу данных и обратно ради каждого документа, а вставит их все сразу. Для того чтобы сделать такую вставку, необходимо передать массив документов методу. В следующем примере вставляются три новых документа в коллекцию *catalogue*. Если в документах не указано поле *_id*, драйвер добавляет поле *_id* со значением *ObjectId* к каждому документу:

```
db.catalogue.insertMany(
[
  { item: "journal", qty: 25, tags: ["blank", "red"], size: { h: 14, w: 21, uom: "cm" } },
  { item: "mat", qty: 85, tags: ["gray"], size: { h: 27.9, w: 35.5, uom: "cm" } },
  { item: "mousepad", qty: 25, tags: ["gel", "blue"], size: { h: 19, w: 22.85, uom: "cm" } }
])
```

Чтобы получить вставленные документы, следует выполнить следующее:

db.catalogue.find({})

3.5. Выборка документов

Операции чтения извлекают документы из коллекции, то есть запрашивают коллекцию для документов, хранящихся в базе данных.

db.collection.find(*query*, *projection*) – метод для чтения документов из коллекции, который выбирает документы в коллекции или представлении и возвращает курсор к выбранным документам.

Параметры метода:

- *query* – документ, который определяет фильтр выбора с использованием операторов запроса. Чтобы вернуть все документы в коллекции, пропустите этот параметр или передайте пустой документ ({});
- *projection* – документ, который определяет поля для возврата в документах, которые соответствуют фильтру запросов. Чтобы вернуть все поля в соответствующих документах, следует пропустить этот параметр.

Задав значения параметров *query* и *projection*, можно указать фильтры запросов или критерии, которые идентифицируют документы для возврата. Ниже приведены общие описания этих параметров.

Параметр *projection* определяет, какие поля возвращаются в соответствующих документах. Этот параметр принимает документ следующей формы:

{ *field*1: <*value*>, *field*2: <*value*> ... }

<*value*> может быть любым из следующих:

- 1 или *true*, чтобы включить поле в возвратные документы;
- 0 или *false*, чтобы исключить поле;
- выражение с использованием операторов проекции.

Следует отметить, что для поля *_id* не нужно явно указывать *_id*: 1 для возврата поля *_id*. Метод *find*() всегда возвращает поле *_id*, если вы не укажете *_id*: 0 для подавления поля.

Ниже будут представлены примеры операций запроса с использованием метода *db.collection.find*() в оболочке *mongo*. Примеры будут использовать коллекцию *catalogue*. Чтобы заполнить эту коллекцию,

выполните следующее:

db.catalogue.insertMany([
{ *item*: "*journal*", *qty*: 25, *size*: { *h*: 14, *w*: 21, *uom*: "*cm*" }, *status*: "*A*" },
{ *item*: "*notebook*", *qty*: 50, *size*: { *h*: 8.5, *w*: 11, *uom*: "*in*" }, *status*: "*A*" },
{ *item*: "*paper*", *qty*: 100, *size*: { *h*: 8.5, *w*: 11, *uom*: "*in*" }, *status*: "*D*" },
{ *item*: "*planner*", *qty*: 75, *size*: { *h*: 22.85, *w*: 30, *uom*: "*cm*" }, *status*: "*D*" },
{ *item*: "*postcard*", *qty*: 45, *size*: { *h*: 10, *w*: 15.25, *uom*: "*cm*" }, *status*: "*A*" }
]);

Метод *find*() без параметров возвращает все документы из коллекции и возвращает все поля для документов. Например, следующая операция возвращает все документы в коллекции *catalogue*:

db.catalogue.find()

Эта операция соответствует следующему оператору *SQL*:

*SELECT * FROM catalogue*

Чтобы указать условия равенства, используйте выражения <*field*>: <*value*> в документе фильтра запросов: { <*field*1>: <*value*1>, ... }.

В следующем примере из коллекции *catalogue* выбираются все документы, статус которых равен "*D*":

db.catalogue.find({ *status*: "*D*" })

Эта операция соответствует следующему оператору *SQL*:

*SELECT * FROM catalogue WHERE status = "D"*

Документ фильтра запросов может использовать операторы запросов для указания условий в следующей форме:

{<*field*1>: { <operator1>: <*value*1> }, ... }

В следующем примере извлекаются все документы из коллекции *catalogue*, где статус равен "*A*" или "*D*":

db.catalogue.find({ *status*: { $*in*: ["*A*", "*D*"] } })

Операция соответствует следующему оператору *SQL*:

*SELECT * FROM catalogue WHERE status in* ("*A*", "*D*")

Составной запрос может указывать условия для нескольких полей в документах коллекции. Логическое соединение *AND* соединяет предложения составного запроса, так что запрос выбирает документы в коллекции, которые соответствуют всем условиям.

Ниже перечислены и описаны основные логические операторы в сравнении с аналогичными операторами языка *SQL* (табл. 3.2).

Таблица 3.2

Основные логические операторы *MongoDB* и *SQL*

Оператор *SQL*	***MongoDB***	**Описание**
<	*$lt*	Меньше
<=	*$lte*	Меньше или равно
>	*$gt*	Больше
>=	*$gte*	Больше или равно
<>	*$ne*	Не равно
NOT	*$not*	Отрицание
EXISTS	*$exists*	Проверка существования поля
OR	*$or*	Или
NOT OR	*$nor*	Не или
RLIKE,	*$regex*	Соответствие регулярному выражению
REGEXP		
LIKE	*$elemMatch*	Соответствие всех полей вложенного документа
–	*$size*	Соответствие размеру массива
–	*$type*	Соответствие, если поле имеет указанный тип

В следующем примере извлекаются все документы в коллекции *catalogue*, где статус равен "*A*", а *qty* меньше ($*lt*) 30:

db.catalogue.find({ *status*: "*A*", *qty*: { $*lt*: 30 } })
Операция соответствует следующему оператору *SQL*:
*SELECT * FROM catalogue WHERE status* = "*A*" *AND qty* < 30

Используя оператор $*or*, вы можете указать составной запрос, который объединяет каждое предложение с логическим соединением *OR*, чтобы запрос выбирал документы в коллекции, которые соответствуют хотя бы одному условию.

В следующем примере извлекаются все документы в коллекции, где статус равен "*A*" или *qty* меньше ($*lt*) 30:

db.catalogue.find({ $*or*: [{ *status*: "*A*" }, { *qty*: { $*lt*: 30 } }] })

Операция соответствует следующему оператору языка *SQL*:

*SELECT * FROM catalogue WHERE status* = "*A*" *OR qty* < 30

Следует отметить, что *MongoDB* рассматривает некоторые типы данных как эквивалентные для целей сравнения. Например, числовые типы подвергаются преобразованию перед сравнением. Однако для большинства типов данных операторы сравнения выполняют сравнения только с теми документами, в которых тип *BSON* целевого поля соответствует типу операнда запроса.

СУБД позволяет комбинировать логические операторы сравнения. В следующем примере документ составного запроса выбирает все документы в коллекции, где статус равен "*A*", а *qty* меньше ($*lt*) 30 или элемент начинается с символа *p*:

```
db.catalogue.find( {
    status: "A",
    $or: [ { qty: { $lt: 30 } }, { item: /^p/ } ]
} )
```

Приведенная операция соответствует следующему оператору языка *SQL*:

*SELECT * FROM catalogue*
WHERE status = "*A*" *AND* (*qty* < 30 *OR item LIKE* "*p*%")

MongoDB поддерживает регулярные выражения $*regex* для выполнения поиска по строковому шаблону. СУБД использует *Perl*-совместимые регулярные выражения с поддержкой *UTF*-8.

Чтобы использовать $*regex*, используйте один из следующих синтаксисов:

{ <*field*>: { $regex: /*pattern*/, $*options*: '<*options*>' } }
{ <*field*>: { $regex: *'pattern'*, $*options*: '<*options*>' } }
{ <*field*>: { $*regex*: /*pattern*/<*options*> } }

Пусть у нас есть коллекция *products* со следующими документами:

{ "*_id*" : 100, "*sku*" : "*abc*123", "*description*" : "*Single line description*." }
{ "*_id*" : 101, "*sku*" : "*abc*789", "*description*" : "*First line\nSecond line*" }
{ "*_id*" : 102, "*sku*" : "*xyz*456", "*description*" : "*Many spaces before line*" }
{ "*_id*" : 103, "*sku*" : "*xyz*789", "*description*" : "*Multiple\nline description*" }

В следующем примере сопоставляются все документы, где поле *sku* похоже на % 789:

db.products.find({ *sku*: { $*regex*: /789$/ } })

Пример аналогичен следующему оператору языка *SQL LIKE*:

*SELECT * FROM products*
WHERE sku like "%789";

Работы с вложенными документами

В качестве поля документа в *MongoDB* может выступать другой документ, который называется вложенным документом. СУБД имеет средства для поиска во вложенных документах. Чтобы указать условие равенства для поля, являющегося вложенным документом, нужно использовать документ фильтра запросов в виде: {<*field*>: <*value*>}, где <*value*> – это документ для сопоставления.

Например, следующий запрос выбирает все документы, поле *size* которого равно документу { *h*: 14, *w*: 21, *uom*: "*cm*" }:

db.catalogue.find({ *size*: { *h*: 14, *w*: 21, *uom*: "*cm*" } })

Соответствия равенства во всем внедренном документе требуют точного соответствия указанного документа <*value*>, включая порядок полей. Например, следующий запрос не соответствует ни одному документу в коллекции *catalogue*:

db.catalogue.find({ *size*: { *w*: 21, *h*: 14, *uom*: "*cm*" } })

Чтобы указать условие запроса для полей во вложенном документе, используйте точечную нотацию ("*field.nestedField*").

При запросе с использованием точечной нотации поле и вложенное поле должны быть заключены в кавычки.

В следующем примере выбираются все документы, в которых поле *uom*, вложенное в поле размера, равно "*in*":

db.catalogue.find({ "*size.uom*": "*in*" })

Документ фильтра запросов может использовать операторов запросов для указания условий в следующей форме: {<*field*1>: { <*operator*1>: <*value*1> }, ...}.

В следующем запросе используется оператор меньше чем ($*lt*) для поля *h*, встроенного в поле *size*:

db.catalogue.find({ "*size.h*": { $*lt*: 15 } })

Следующий запрос выбирает все документы, где вложенное поле *h* меньше 15, вложенное поле *uom* равно "*in*", а поле *status* равно "*D*":

db.catalogue.find({ "*size.h*": { $*lt*: 15 }, "*size.uom*": "*in*", *status*: "*D*" })

Работа с массивами

СУБД содержит развитые средства для осуществления поиска в массивах, в том числе в массивах документов. Ниже будут представлены примеры операций запроса с использованием метода *db.collection.find*(). Примеры будут использовать коллекцию *catalogue*. Чтобы заполнить ее, выполните следующее:

db.catalogue.insertMany([
{ *item*: "*journal*", *qty*: 25, *tags*: ["*blank*", "*red*"], *dim_cm*: [14, 21] },
{ *item*: "*notebook*", *qty*: 50, *tags*: ["*red*", "*blank*"], *dim_cm*: [14, 21] },
{ *item*: "*paper*", *qty*: 100, *tags*: ["*red*", "*blank*", "*plain*"], *dim_cm*: [14, 21] },
{ *item*: "*planner*", *qty*: 75, *tags*: ["*blank*", "*red*"], *dim_cm*: [22.85, 30] },
{ *item*: "*postcard*", *qty*: 45, *tags*: ["*blue*"], *dim_cm*: [10, 15.25] }
]);

Чтобы указать условие равенства для массива, нужно использовать документ запроса {<*field*>: <*value*>}, где <*value*> – массив для сопоставления, включая порядок элементов. В следующем примере запрашиваются все документы, в которых значение поля *tags* представляет собой массив с ровно двумя элементами, "*red*" и "*blank*", в указанном порядке:

db.catalogue.find({ *tags*: ["*red*", "*blank*"] })

Если вместо этого вы хотите найти массив, который содержит как элементы "*red*", так и "*blank*", независимо от порядка или других элементов в массиве, используйте оператор $*all*:

db.catalogue.find({ *tags*: { $*all*: ["*red*", "*blank*"] } })

Чтобы запросить, содержит ли поле массива хотя бы один элемент с указанным значением, используйте фильтр {<*field*>: <*value*>}, где <*value*> – значение элемента.

В следующем примере запрашиваются все документы, в которых тегами является массив, содержащий в качестве одного из элементов строку "*red*":

db.catalogue.find({ *tags*: "*red*" })

Чтобы указать условия для элементов в поле массива, используйте операторы запросов в документе фильтра запросов: {<*array field*>: {<*operator*1>:<*value*1>, ...}}.

Например, следующая операция запрашивает все документы, где массив *dim_cm* содержит хотя бы один элемент, значение которого больше 25:

db.catalogue.find({ *dim_cm*: { $*gt*: 25 } })

При указании составных условий для элементов массива можно указывать запрос так, чтобы либо один элемент массива удовлетворял бы этим условиям, либо любая комбинация элементов массива удовлетворяла условиям.

В следующем примере запрашиваются документы, в которых массив *dim_cm* содержит элементы, которые в некоторой комбинации удовлетворяют условиям запроса:

db.catalogue.find({ *dim_cm*: { $*gt*: 15, $*lt*: 20 } })

Один элемент может удовлетворять условию больше 15, а другой элемент – условию меньше 20, или один элемент может удовлетворять обоим условиям.

Оператор $*elemMatch* можно использовать, чтобы указать несколько критериев для элементов массива, чтобы хотя бы один элемент массива удовлетворял всем указанным критериям.

В следующем примере запрашиваются документы, в которых массив *dim_cm* содержит хотя бы один элемент, который больше ($*gt*) 22 и меньше ($*lt*) 30:

db.catalogue.find({ *dim_cm*: { $*elemMatch*: { $*gt*: 22, $*lt*: 30 } } })

Используя точечную запись, можно указать условия запроса для элемента с определенным индексом или позицией массива, при этом следует помнить, что массив использует индексацию с нуля. При запросе с использованием точечной нотации поле и вложенное поле должны быть заключены в кавычки.

В следующем примере запрашиваются все документы, где второй элемент в массиве *dim_cm* больше 25:

db.catalogue.find({ "*dim_cm*.1": { $*gt*: 25 } })

Оператор $*size* используется для запроса массивов по количеству элементов. Например, следующее выбирает документы, в которых теги массива имеют три элемента:

db.catalogue.find({ "*tags*": { $*size*: 3 } })

Работа с вложенными в массив документами

Ниже будут представлены примеры операций запроса, вложенных в массив документов. Примеры будут использовать коллекцию *catalogue*. Чтобы заполнить коллекцию инвентаря, выполните следующее:

```
db.catalogue.insertMany( [
   { item: "journal", instock: [ { warehouse: "A", qty: 5 }, { warehouse: "C", qty: 15 } ] },
   { item: "notebook", instock: [ { warehouse: "C", qty: 5 } ] },
   { item: "paper", instock: [ { warehouse: "A", qty: 60 }, { warehouse: "B", qty: 15 } ] },
   { item: "planner", instock: [ { warehouse: "A", qty: 40 }, { warehouse: "B", qty: 5 } ] },
   { item: "postcard", instock: [ { warehouse: "B", qty: 15 }, { warehouse: "C", qty: 35 } ] }
]);
```

В следующем примере выбираются все документы, где элемент в массиве *instock* соответствует указанному документу:

```
db.catalogue.find( { "instock": { warehouse: "A", qty: 5 } } )
```

Соответствия равенства для всего вложенного документа требуют точного соответствия указанного документа, включая порядок полей. Например, следующий запрос не соответствует ни одному документу в коллекции:

```
db.catalogue.find( { "instock": { qty: 5, warehouse: "A" } } )
```

Если вам неизвестна позиция индекса документа, вложенного в массив, объедините имя поля массива с точкой (.) и именем поля во вложенном документе.

В следующем примере выбираются все документы, для которых в массиве *instock* есть хотя бы один встроенный документ, содержащий поле *qty*, значение которого меньше или равно 20:

```
db.catalogue.find( { 'instock.qty': { $lte: 20 } } )
```

Используя точечную запись, вы можете указать условия запроса для поля в документе по определенному индексу или позиции массива. Массив использует индексацию с нуля.

В следующем примере выбираются все документы, для которых массив *instock* имеет в качестве первого элемента документ, содержащий поле *qty*, значение которого меньше или равно 20:

db.catalogue.find({ *'instock*.0.*qty'*: { $*lte*: 20 } })

При указании условий для нескольких полей, вложенных в массив документов, вы можете указать запрос так, чтобы либо один документ удовлетворял этим условиям, либо любая комбинация документов (включая один документ) в массиве удовлетворяла условиям.

Оператор $*elemMatch* используется, чтобы указать несколько критериев для массива встроенных документов, чтобы хотя бы один встроенный документ удовлетворял всем указанным критериям.

В следующем примере запрашиваются документы, в которых массив *instock* содержит хотя бы один встроенный документ, содержащий поле *qty*, равное 5, и поле хранилища, равное "*A*":

db.catalogue.find({ "*instock*": { $*elemMatch*: { *qty*: 5, *warehouse*: "*A*" } } })

В следующем примере запрашиваются документы, для которых в массиве *instock* есть хотя бы один внедренный документ, содержащий поле *qty*, которое больше 10 и меньше или равно 20:

db.catalogue.find({ "*instock*": { $*elemMatch*: { *qty*: { $*gt*: 10, $*lte*: 20 } } } })

Если условия составного запроса в поле массива не используют оператор *$elemMatch*, запрос выбирает те документы, массив которых содержит любую комбинацию элементов, удовлетворяющую условиям.

Например, следующий запрос сопоставляет документы, где любой документ, вложенный в массив *instock*, имеет поле *qty* больше 10, а любой документ (но не обязательно тот же внедренный документ) в массиве имеет поле *qty* меньше или равно 20:

db.catalogue.find({ "*instock.qty*": { $*gt*: 10, $*lte*: 20 } })

В следующем примере запрашиваются документы, в которых массив *instock* содержит, по меньшей мере, один встроенный доку-

мент с полем *qty*, равным 5, и, по меньшей мере, один встроенный документ (но не обязательно тот же встроенный документ) с полем хранилища, равным "*A*":

db.catalogue.find({ "*instock.qty*": 5, "*instock.warehouse*": "*A*" })

Определение структуры возвращаемых запросом данных

По умолчанию запросы в *MongoDB* возвращают все поля в соответствующих документах. Чтобы ограничить объем данных, которые *MongoDB* отправляет приложениям, вы можете использовать документ проекции для указания или ограничения полей для возврата.

Ниже представлены примеры операций запроса с проекцией с применением метода *db.collection.find*(), использующие коллекцию *catalogue*. Чтобы заполнить коллекцию *catalogue*, нужно выполнить следующее:

db.catalogue.insertMany([
{ item: "journal", status: "A", size: { h: 14, w: 21, uom: "cm" }, instock: [{ warehouse: "A", qty: 5 }] },
{ item: "notebook", status: "A", size: { h: 8.5, w: 11, uom: "in" }, instock: [{ warehouse: "C", qty: 5 }] },
{ item: "paper", status: "D", size: { h: 8.5, w: 11, uom: "in" }, instock: [{ warehouse: "A", qty: 60 }] },
{ item: "planner", status: "D", size: { h: 22.85, w: 30, uom: "cm" }, instock: [{ warehouse: "A", qty: 40 }] },
{ item: "postcard", status: "A", size: { h: 10, w: 15.25, uom: "cm" }, instock: [{ warehouse: "B", qty: 15 }, { warehouse: "C", qty: 35 }] }
]);

Если не укажете документ проекции, метод *db.collection.find*() вернет все поля в соответствующих документах. В следующем примере возвращаются все поля из всех документов в инвентаре, где поле *status* равно "*A*":

db.catalogue.find({ *status*: "*A*" })

Приведенная операция соответствует следующему оператору языка *SQL*:

*SELECT * from catalogue WHERE status = "A"*

Документ проекции может явно включать несколько полей, установив для <*field*> значение 1 в документе проекции. Следующая операция возвращает все документы, которые соответствуют запросу:

db.catalogue.find({ *status*: "*A*" }, { *item*: 1, *status*: 1 })

В наборе результатов в соответствующих документах возвращаются только поля *item*, *status* и по умолчанию поле *_id*. Операция соответствует следующему оператору *SQL*:

SELECT _id, item, status from catalogue WHERE status = "*A*"

Поле *_id* можно удалить из результатов вывода, установив 0 в параметрах документа проекции, как в следующем примере:

db.catalogue.find({ *status*: "*A*" }, { *item*: 1, *status*: 1, *_id*: 0 })

Операция соответствует следующему оператору *SQL*:

SELECT item, status from catalogue WHERE status = "*A*"

Существует возможность вернуть все, кроме исключенных полей: вместо перечисления полей для возврата нужно использовать соответствующий документ проекции с перечислением определенных полей, подлежащих исключению. В следующем примере возвращаются все поля, кроме полей статуса и запасов в соответствующих документах:

db.catalogue.find({ *status*: "*A*" }, { *status*: 0, *instock*: 0 })

Чтобы осуществить возврат определенных полей во встроенных документах, нужно использовать точечную запись для обозначения встроенного поля и установить значение 1 в документе проекции, например:

db.catalogue.find({ *status*: "*A*" }, { *item*: 1, *status*: 1, "*size.uom*": 1 })

Приведенный выше пример возвращает поле *_id* (возвращается по умолчанию), поле *item*, поле *status*, поле *uom*, которое отображается встроенным в документ *size*.

Для того чтобы не выводить определенные поля во встроенных документах, нужно использовать точечную нотацию для ссылки на встроенное поле в проекционном документе и установить значение 0. В следующем примере указывается проекция для исключения поля *uom* в документе размера. Все остальные поля возвращаются в соответствующих документах:

db.catalogue.find({ *status*: "*A*" }, { "*size.uom*": 0 })

Для того чтобы определить формат вывода вложенных документов в массиве, нужно использовать точечную нотацию для проекции определенных полей внутри документов, встроенных в массив, например:

db.catalogue.find({ *status*: "*A*" }, { *item*: 1, *status*: 1, "*instock.qty*": 1 })

Приведенный пример возвращает поле *_id* (возвращается по умолчанию), поле *item*, поле *status*, поле *qty* в документах, встроенных в массив *instock*.

Для полей, которые содержат массивы, *MongoDB* предоставляет следующие операторы проекции для управления массивами: $*elemMatch*, $*slice* и $.

В следующем примере используется оператор проекции $*slice* для возврата последнего элемента в массиве *instock*:

db.catalogue.find({*status*:"*A*"},{*item*:1, *status*:1, *instock*:{ $*slice*: -1}})

$*elemMatch*, $*slice* и $ являются единственным способом проецирования определенных элементов для включения в возвращаемый массив. Проецировать определенные элементы массива, используя индекс массива, невозможно. Например, проекция {'*instock*.0': 1} не будет проецировать массив с первым элементом.

Сортировка документов в запросе

Для сортировки результата запроса, инициируемого методом *find*(), используется присоединенный метод *sort*():

db.<Коллекция>.*find*().*sort*({<Атрибут>: 1 |-1,..}).

Имя атрибута – имя поля, по которому необходимо осортировать. При этом, если значением параметра <Атрибут> будет 1, то это приведет к сортировке по возрастанию, а -1 – по убыванию значений атрибута. Например,

db.authors.find({}, {*_id*:0,"*Au_lname*": 1, "*Au_fname*": 1}).*sort* ({"*Au_lname*": 1})

выведет фамилии и имена авторов, отсортированные по возрастанию фамилии.

Ограничение множества выводимых документов

Для ограничения количества выводимых документов используются методы:

- *limit*(<Количество выводимых документов>);
- *skip*(<Количество пропускаемых в начале документов>).

Например, запрос

db.authors.find(*null*, {*_id*:0, "*Au_lname*":1})
.limit(5).*skip*(2).*sort*({"*Au_ lname*": -1})

выведет из коллекции *authors* отсортированные по убыванию фамилии пяти авторов, пропустив при этом две первые фамилии.

3.6. Обновление документов

Обновить существующий документ или документы в коллекции можно с использованием следующих методов оболочки *mongo*:

- *db.collection.updateOne*(<*filter*>, <*update*>, <*options*>) – обновляет не более одного документа, который соответствует указанному фильтру, хотя несколько документов могут соответствовать указанному фильтру;
- *db.collection.updateMany*(<*filter*>, <*update*>, <*options*>) – обновляет все документы, которые соответствуют указанному фильтру;
- *db.collection.replaceOne*(<*filter*>, <*update*>, <*options*>) – заменяет не более одного документа, который соответствует указанному фильтру, даже если указанному фильтру может соответствовать несколько документов;
- *db.collection.update*(<*filter*>, <*update*>, <*options*>) – обновляет или заменяет один документ, который соответствует указанному фильтру, или обновляет все документы, которые соответствуют ука-

занному фильтру. По умолчанию метод обновляет один документ. Чтобы обновить несколько документов, используют параметр *multi*.

В приведенных выше общих определениях методов используются параметры:

• <*filter*> – документ, определяющий критерии выбора для обновления, в которм доступны те же селекторы запросов, что и в методе *find*();

• {} – обновит первый документ, возвращенный в коллекции;

• <*update*> – документ или конвейер агрегации, который содержит выражения оператора обновления или следующие этапы агрегации: $*addFields* или его псевдоним $*set*, $*project* и его псевдоним $*unset* $*replaceRoot* и его псевдоним $*replaceWith*;

• <*options*> – задают поведение метода и не являются обязательными:

• *upsert* (типа *boolean*) со значением *true* при отсутствии документа, удовлетворяющего условию селектора, добавляет документ с заданными атрибутами, а значении *false* (значение по умолчанию) новый документ не создается;

• *multi* (типа *boolean*): по значению *true* обновляются все документы, удовлетворяющие условию селектора, а при значении *false* (значение по умолчанию) обновляется один документ.

Добавление, удаление или изменение атрибутов в документе задается модификаторами (*Modifier*), устанавливаемыми перед их значениями. Изменяемые атрибуты записываются в виде:

```
{
 <Modifier1>: { <field1>: <value1>, ... },
 <Modifier1>: { <field2>: <value2>, ... },
 ...
}
```

Ниже приводятся модификаторы, используемые в методе *update*() (табл. 3.3).

Таблица 3.3

Сводная таблица модификаторов метода *update*()

Модификатор	Описание
$set	Обновляет, а при отсутствии – создает атрибут
$unset	Удаляет атрибут
$inc	Увеличивает значение атрибута на заданное число

Окончание табл. 3.3

Модификатор	Описание
$pop	При значении 1 удаляет последний, при -1 – первый элемент массива
$push	Добавляет в массив новый элемент
$pushAll	Помещает несколько новых элементов в массив
$addToSet	Добавляет новый элемент в массив (исключаются дубликаты)
$pull	Удаляет из массива значение (при его наличии)
$pullAll	Удаляет из массива все подходящие значения

Общая специфика, которую необходимо учитывать при обновлении документов, заключается в следующем:

- все операции записи являются атомарными и проводятся на уровне одного документа;
- невозможно обновить значение поля *_id*, а также нельзя заменить существующий документ документом замены с другим значением поля *_id*;
- порядок полей сохраняется после операций записи, при этом:

– поле *_id* всегда является первым полем в документе;

– обновления, которые включают переименование имен полей, могут привести к переупорядочению полей в документе;

– если *updateOne*(), *updateMany*() или *replaceOne*() включают в себя *upsert*: *true* и ни один документ не соответствует указанному фильтру, то операция создает новый документ и вставляет его. Если есть соответствующие документы, то операция изменяет или заменяет соответствующий документ или документы;

– некоторые операторы обновления, такие как $*set*, если указанное для обновления поле не существует, создают его и выполняют запрос.

Рассмотрим примеры использования различных вариантов обновления одного и многих документов. Приводимые ниже примеры будут использовать коллекцию *catalogue*. Чтобы создать и/или заполнить ее, следует выполнить следующее:

```
db.catalogue.insertMany( [
   { item: "canvas", qty: 100, size: { h: 28, w: 35.5, uom: "cm" }, status: "A" },
   { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
```

```
    { item: "mat", qty: 85, size: { h: 27.9, w: 35.5, uom: "cm" }, status: "A" },
    { item: "mousepad", qty: 25, size: { h: 19, w: 22.85, uom: "cm" }, status: "P" },
    { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "P" },
    { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
    { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
    { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" },
    { item: "sketchbook", qty: 80, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
    { item: "sketch pad", qty: 95, size: { h: 22.85, w: 30.5, uom: "cm" }, status: "A" }
  ] );
```

Обновление одного документа

В следующем примере метод *db.collection.updateOne*() в коллекции *catalogue* используется для обновления первого соответствующего документа, в котором элемент равен "*paper*":

```
db.catalogue.updateOne(
  { item: "paper" },
  {
    $set: { "size.uom": "cm", status: "P" },
    $currentDate: { lastModified: true }
  }
)
```

В примере используется оператор $*set* для обновления значения поля *size.uom* до "*cm*" и значения поля *status* до "*P*", а также оператор $*currentDate* для обновления значения поля *lastModified* до текущей даты. Если поле *lastModified* не существует, $*currentDate* его создаст.

Обновление нескольких документов

В следующем примере метод *db.collection.updateMany*() в коллекции *catalogue* используется для обновления всех соответствующих документов, где поле *qty* меньше 50:

```
db.catalogue.updateMany(
   { "qty": { $lt: 50 } },
   {
     $set: { "size.uom": "in", status: "P" },
     $currentDate: { lastModified: true }
   }
)
```

В примере использует оператор $*set* для обновления значения поля *size.uom* на "*in*" и значения поля *status* на "*P*", а также оператор $*currentDate* для обновления значения поля *lastModified* до текущей даты. Если поле *lastModified* не существует, $*currentDate* его создаст.

Замена документа

Чтобы заменить все содержимое документа, нужно передать полностью новый документ в качестве второго аргумента в метод *db.collection.replaceOne*().

При замене документа заменяющий документ должен состоять только из пар «поле–значение» и не включать выражения операторов обновления.

Документ замены может иметь поля, отличные от исходного документа. В документе замены вы можете опустить поле *_id*, так как оно является неизменным. В запрос можно включить поле *_id*, тогда оно должно иметь то же значение, что и текущее.

Следующий пример заменяет первый документ из коллекции *catalogue*, где поле *item* равно "*paper*":

```
db.catalogue.replaceOne(
   { item: "paper" },
   { item: "paper", instock: [ { warehouse: "A", qty: 60 }, { warehouse: "B", qty: 40 } ] })
```

3.7. Удаление документов

Удалить существующий документ или документы в коллекции можно с использованием следующих методов оболочки *mongo*:

- *db.collection.deleteMany*() – удаляет все документы, соответствующие фильтру, из коллекции;
- *db.collection.deleteOne*() – удаляет один документ из коллекции.

Общая специфика обновления документов следующая:

• операции удаления не удаляют индексы, даже если удаляются все документы из коллекции;

• все операции записи в *MongoDB* являются атомарными на уровне одного документа.

Примеры на этой странице используют коллекцию *catalogue*. Чтобы заполнить коллекцию *catalogue*, выполните следующее:

```
db.catalogue.insertMany( [
   { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
   { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "P" },
   { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
   { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
   { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" },
] );
```

Удаление нескольких документов

Чтобы удалить все документы из коллекции, передайте пустой документ фильтра {} методу *db.collection.deleteMany*(). В следующем примере удаляются все документы из инвентарной коллекции:

```
db.catalogue.deleteMany({})
```

Метод возвращает документ со статусом операции.

Удалить все документы, соответствующие условию, можно, указав критерии или фильтры, которые определяют документы для удаления. Фильтры используют тот же синтаксис, что и операции чтения.

Чтобы указать условия равенства, используйте выражения следующего вида в документе фильтра запросов:

```
{ <field1>: <value1>, <field1>: <value1>, ... }
```

Документ фильтра запросов может использовать операторы запросов для указания условий и в более сложной форме:

{ <*field*1>: { <*operator*1>: <*value*1> }, ... }

Чтобы удалить все документы, которые соответствуют критериям удаления, необходимо передать параметр фильтра методу *deleteMany*().

В следующем примере удаляются все документы из коллекции *catalogue*, где поле *status* равно "*A*":

db.catalogue.deleteMany({ *status* : "*A*" })

Метод возвращает документ со статусом операции.

Удаление только одного документа, который соответствует условию

Чтобы удалить не более одного документа, который соответствует указанному фильтру (даже если несколько документов могут соответствовать указанному фильтру), необходимо использовать метод *db.collection.deleteOne*().

В следующем примере удаляется первый документ с полем *status* равным *"D"*:

db.catalogue.deleteOne({ *status*: "*D*" })

3.8. Операции массовой записи

Кроме традиционных операций изменения данных, описанных выше и предполагающих манипуляции с определенной операцией с небольшим количеством документов, *MongoDB* предоставляет клиентам возможность выполнять несколько операций записи в больших объемах одной коллекции.

Метод *db.collection.bulkWrite***()** предоставляет возможность выполнять массовые операции вставки, обновления и удаления.

Метод *bulkWrite*() поддерживает следующие операции записи: вставку (*insertOne*), обновление (*updateOne*, *updateMany*), замену (*replaceOne*) и удаление (*deleteOne*, *deleteMany*). Каждая операция записи передается в метод *bulkWrite*() как документ в массиве.

Метод имеет следующий общий синтаксис:

db.collection.bulkWrite([<*operation* 1>, <*operation* 2>, ...],
{

```
    writeConcern: <document>,
    ordered: <boolean>
  }
)
```

В данной записи [<*operation* 1>, <*operation* 2>, ...] – массив операций записи. Допустимые операции: *insertOne*, *updateOne*, *updateMany*, *deleteOne*, *deleteMany*, *replaceOne*; *writeConcern* – не обязательный параметр, представляющий собой документ, выражающий озабоченность по поводу записи; *ordered* – не обязательный параметр со значением по умолчанию – *true*, представляющий собой логическое значение, которое указывает, должен ли экземпляр *mongod* выполнять упорядоченную или неупорядоченную операцию.

После своего выполнения метод возвращает:

- логическое значение *true*, если операция выполнялась с проблемой записи, или *false*, если проблема записи была отключена;
- количество для каждой операции записи;
- массив, содержащий *_id* для каждого успешно вставленного или добавленного документа;
- операции массовой записи могут быть как упорядоченными, так и неупорядоченными.

С упорядоченным списком *MongoDB* выполняет операции последовательно. Если во время обработки одной из операций записи произойдет ошибка, *MongoDB* вернется без обработки оставшихся операций записи в списке.

С неупорядоченным списком операций *MongoDB* может выполнять операции параллельно, но такое поведение не гарантируется. Если во время обработки одной из операций записи произойдет ошибка, *MongoDB* продолжит обрабатывать оставшиеся операции записи в списке.

Выполнение записи упорядоченного списка операций в изолированном наборе обычно выполняется медленнее, чем выполнение записи неупорядоченного списка, поскольку в упорядоченном каждая операция должна ждать завершения предыдущей.

По умолчанию *bulkWrite*() выполняет упорядоченные операции. Чтобы указать, что необходимо производить неупорядоченные операции записи, нужно установить значение пары *order*: *false* в документе параметров.

Пусть коллекция *characters* содержит следующие документы:

```
{ "_id" : 1, "char" : "Brisbane", "class" : "monk", "lvl" : 4 },
{ "_id" : 2, "char" : "Eldon", "class" : "alchemist", "lvl" : 3 },
{ "_id" : 3, "char" : "Meldane", "class" : "ranger", "lvl" : 3 }
```

Следующий *bulkWrite*() выполняет несколько операций над этой коллекцией: вставка, обновление и удаление:

```
try {
  db.characters.bulkWrite(
    [
      { insertOne :
        {
          "document" :
          {
            "_id" : 4, "char" : "Dithras", "class" : "barbarian", "lvl" :
4
          }
        }
      },
      { insertOne :
        {
          "document" :
          {
            "_id" : 5, "char" : "Taeln", "class" : "fighter", "lvl" : 3
          }
        }
      },
      { updateOne :
        {
          "filter" : { "char" : "Eldon" },
          "update" : { $set : { "status" : "Critical Injury" } }
        }
      },
      { deleteOne :
        { "filter" : { "char" : "Brisbane"} }
      },
      { replaceOne :
        {
          "filter" : { "char" : "Meldane" },
```

```
            "replacement" : { "char" : "Tanys", "class" : "oracle", "lvl" : 4 }
          }
        }
      ]
  );
}
catch (e) {
  print(e);
}
```

Результат выполнения операции возвращает следующее:

```
{
  "acknowledged" : true,
  "deletedCount" : 1,
  "insertedCount" : 2,
  "matchedCount" : 2,
  "upsertedCount" : 0,
  "insertedIds" : {
    "0" : 4,
    "1" : 5
  },
  "upsertedIds" : {

  }
}
```

3.9. Операции агрегации. Технология *MapReduce*

Агрегация – это группировка значений многих документов. Операции агрегирования позволяют манипулировать сгруппированными данными. В языке *SQL* аналогом агрегации является команда *group by*.

Операции агрегации обрабатывают записи данных и возвращают вычисленные результаты. Операции агрегации группируют значения из нескольких документов вместе, могут выполнять различные операции над сгруппированными данными для возврата одного результата.

MongoDB предоставляет три способа выполнения агрегации: конвейер агрегации, технология *MapReduce* и агрегирования по одному назначению.

Конвейер агрегации

Структура агрегирования *MongoDB* основана на концепции конвейеров обработки данных. Эта концепция дает возможность выполнить операции над некоторым вводом и использовать вывод как ввод для следующей команды. *MongoDB* также поддерживает данную концепцию.

Ниже приведен список возможных этапов, и каждый из них может быть взят, как множество документов для ввода и сгенерирует набор документов, полученных в результате обработки:

- *$project* – используется для выбора некоторых специальных полей из коллекции;
- *$match* – операция фильтрации, которая может уменьшить количество документов, которые передаются для ввода в следующий этап;
- *$group* – непосредственно, сама агрегация (группировка);
- *$sort* – сортирует документы;
- *$skip* – игнорирование списка документов в имеющемся множестве;
- *$limit* – ограничивает количество документов для вывода на количество, переданное методу, начиная, с текущей позиции;
- *$unwind* – позволяет развернуть поля-подмассивы путем дублирования в выборке родительской сущности для каждого поля такого подмассива.

Документы поступают в многоступенчатый конвейер, который преобразует их в агрегированный результат. Основные этапы конвейера предоставляют фильтры, которые работают как запросы и преобразования документов, и изменяют форму выходного документа.

Другие операции конвейера предоставляют инструменты для группировки и сортировки документов по конкретному полю или полям, а также инструменты для агрегирования содержимого массивов, включая массивы документов. Кроме того, этапы конвейера могут использовать операторы для таких задач, как вычисление среднего значения или конкатенация строки.

В *MongoDB* для агрегации используется метод *aggregate*(). Например:

```
db.orders.aggregate([
   { $match: { status: “A” } },
   { $group: { _id: “$cust_id”, total: { $sum: “$amount” } } }
])
```

В операторе присутствует два последовательных этапа. Разберем их работу.

Первый этап: этап $*match* фильтрует документы по полю *status* и передает на следующий этап те документы, которые имеют статус, равный «*A*».

Второй этап: этап $*group* группирует документы по полю *cust_id*, чтобы вычислить сумму по полю *sum* для каждого уникального *cust_id*.

В целом конвейер обеспечивает эффективное агрегирование данных с использованием собственных операций в *MongoDB* и является предпочтительным методом агрегирования данных в *MongoDB*. В табл. 3.4 приведен список операций для агрегации документов.

Таблица 3.4

Сводная таблица модификаторов метода *update*()

Выражение	**Описание**	**Пример**
$sum	Суммирует указанные значения всех документов в коллекции	*db.*ИМЯ_КОЛЛЕКЦИИ.*aggregate*([{*$group: {_id : "$title",total_salary: {$sum : "$salary"*}}}])
$avg	Рассчитывает среднее значение указанного поля для всех документов коллекции	*db.*ИМЯ_КОЛЛЕКЦИИ.*aggregate*([{*$group: {_id : "$title", avg_salary : {$avg: "$salary"*}}}])
$min	Получает минимальное значение указанного поля документа в коллекции	*db.*ИМЯ_КОЛЛЕКЦИИ.a*ggregate*([{*$group: {_id: "$title", min_salary : {$min: "$salary"*}}}])
$max	Получает максимальное значение указанного поля документа в коллекции	*db.*ИМЯ_КОЛЛЕКЦИИ.*aggregate*([{*$group: {_id: "$title", max_salary: {$max: "$salary"*}}}])
$push	Вставляет значение в массив в результирующем документе	*db.*ИМЯ_КОЛЛЕКЦИИ.*aggregate*([{*$group: {_id: "$title",skills: {$push: "$skills"*}}}])
$addToSet	Вставляет значение в массив в результирующем документе, но не создает дубликаты	*db.*ИМЯ_КОЛЛЕКЦИИ.*aggregate*([{*$group: {_id: "$title", skills: {$addToSet: "$skills"*}}}])

Окончание табл. 3.4

Выражение	Описание	Пример
$first	Получает первый документ из сгруппированных. Обычно используется вместе с сортировкой	*db*.ИМЯ_КОЛЛЕКЦИИ.*aggregate*([{*$group: {_id: "$title", first_skill: {$first: "$skills"}}}])*
$last	Получает крайний документ из сгруппированных. Обычно используетсявместе с сортировкой	*db*.ИМЯ_КОЛЛЕКЦИИ.*aggregate*([{*$group : {_id: "$title", last_skill: {$last: "$skills"*}}}])

Технология *MapReduce*

MongoDB также предоставляет реализацию операции *MapReduce* для выполнения агрегации. Следует отметить, что эта реализация носит внешний характер, поскольку использует пользовательские функции *JavaScript* для выполнения операций, а не собственно механизмы СУБД. Конвейер агрегации обеспечивает лучшую производительность и более согласованный интерфейс, чем *MapReduce*, а различные операции *MapReduce* могут быть переписаны с использованием операторов конвейера агрегации.

В общем случае *MapReduce* – это подход к обработке данных, который имеет два преимущества по сравнению с традиционными решениями. Главное преимущество – это производительность, так как теоретически *MapReduce* может быть распараллелен, что позволяет обрабатывать огромные массивы данных на множестве ядер/-процессоров/машин. Однако, как было изложено выше, это пока не является преимуществом *MongoDB*, так как реализация *MapReduce* в *MongoDB* основана на *JavaScript*, который сам по себе однопоточен. Вторым преимуществом *MapReduce* является возможность описывать обработку данных с использованием традиционных конструкций языков программирования. По сравнению с тем, что можно сделать с помощью *SQL*, возможности кода внутри *MapReduce* намного богаче: они позволяют расширить рамки возможного даже без использования специализированных решений.

Map Reduction – это парадигма обработки данных для объединения больших объемов данных в полезные агрегированные результаты. Для выполнения этих операций *MongoDB* предоставляет команду базы данных *MapReduce*.

db.collection.mapReduce() имеет следующий синтаксис:

```
db.collection.mapReduce(
                    <map>,
                    <reduce>,
                    {
                     out: <collection>,
                     query: <document>,
                     sort: <document>,
                     limit: <number>,
                     finalize: <function>,
                     scope: <document>,
                     jsMode: <boolean>,
                     verbose: <boolean>,
                     bypassDocumentValidation: <boolean>
                    }
                   )
```

Параметры метода:

• *map* – функция *JavaScript*, которая связывает или «сопоставляет» (*map*) значение с ключом и генерирует пару «ключ–значение»;

• *reduce* – функция *JavaScript*, которая «сводит» (*reduce*) к одному объекту все значения, связанные с определенным ключом;

• *options* – документ, определяющий дополнительные параметры для *db.collection.mapReduce*(). С помощью этого параметра можно сортировать, фильтровать или ограничивать анализируемые данные, а также передавать метод *finalize*, который будет применен к результату, возвращенному этапом *reduce*.

MapReduce – процесс двухступенчатый. Сначала выполняется этап *map* (отображение), затем – *reduce* (свертка). На этапе отображения входные документы трансформируются (*map*) и производят (*emit*) пары «ключ–значение» (как ключ, так и значение могут быть составными). При свертке (*reduce*) на входе появляется ключ и массив значений, порожденных для этого ключа, а на выходе появляется финальный результат.

Все функции *MapReduce* в *MongoDB* реализуются с использованием *JavaScript* и выполняются в процессе *mongod*. Операции *MapReduce* принимают документы одной коллекции в качестве входных данных и выполняют любую произвольную сортировку и ограничение перед началом этапа *map*. *MapReduce* может возвращать ре-

зультаты операции в качестве документа или может записывать результаты в коллекции.

В *MongoDB* операции *MapReduce* используют пользовательские функции *JavaScript* для сопоставления или связывания значений с ключом. Если ключ имеет несколько значений, сопоставленных с ним, операция уменьшает значения для ключа до одного объекта.

Рассмотрим следующую операцию (рис. 3.6).

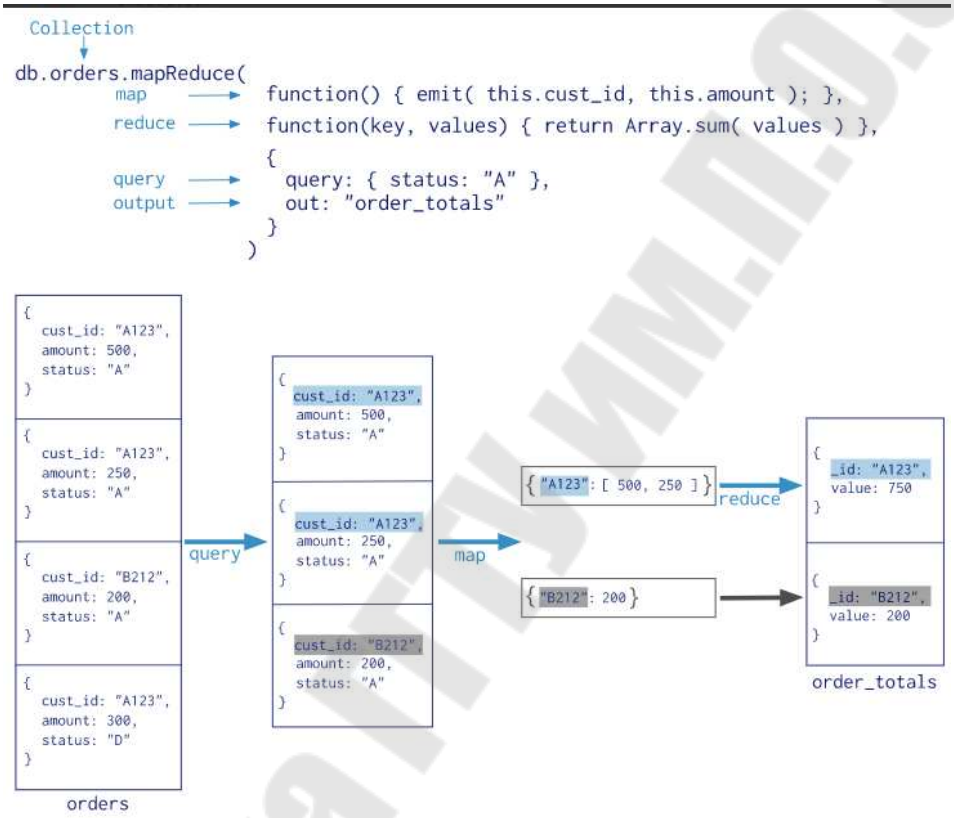

Рис. 3.6. Процесс *MapReduce MongoDB*

В операции *MapReduce MongoDB* применяет фазу *map* к каждому входному документу (то есть к документам в коллекции, которые соответствуют условию запроса). Функция *map* производит (*emit*) пары «ключ–значение». Для тех ключей, которые имеют несколько значений, *MongoDB* применяет фазу *reduce*, которая собирает и уплотняет агрегированные данные, а затем сохраняет результаты в коллекции. Выход функции *reduce* может осуществляться через функцию финализации для дальнейшего сжатия или обработки результатов агрегации.

Использование пользовательских функций *JavaScript* обеспечивает гибкость при отображении операций сокращения. Например, при обработке документа функция *map* может создавать более одного со-

поставления ключа и значения или не отображать сопоставление. Операции *reduce* также могут использовать пользовательскую функцию *JavaScript* для внесения окончательных изменений в результаты, таких как выполнение дополнительных вычислений.

Операция *MapReduce* может записывать результаты в коллекцию или возвращать результаты в строке. Если осуществляется запись в коллекцию, можно выполнять последующие операции *MapReduce* в той же коллекции входов, которые объединяют замену, объединение или уменьшение новых результатов с предыдущими результатами.

При возврате результатов оперативной операции уменьшения карты документы должны быть в пределах предельного размера документа *BSON*, который в настоящее время составляет 16 мегабайт.

Рассмотрим несколько примеров.

Пример 1

Для дальнейшего использования будем применять коллекцию *hits*, в которую вставим следующий набор данных:

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

Запишем создание функций ***map*** и ***reduce***, используя конструкции языка *JavaScript*:

```
var map = function() {
var key = {resource: this.resource, year: this.date.getFullYear(),
month: this.date.getMonth(), day: this.date.getDate()};
emit(key, {count: 1});
};
var reduce = function(key, values) {
var sum = 0;
values.forEach(function(value) {
```

```
sum += value['count'];
});
return {count: sum};
};
```

В оболочке *mongo* метод *db.collection.mapReduce*() является оболочкой для команды *MapReduce*. В следующих примерах используется метод *db.collection.mapReduce*(). Применительно к нашему примеру выполним команду *MapReduce* над коллекцией *hits*:

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

Установив *out* в *inline*, мы указываем, что *MapReduce* должен непосредственно вернуть результат в консоль. Вместо этого мы могли бы написать {*out*: *'hit_stats'*}, и результат был бы сохранен в коллекцию *hit_stats*:

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

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

Пример 2

Пусть мы имеем коллекцию *orders* некоторых заказов со следующими документами:

```
db.orders.insertMany([
   { _id: 1, cust_id: "Ant O. Knee", ord_date: new Date("2020-03-01"), price: 25, items: [ { sku: "oranges", qty: 5, price: 2.5 }, { sku: "apples", qty: 5, price: 2.5 } ], status: "A" },
   { _id: 2, cust_id: "Ant O. Knee", ord_date: new Date("2020-03-08"), price: 70, items: [ { sku: "oranges", qty: 8, price: 2.5 }, { sku: "chocolates", qty: 5, price: 10 } ], status: "A" },
   { _id: 3, cust_id: "Busby Bee", ord_date: new Date("2020-03-08"), price: 50, items: [ { sku: "oranges", qty: 10, price: 2.5 }, { sku: "pears",
```

qty: 10, price: 2.5 }], status: "A" },
{ _id: 4, cust_id: "Busby Bee", ord_date: new Date("2020-03-18"), price: 25, items: [{ sku: "oranges", qty: 10, price: 2.5 }], status: "A" },
{ _id: 5, cust_id: "Busby Bee", ord_date: new Date("2020-03-19"), price: 50, items: [{ sku: "chocolates", qty: 5, price: 10 }], status: "A"},
{ _id: 6, cust_id: "Cam Elot", ord_date: new Date("2020-03-19"), price: 35, items: [{ sku: "carrots", qty: 10, price: 1.0 }, { sku: "apples", qty: 10, price: 2.5 }], status: "A" },
{ _id: 7, cust_id: "Cam Elot", ord_date: new Date("2020-03-20"), price: 25, items: [{ sku: "oranges", qty: 10, price: 2.5 }], status: "A" },
{ _id: 8, cust_id: "Don Quis", ord_date: new Date("2020-03-20"), price: 75, items: [{ sku: "chocolates", qty: 5, price: 10 }, { sku: "apples", qty: 10, price: 2.5 }], status: "A" },
{ _id: 9, cust_id: "Don Quis", ord_date: new Date("2020-03-20"), price: 55, items: [{ sku: "carrots", qty: 5, price: 1.0 }, { sku: "apples", qty: 10, price: 2.5 }, { sku: "oranges", qty: 10, price: 2.5 }], status: "A" },
{ _id: 10, cust_id: "Don Quis", ord_date: new Date("2020-03-23"), price: 25, items: [{ sku: "oranges", qty: 10, price: 2.5 }], status: "A" }
])

Нам необходимо вернуть общую цену на клиента. Для этого выполним операцию *MapReduce* для сбора заказов для группировки по *cust_id* и вычислим сумму цены для каждого *cust_id*.

1. Определим переменную *mapFunction*1 для использования в дальнейшем как функцию *map* для обработки каждого входного документа:

```
var mapFunction1 = function() {
  emit(this.cust_id, this.price);
};
```

В функции *this* относится к документу, который обрабатывается операцией *MapReduce*.

Функция сопоставляет цену *cust_id* для каждого документа и выдает пару *cust_id* и *price*.

2. Определим переменную *reduceFunction1* для использования в дальнейшем в качестве функции *reduce* с двумя аргументами *keyCustId* и *valuesPrices*:

```
var reduceFunction1 = function(keyCustId, valuesPrices) {
```

```
  return Array.sum(valuesPrices);
};
```

valuesPrices – это массив, элементами которого являются значения цен, выдаваемые функцией карты и сгруппированные по *keyCustId*.

Функция уменьшает массив *valuesPrice* до суммы его элементов.

3. Выполним *MapReduce* операцию для всех документов в коллекции заказов с помощью ранее созданных функций: *map – mapFunction1* и *reduce – reduceFunction1*:

```
db.orders.mapReduce(
  mapFunction1,
  reduceFunction1,
  { out: "map_reduce_example" }
)
```

Эта операция выводит результаты в коллекцию с именем *map_reduce_example*. Если коллекция *map_reduce_example* уже существует, операция заменит содержимое результатами этой операции уменьшения карты.

4. Запросим коллекцию *map_reduce_example*, чтобы проверить результаты:

```
db.map_reduce_example.find().sort( { _id: 1 } )
```

Операция возвращает следующие документы:

```
{ "_id" : "Ant O. Knee", "value" : 95 }
{ "_id" : "Busby Bee", "value" : 125 }
{ "_id" : "Cam Elot", "value" : 60 }
{ "_id" : "Don Quis", "value" : 155 }
```

Описанный выше набор операций можно выполнить эквивалентным способом, используя доступные операторы конвейера агрегации, без определения пользовательских функций:

```
db.orders.aggregate([
  { $group: { _id: "$cust_id", value: { $sum: "$price" } } },
```

```
  { $out: "agg_alternative_1" }
])
```

Этап $*group* осуществляет группировку по полю *cust_id* и вычисляет поле значения ($*sum*). Поле значения содержит общую цену для каждого *cust_id*. Этот этап передает на следующий этап ($*out*) сформированные им документы:

```
{ "_id" : "Don Quis", "value" : 155 }
{ "_id" : "Ant O. Knee", "value" : 95 }
{ "_id" : "Cam Elot", "value" : 60 }
{ "_id" : "Busby Bee", "value" : 125 }
```

Затем $*out* записывает вывод в коллекцию *agg_alternative*_1. В качестве альтернативы вы можете использовать $*merge* вместо $*out*.

Выполним запрос к коллекции *agg_alternative*_1, чтобы проверить результаты:

```
db.agg_alternative_1.find().sort( { _id: 1 } )
```

Операция вернет следующие документы:

```
{ "_id" : "Ant O. Knee", "value" : 95 }
{ "_id" : "Busby Bee", "value" : 125 }
{ "_id" : "Cam Elot", "value" : 60 }
{ "_id" : "Don Quis", "value" : 155 }
```

Пример 3

Рассмотрим более сложный пример, который позволит рассчитать заказ и общее количество со средним количеством на единицу для созданной выше коллекции *orders*.

Для такого расчета необходимо выполнить *MapReduce* операцию для всех документов, у которых значение *ord_date* больше или равно 2020-03-01. Эта операция группирует заказы по полю *item.sku* и вычисляет количество заказов и общее количество заказанных товаров для каждой единицы товара. Затем операция вычисляет среднее количество на заказ для каждого значения *sku* и объединяет результаты в коллекцию выходных данных. При объединении результатов, если существующий документ имеет тот же ключ, что и новый результат, операция перезаписывает существующий документ. Если нет су-

ществующего документа с таким же ключом, операция вставляет документ.

Стадии обработки будут следующими:

1. Определим функцию *map* для обработки каждого входного документа:

```
var mapFunction2 = function () {
  for (var idx = 0; idx < this.items.length; idx++) {
    var key = this.items[idx].sku;
    var value = { count: 1, qty: this.items[idx].qty };

    emit(key, value);
  }
};
```

В функции *this* относится к документу, который обрабатывается *MapReduce* операцией. Для каждого элемента функция связывает *sku* с новым объектом *value*, который содержит *count* 1 и элемент *qty* для заказа, и выдает *sku* и *value* пару.

2. Определим соответствующую функцию *reduce* с двумя аргументами *keySKU* и *countObjVals*:

```
var reduceFunction2 = function (keySKU, countObjVals) {
  reducedVal = { count: 0, qty: 0 };

  for (var idx = 0; idx < countObjVals.length; idx++) {
    reducedVal.count += countObjVals[idx].count;
    reducedVal.qty += countObjVals[idx].qty;
  }
  return reducedVal;
};
```

CountObjVals – это массив, элементы которого являются объектами, сопоставленными со сгруппированными значениями *keySKU*, переданными *map* функцией функции *reduce*.

Функция сокращает массив *countObjVals* до одного объекта *ReducedValue*, который содержит поля *count* и *qty*. В *ReducedVal* поле *count* содержит сумму полей *count* из отдельных элементов массива, а поле *qty* содержит сумму полей *qty* из отдельных элементов массива.

3. Определим функцию *finalize* с двумя аргументами *key* и *reducedVal*:

```
var finalizeFunction2 = function (key, reducedVal) {
  reducedVal.avg = reducedVal.qty/reducedVal.count;
  return reducedVal;
};
```

Функция изменяет объект *reducedVal*, добавляя вычисляемое поле с именем *avg*, и возвращает измененный объект.

4. Собственно выполним *MapReduce* операцию для коллекции *orders* с помощью функций *mapFunction*2, *reduceFunction*2 и *finalizeFunction*2:

```
db.orders.mapReduce(
   mapFunction2,
   reduceFunction2,
   {
    out: { merge: "map_reduce_example2" },
    query: { ord_date: { $gte: new Date("2020-03-01") } },
    finalize: finalizeFunction2
   }
 );
```

Эта операция использует поле запроса для выбора только тех документов, у которых *ord_date* больше или равно новой дате (2020-03-01). Затем он выводит результаты в коллекцию *map_redu-ce_example*2.

Если коллекция *map_reduce_example*2 уже существует, операция объединит существующее содержимое с результатами этой операции уменьшения карты. То есть, если существующий документ имеет тот же ключ, что и новый результат, операция перезаписывает существующий документ. Если нет существующего документа с таким же ключом, операция вставляет документ.

5. Запросим коллекцию *map_reduce_example*2, чтобы проверить результаты:

```
db.map_reduce_example2.find().sort( { _id: 1 } )
```

Эта операция вернет следующие документы:

{ "_id" : "apples", "value" : { "count" : 3, "qty" : 30, "avg" : 10 } }
{ "_id" : "carrots", "value" : { "count" : 2, "qty" : 15, "avg" : 7.5 } }
{ "_id" : "chocolates", "value" : { "count" : 3, "qty" : 15, "avg" : 5 } }
{ "_id" : "oranges", "value" : { "count" : 6, "qty" : 58, "avg" : 9.666666666666666 } }
{ "_id" : "pears", "value" : { "count" : 1, "qty" : 10, "avg" : 10 } }

Описанный выше набор операций можно выполнить эквивалентным способом, используя доступные операторы конвейера агрегации:

```
db.orders.aggregate( [
   { $match: { ord_date: { $gte: new Date("2020-03-01") } } },
   { $unwind: "$items" },
   { $group: { _id: "$items.sku", qty: { $sum: "$items.qty" }, orders_ids: { $addToSet: "$_id" } } },
   { $project: { value: { count: { $size: "$orders_ids" }, qty: "$qty", avg: { $divide: [ "$qty", { $size: "$orders_ids" } ] } } } },
   { $merge: { into: "agg_alternative_3", on: "_id", whenMatched: "replace", whenNotMatched: "insert" } }
] )
```

Разберем последовательность выполнения этапов этого конвейера.

На этапе $*match* выбираются только те документы, для которых *ord_date* больше или равно *new Date* (2020-03-01).

Этап *$unwinds* разбивает документ по полю массива *items*, чтобы вывести документ для каждого элемента массива. Например:

{ "_id" : 1, "cust_id" : "Ant O. Knee", "ord_date" : ISODate("2020-03-01T00:00:00Z"), "price" : 25, "items" : { "sku" : "oranges", "qty" : 5, "price" : 2.5 }, "status" : "A" }
{ "_id" : 1, "cust_id" : "Ant O. Knee", "ord_date" : ISODate("2020-03-01T00:00:00Z"), "price" : 25, "items" : { "sku" : "apples", "qty" : 5, *"price" : 2.5 }, "status" : "A" }*
{ "_id" : 2, "cust_id" : "Ant O. Knee", "ord_date" : ISODate("2020-03-08T00:00:00Z"), "price" : 70, "items" : { "sku" : "oranges", "qty" : 8, "price" : 2.5 }, "status" : "A" }
{ "_id" : 2, "cust_id" : "Ant O. Knee", "ord_date" : ISODate("2020-03-08T00:00:00Z"), "price" : 70, "items" : { "sku" : "chocolates", "qty" : 5, "price" : 10 }, "status" : "A" }

{ "_id" : 3, "cust_id" : "Busby Bee", "ord_date" : ISODate("2020-03-08T00:00:00Z"), "price" : 50, "items" : { "sku" : "oranges", "qty" : 10, "price" : 2.5 }, "status" : "A" }

{ "_id" : 3, "cust_id" : "Busby Bee", "ord_date" : ISODate("2020-03-08T00:00:00Z"), "price" : 50, "items" : { "sku" : "pears", "qty" : 10, "price" : 2.5 }, "status" : "A" }

{ "_id" : 4, "cust_id" : "Busby Bee", "ord_date" : ISODate("2020-03-18T00:00:00Z"), "price" : 25, "items" : { "sku" : "oranges", "qty" : 10, "price" : 2.5 }, "status" : "A" }

{ "_id" : 5, "cust_id" : "Busby Bee", "ord_date" : ISODate("2020-03-19T00:00:00Z"), "price" : 50, "items" : { "sku" : "chocolates", "qty" : 5, "price" : 10 }, "status" : "A" }

…

Этап *$group* формирует группы по *items.sku*, рассчитывая для каждого *sku* поле *qty* и массив *orders_ids*. Поле *qty* содержит общее количество заказанных товаров для каждого *items.sku* (*$addToSet*).

Массив *orders_ids* содержит набор отдельных идентификаторов *order _id* для *items.sku* (*$addToSet*):

{ "_id" : "chocolates", "qty" : 15, "orders_ids" : [*2*, *5*, *8*] }
{ "_id" : "oranges", "qty" : 63, "orders_ids" : [*4*, *7*, *3*, *2*, *9*, *1*, *10*] }
{ "_id" : "carrots", "qty" : 15, "orders_ids" : [*6*, *9*] }
{ "_id" : "apples", "qty" : 35, "orders_ids" : [*9*, *8*, *1*, *6*] }
{ "_id" : "pears", "qty" : 10, "orders_ids" : [*3*] }

Этап *$project* изменяет форму выходного документа, чтобы он имел два поля: *_id* и *value*. На этом этапе устанавливается поле *value.count*, равное размеру массива *orders_ids* (*$size*); поле *value.qty*, равное полю *qty* входного документа; поле *value.avg*, равное среднему количеству в заказе:

{ "_id" : "apples", "value" : { "count" : 4, "qty" : 35, "avg" : 8.75 } }
{ "_id" : "pears", "value" : { "count" : 1, "qty" : 10, "avg" : 10 } }
{ "_id" : "chocolates", "value" : { "count" : 3, "qty" : 15, "avg" : 5 } }
{ "_id" : "oranges", "value" : { "count" : 7, "qty" : 63, "avg" : 9 } }
{ "_id" : "carrots", "value" : { "count" : 2, "qty" : 15, "avg" : 7.5 } }

Этап $*merge* записывает вывод в коллекцию *agg_alternative*_3. Если существующий документ имеет тот же ключ *_id*, что и новый результат, операция перезаписывает существующий документ. Если нет существующего документа с таким же ключом, операция вставляет документ.

Выполним запрос к коллекции *agg_alternative*_3, чтобы проверить результаты:

```
db.agg_alternative_3.find().sort( { _id: 1 } )
```

Операция возвращает следующие документы:

```
{ "_id" : "apples", "value" : { "count" : 4, "qty" : 35, "avg" : 8.75 } }
{ "_id" : "carrots", "value" : { "count" : 2, "qty" : 15, "avg" : 7.5 } }
{ "_id" : "chocolates", "value" : { "count" : 3, "qty" : 15, "avg" : 5 } }
{ "_id" : "oranges", "value" : { "count" : 7, "qty" : 63, "avg" : 9 } }
{ "_id" : "pears", "value" : { "count" : 1, "qty" : 10, "avg" : 10 } }
```

3.10. Сопоставление операторов *SQL* и *MongoDB*

Выше мы уже приводили в качестве примеров отдельные варианты сопоставления операторов языка *SQL*, используемого для работы с реляционными базами данных, и команд нереляционной СУБД *MongoDB*.

Ниже мы более подробно рассмотрим примеры соответствующих конструкций.

В примерах для языка *SQL* предполагается наличие таблицы с именем *people*, примеры *MongoDB* будут использовать коллекцию с именем *people*, которая содержит документы следующего прототипа:

```
{
  _id: ObjectId("509a8fb2f3f4948bd2f983a0"),
  user_id: "abc123",
  age: 55,
  status: 'A'
}
```

Создание и изменения объектов базы данных

В табл. 3.5 представлены различные операторы *SQL*, относя-

щиеся к действиям на уровне таблицы, и соответствующие операторы *MongoDB*.

1. Создание таблицы или коллекции.

Создание таблицы на *SQL*:

```
CREATE TABLE people (
   id MEDIUMINT NOT NULL
      AUTO_INCREMENT,
   user_id varchar(30),
   age Number,
   status char(1),
   PRIMARY KEY (id)
)
```

Создание коллекции *MongoDB* явно:

```
db.createCollection("people")
```

Неявное создание коллекции *MongoDB*:

```
db.people.insertOne( {
   user_id: "abc123",
   age: 55,
   status: "A"
 } )
```

Коллекция неявно создается при первой операции *insertOne*() или *insertMany*(). Первичный ключ *_id* добавляется автоматически, если поле *_id* не указано.

2. Изменение таблицы и коллекции.

Добавление столбца на *SQL*:

```
ALTER TABLE people ADD join_date DATETIME
```

В *MongoDB* коллекции, соответствующие таблицам реляционных баз данных, не описывают структуру своих документов, то есть на уровне коллекции не происходит структурных изменений. Однако на уровне документа операции *updateMany*() могут добавлять поля в существующие документы с помощью оператора $*set*.

```
db.people.updateMany(
    { },
    { $set: { join_date: new Date() } }
)
```

Удаление столбца на *SQL*:

```
ALTER TABLE people DROP COLUMN join_date
```

Поскольку коллекции не описывают структуру своих документов, то никаких аналогичных операций в *MongoDB* нет. Однако на уровне документа операции *updateMany*() могут удалять поля из документов с помощью оператора $*unset*.

```
db.people.updateMany(
    { },
    { $unset: { "join_date": "" } }
)
```

Создание индексов в реляционной таблице на *SQL*:

```
CREATE INDEX idx_user_id_asc ON people(user_id)
CREATE INDEX idx_user_id_asc_age_desc ON people(user_id, age DESC)
```

Создание индексов в коллекции таблице в *MongoDB*:

```
db.people.createIndex( { user_id: 1 } )
db.people.createIndex( { user_id: 1, age: -1 } )
```

3. Удаление таблицы или коллекции.
Удаление таблицы на *SQL*:

```
DROP TABLE people
```

Удаление коллекции в *MongoDB*:

```
db.people.drop()
```

Вставка (добавление) данных

Вставка данных в таблицу:

INSERT INTO people (*user_id, age, status*) *VALUES* ("*bcd*001", 45, "*A*")

Вставка данных в коллекцию:

db.people.insertOne({ *user_id*: "*bcd*001", *age*: 45, *status*: "*A*" })

Выборка данных

В табл. 3.5 представлены различные операторы *SQL SELECT*, связанные с чтением записей из реляционной таблицы *people*, и соответствующие операторы *MongoDB find()* чтения документов из коллекции *people*.

Таблица 3.5

Операторы *SQL SELECT* и *MongoDB find*()

Операторы *SQL SELECT*	**Операторы *MongoDB find*()**
*SELECT * FROM people*	*db.people.find*()
SELECT id, user_id, status FROM people	*db.people.find*({ }, { *user_id: 1, status: 1* })
SELECT user_id, status FROM people	*db.people.find*({ }, { *user_id: 1, status: 1, _id: 0* })
*SELECT * FROM people WHERE status = "A"*	*db.people.find*({ *status: "A"* })
SELECT user_id, status FROM people WHERE status = "A"	*db.people.find*({ *status: "A"* }, { *user_id: 1, status: 1, _id: 0* })
*SELECT * FROM people WHERE status != "A"*	*db.people.find*({ *status:* { *$ne: "A"* } })
*SELECT * FROM people WHERE status = "A" AND age = 50*	*db.people.find*({ *status: "A", age: 50* })
*SELECT * FROM people WHERE status = "A" OR age = 50*	*db.people.find*({ *$or:* [{ *status: "A"* } , { *age: 50* }] })
*SELECT * FROM people WHERE age > 25*	*db.people.find*({ *age:* { *$gt: 25* } })
*SELECT * FROM people WHERE age < 25*	*db.people.find*({ *age:* { *$lt: 25* } })
*SELECT * FROM people WHERE age > 25 AND age <= 50*	*db.people.find*({ *age:* { *$gt: 25, $lte: 50* } })

Окончание табл. 3.5

Операторы *SQL SELECT*	**Операторы *MongoDB find*()**
*SELECT * FROM people* *WHERE user_id like "%bc%"*	*db.people.find*({ *user_id: /bc/* }) -или- *db.people.find*({ *user_id:* { *$regex: /bc/* } })
*SELECT * FROM people* *WHERE user_id like "bc%"*	*db.people.find*({ *user_id: /^bc/* }) -или- *db.people.find*({ *user_id:* { *$regex: /^bc/* } })
*SELECT * FROM people* *WHERE status = "A"* *ORDER BY user_id ASC*	*db.people.find*({ *status: "A"* }).*sort*({ *user_id: 1* })
*SELECT * FROM people* *WHERE status = "A"* *ORDER BY user_id DESC*	*db.people.find*({ *status: "A"* }).*sort*({ *user_id: -1* })
SELECT COUNT(*) FROM people	*db.people.count*() -или- *db.people.find*().*count*()
SELECT COUNT(user_id) *FROM people*	*db.people.count*({ *user_id:* { *$exists: true* } }) -или- *db.people.find*({ *user_id:* { *$exists: true* } }).*count*()
SELECT COUNT(*) *FROM people* *WHERE age > 30*	*db.people.count*({ *age:* { *$gt: 30* } }) -или- *db.people.find*({ *age:* { *$gt: 30* } }).*count*()
SELECT DISTINCT(status) *FROM people*	*db.people.aggregate*([{ *$group :* { *_id : "$status"* } }]) -или- *db.people.distinct*(*"status"*)
*SELECT * FROM people* *LIMIT 1*	*db.people.findOne*() -или- *db.people.find*().*limit*(*1*)
*SELECT * FROM people LIMIT 5* *SKIP 10*	*db.people.find*().*limit*(*5*).*skip*(*10*)
*EXPLAIN SELECT * FROM people* *WHERE status = "A"*	*db.people*

Следует отметить, что метод *find*() всегда включает поле *_id* в возвращаемые документы, если специально это не исключено через соответствующую операцию проекции.

Обновление данных
Обновление записей в таблице с использованием *SQL*:

UPDATE people SET status = "C" WHERE age > 25
UPDATE people SET age = age + 3 WHERE status = "A"

Соответствующее обновление документов в коллекции с использованием *MongoDB*:

db.people.updateMany({ *age*: { $*gt*: 25 } }, { $*set*: { *status*: "*C*" } })
db.people.updateMany({ *status*: "*A*" }, { $*inc*: { *age*: 3 } })

Удаление данных
Удаление записей в таблице с использованием *SQL*:

DELETE FROM people WHERE status = "D"
DELETE FROM people

Соответствующее удаление документов в коллекции с использованием *MongoDB*:

db.people.deleteMany({ *status*: "*D*" })
db.people.deleteMany({})

3.11. Транзакции

В *MongoDB* операция над отдельным документом коллекции является атомарной. Поскольку существует возможность использовать встроенные документы и массивы для фиксации взаимосвязей между данными в единой структуре документа вместо применения нормализации для нескольких документов и коллекций, такая атомарность одного документа устраняет необходимость в транзакциях с несколькими документами для многих практических случаев использования.

В ситуациях, когда требуется атомарность чтения и записи в нескольких документах (в одной или нескольких коллекциях), *MongoDB* поддерживает транзакции с несколькими документами. С распределенными транзакциями транзакции могут использоваться в нескольких операциях, коллекциях, базах данных, документах и сегментах (шардах).

Для транзакций доступны следующие методы оболочки *mongo*: *Session.startTransaction*(), *Session.commitTransaction*(), *Session.abort-Transaction*(). Проиллюстрируем их использование на примере:

```
// создаем коллекции:
db.getSiblingDB("mydb1").foo.insert( {abc: 0}, { writeConcern: { w: "majority", wtimeout: 2000 } } );
db.getSiblingDB("mydb2").bar.insert( {xyz: 0}, { writeConcern: { w: "majority", wtimeout: 2000 } } );
// запускаем сеанс:
session = db.getMongo().startSession( { readPreference: { mode: "primary" } } );
coll1 = session.getDatabase("mydb1").foo;
coll2 = session.getDatabase("mydb2").bar;
// запускаем транзакцию:
session.startTransaction( { readConcern: { level: "local" }, write-Concern: { w: "majority" } } );
// операции внутри транзакции:
try {
  coll1.insertOne( { abc: 1 } );
  coll2.insertOne( { xyz: 999 } );
} catch (error) {
  // прекращение транзакции при ошибке:
  session.abortTransaction();
  throw error;
}
// фиксируем транзакцию, используя проблему записи, установленную в начале транзакции:
session.commitTransaction();
session.endSession();
```

В примере ниже используется код на языке *C# API* обратного вызова для работы с транзакциями, этот код запускает транзакцию, выполняет указанные операции и фиксируется (или прерывается в случае ошибки). *API* также включает логику повтора для ошибок фиксации *TransientTransactionError* или *UnknownTransactionCommit-Result*:

```
var client = new MongoClient(connectionString);
// предварительное создание коллекций:
```

```
var database1 = client.GetDatabase("mydb1");
var collection1 = database1.GetCollection<BsonDocument>("foo").
WithWriteConcern(WriteConcern.WMajority);
collection1.InsertOne(new BsonDocument("abc", 0));
var database2 = client.GetDatabase("mydb2");
var collection2 = database2.GetCollection<BsonDocument>("bar").
WithWriteConcern(WriteConcern.WMajority);
collection2.InsertOne(new BsonDocument("xyz", 0));
// запуск клиентского сеанса:
using (var session = client.StartSession())
{
   // определите параметры для использования в транзакции:
   var transactionOptions = new TransactionOptions(
      readPreference: ReadPreference.Primary,
      readConcern: ReadConcern.Local,
      writeConcern: WriteConcern.WMajority);
   // определите последовательность операций для выполнения внутри транзакций:
   var cancellationToken = CancellationToken.None; // normally a real token would be used
   result = session.WithTransaction(
      (s, ct) =>
      {
         collection1.InsertOne(s, new BsonDocument("abc", 1), cancellationToken: ct);
         collection2.InsertOne(s, new BsonDocument("xyz", 999), cancellationToken: ct);
         return "Inserted into collections in different databases";
      },
      transactionOptions,
      cancellationToken);
}
```

Распределенные транзакции относятся к транзакциям с несколькими документами в сегментированных кластерах и наборах реплик. Многодокументные транзакции (будь то сегментированные кластеры или наборы реплик) также известны как распределенные транзакции.

В ситуациях, когда требуются атомарность чтения и запись в нескольких документах (в одной или нескольких коллекциях), *MongoDB* последних версий поддерживает многодокументные транзакции.

Многодокументные транзакции являются атомарными, предусматривающими реализацию принципа «все или ничего»:

1) когда транзакция фиксируется, все изменения данных, внесенные в транзакцию, сохраняются и отображаются вне транзакции, то есть транзакция не будет фиксировать одни свои изменения при откате других. Пока транзакция не зафиксирована, изменения данных, внесенные в транзакцию, не видны за пределами транзакции. Однако когда транзакция записывает данные в несколько сегментов (шардов), не все внешние операции чтения должны ждать, пока результат зафиксированной транзакции станет видимым в шардах. Например, если транзакция зафиксирована и запись 1 видна на шарде *A*, но запись 2 еще не видна на шарде *B*, внешнее чтение с проблемой чтения «локально» может читать результаты записи 1, не видя записи 2;

2) когда транзакция прерывается, все изменения данных, внесенные в транзакцию, отменяются, не становясь видимыми. Например, если какая-либо операция в транзакции завершается неудачно, транзакция прерывается, и все изменения данных, сделанные в транзакции, отменяются, не становясь видимыми.

Следует отметить, что в большинстве случаев транзакция с несколькими документами требует более высоких затрат производительности по сравнению с записью одного документа, и доступность транзакций с несколькими документами не должна заменять эффективный дизайн схемы. Для многих сценариев денормализованная модель данных (встроенные документы и массивы) по-прежнему будет оптимальной. То есть для многих сценариев правильное моделирование данных сводит к минимуму необходимость в многодокументных транзакциях.

3.12. Индексы

Индексы – это специальные структуры данных, которые хранят небольшую часть набора данных коллекции в удобной для просмотра и поиска форме. Индекс хранит значение определенного поля или набора полей, упорядоченных по значению. Упорядочение записей индекса поддерживает поиск эффективных совпадений для операций запросов на равенство значению и вхождению значения в диапазон.

Кроме того, *MongoDB* может возвращать отсортированные результаты, используя порядок в индексе.

Для реализации индексов *MongoDB* использует структуру данных *B*-дерево («сбалансированное дерево»). Индекс *B*-дерева имеет иерархическую древовидную структуру. В верхней части дерева находится блок заголовка. Этот блок содержит указатели на соответствующий блок ветвления для любого заданного диапазона значений ключа. Блок ветвления обычно указывает на соответствующий конечный блок для более конкретного диапазона или (для большего индекса) указывает на другой блок ветвления. Листовой блок содержит список значений ключей и указателей на расположение документов на диске.

Пример такой структуры изображен на рис. 3.7.

MongoDB будет обходить этот индекс следующим образом. Если нужно получить доступ к некоторой записи, например, *BAKER*, сначала считывается блок заголовка. Это блок указывает, что значения ключей, начинающиеся с *A* по *K*, хранятся в левом блоке ветви. Происходит считывание советующего блока ветвления. Этом блок ветвления, указывает, что значения ключей, начинающиеся с *A* по *D*, хранятся в крайнем левом листовом блоке. Просматривая этот листовой блок, мы находим значение *BAKER* и связанное с ним расположение на диске, которое затем будем использовать для доступа к соответствующему документу.

Блоки листьев содержат ссылки как на предыдущий, так и на следующий блок. Это позволяет сканировать индекс в порядке возрастания или убывания, а также позволяет обрабатывать запросы диапазона с использованием операторов *$gt* или *$lt* с использованием индекса.

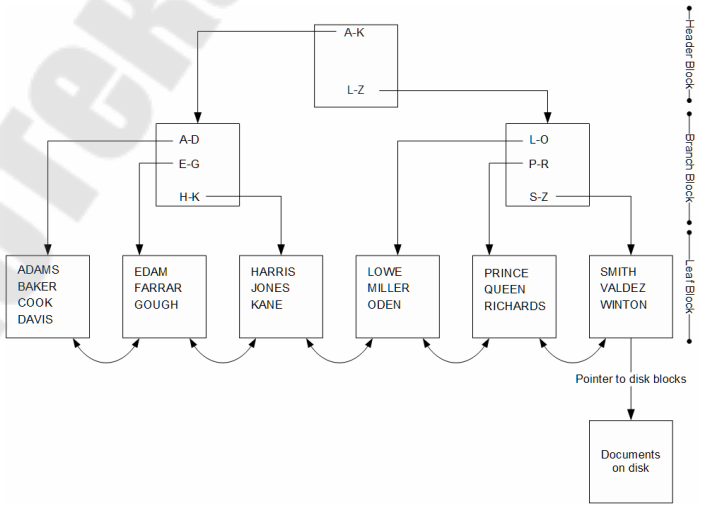

Рис. 3.7. Структура данных *B*-дерево

Индексы *B*-дерева имеют следующие преимущества перед другими стратегиями индексирования.

Поскольку каждый листовой узел находится на одной глубине, производительность очень предсказуема. Теоретически ни один документ в коллекции не будет удален более чем на три или четыре ввода-вывода.

B-деревья обеспечивают хорошую производительность для больших коллекций, поскольку глубина не превышает четырех блоков (один блок заголовка, два уровня блоков ветвления и один уровень конечного блока). Как правило, для поиска любого документа не требуется более четырех операций ввода-вывода. Поскольку блок заголовка почти всегда уже загружен в память, а блоки ветвления обычно загружаются в память, фактическое количество операций чтения с физического диска обычно составляет всего одно или два.

Индекс *B*-дерева поддерживает запросы диапазона, а также точный поиск. Это возможно благодаря ссылкам на предыдущий и следующий листовые блоки.

Индекс *B*-дерева обеспечивает гибкое и эффективное выполнение запросов. Однако поддержание *B*-дерева при изменении данных может быть дорогостоящим. Например, рассмотрим возможность вставки документа со значением ключа *NIVEN* в коллекцию на нашей диаграмме выше. Чтобы вставить коллекцию, мы должны добавить новую запись в блок *L–O*. Если в этом блоке есть свободное место, то стоимость будет существенной, но, возможно, не чрезмерной. Но что будет, если в блоке нет свободного места?

Если в листовом блоке для новой записи нет свободного места, требуется разделение индекса. Новый блок должен быть выделен, и половина записей в существующем блоке переместится в новый блок. Кроме того, существует требование добавить новую запись в блок ветвления (чтобы указать на вновь созданный листовой блок). Если в блоке ветвления нет свободного места, то блок ветвления также необходимо разделить. Такое разбиение индекса – дорогостоящая операция: необходимо выделить новые блоки, а записи индекса переместить из одного блока в другой.

В целом индексы поддерживают эффективное выполнение запросов в *MongoDB*. Без индексов *MongoDB* должен выполнить полное сканирование коллекции, то есть сканировать каждый документ в коллекции, чтобы выбрать те документы, которые соответствуют содержанию запроса. Если для коллекции существует соответствующий индекс, *MongoDB* может использовать этот индекс, чтобы ограничить

количество документов, которые должны быть проверены для выполнения запроса.

На рис. 3.8 показан запрос, который из коллекции *users* выбирает (*find*) и упорядочивает (*sort*) совпадающие документы. Для данной коллекции определен индекс по полю *score* ({*score*: 1}).

MongoDB определяет индексы на уровне коллекции и поддерживает индексы для любого поля или подполя документов в заданной коллекции.

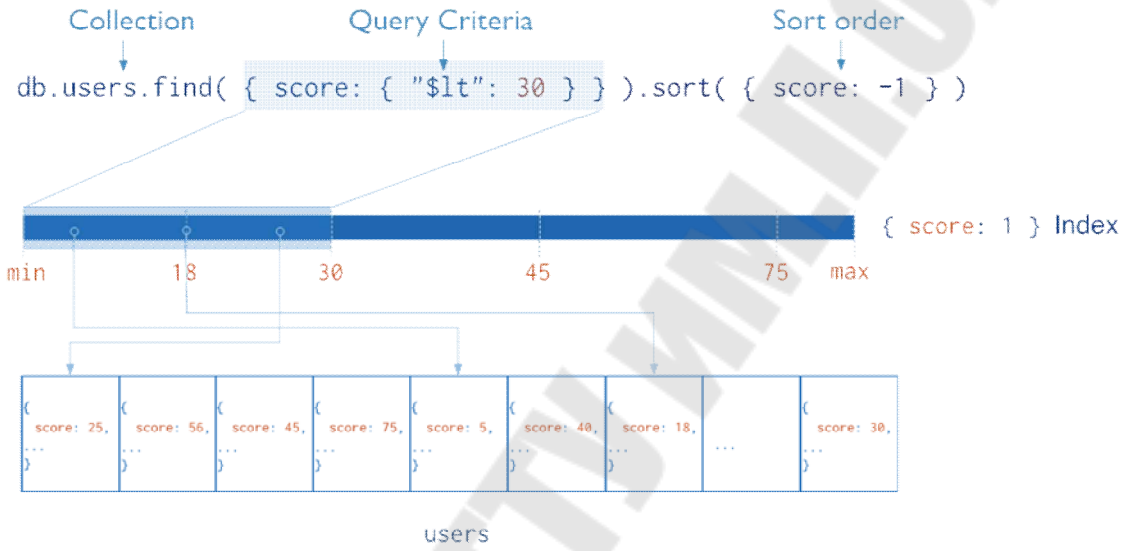

Рис. 3.8. Запрос индекса по полю *score*

Индекс по полю *_id*

MongoDB по умолчанию создает уникальный индекс для поля *_id* во время создания любой коллекции. Индекс *_id* не позволяет клиентам вставлять два документа с одинаковым значением поля *_id*. Удалить этот индекс по полю *_id* невозможно.

Следует отметить, что в сегментированных кластерах, если не используется поле *_id* в качестве ключа сегмента, внутренняя логика приложения, работающего с базой данных, должна гарантировать уникальность значений в поле *_id*, чтобы предотвратить ошибки. Чаще всего это делается с помощью стандартного поля типа *ObjectId*, которое автоматически генерирует необходимые уникальные значения.

Создание индекса

Чтобы создать индекс в *Mongo Shell*, используют метод *db.collection.createIndex*(*<key and index type specification>*, *<options>*), где *<key and index type specification>* – документ, содержащий пары

поля и значения, в которых поле является ключом индекса, а значение описывает тип индекса для этого поля; для индекса по возрастанию в поле указывается значение 1, для индекса по убыванию указывается значение -1; <*options*> – не обязательный параметр, представляющий собой документ, содержащий набор параметров, управляющих созданием индекса.

В следующем примере создается простой ключевой нисходящий индекс по полю *name*:

```
db.collection.createIndex( { name: -1 } )
```

Метод *db.collection.createIndex* создает индекс только в том случае, если индекс с той же спецификацией еще не существует.

Имена индексов

Имя индекса по умолчанию – это объединение проиндексированных ключей и направления каждого ключа в индексе (например, 1 или -1) с использованием подчеркивания в качестве разделителя.

Например, индекс, созданный для {*item*: 1, *quantity*: -1}, имеет имя *item*_1_*quantity*_-1.

Существует возможность создавать индексы с явно указываемым именем, чтобы сделать их чтение более удобным.

Например, рассмотрим приложение, которое часто запрашивает коллекцию *products*, чтобы заполнить данные о существующих запасах. Следующий метод *createIndex*() создает индекс на основе полей *item* и *quantity* с именем *query for catalogue*:

```
db.products.createIndex(
  { item: 1, quantity: -1 } ,
  { name: "query for catalogue" }
)
```

Просмотреть имена уже созданных индексов для коллекции можно с помощью метода *db.collection.getIndexes*().

Следует отметить, что нельзя переименовать уже созданный индекс. Если такая необходимость, все же, существует, нужно удалить старый индекс и создать новый с другим именем.

Типы индексов

MongoDB предоставляет ряд различных типов индексов для поддержки определенных типов данных и запросов.

Простой индекс – индекс по возрастанию/убыванию для одного поля документа коллекции.

MongoDB обеспечивает полную поддержку индексов по любому полю в коллекции документов. По умолчанию все коллекции имеют индекс в поле *_id*, а приложения и пользователи могут добавлять дополнительные индексы для поддержки важных запросов и операций.

В этом учебно-методическом пособии описываются индексы по возрастанию или убыванию для одного поля (рис. 3.9).

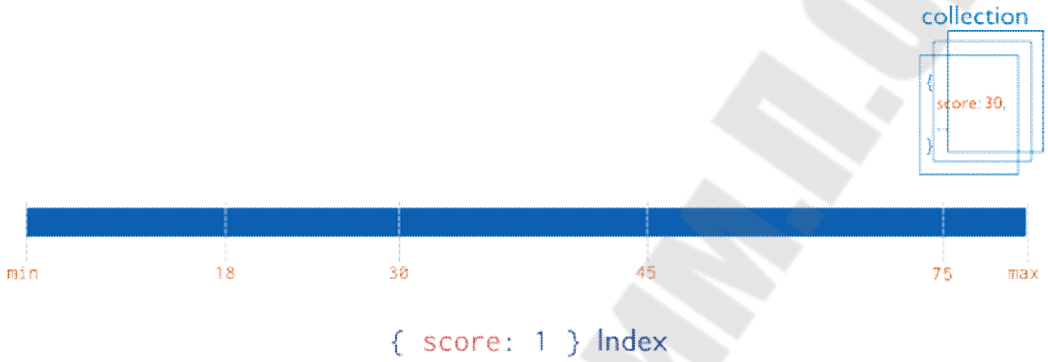

Рис. 3.9. Индексы по возрастанию или убыванию одного поля

Для операций индексации и сортировки с одним полем порядок сортировки (по возрастанию или по убыванию) ключа индекса не имеет значения, поскольку *MongoDB* может перемещаться по индексу в любом направлении.

Составной индекс – индекс, в котором содержатся ссылки на несколько полей в документах коллекции. *MongoDB* устанавливает максимальное ограничение в 32 поля для любого составного индекса.

На рис. 3.10 показан пример составного индекса по двум полям *userid* и *score*.

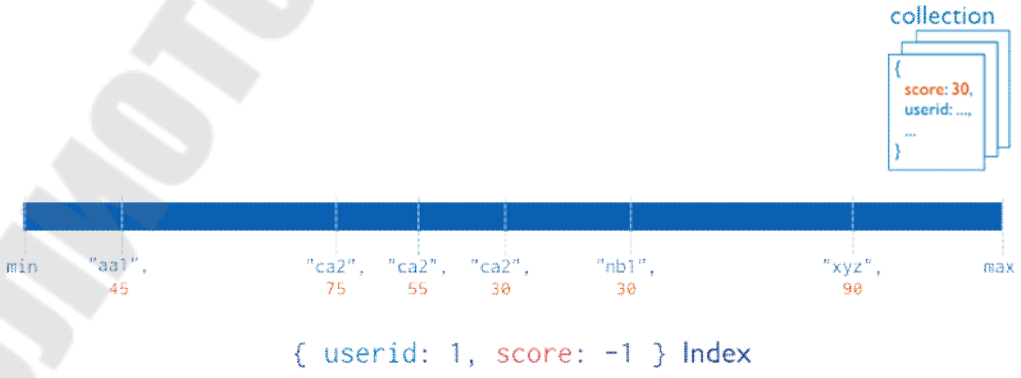

Рис. 3.10. Пример составного индекса

Порядок полей, перечисленных в составном индексе, имеет значение. Например, если составной индекс состоит из {*userid*: 1, *score*: -1}, индекс сначала сортируется по идентификатору *userid*, а затем в пределах каждого значения идентификатора *userid* сортируется по полю *score*.

Для составных индексов и операций сортировки порядок сортировки ключей индекса определяет, может ли индекс поддерживать операцию сортировки.

Рассмотрим коллекцию *events*, которая содержит документы с полями *username* и *date*. Приложения могут выдавать запросы, которые возвращают результаты, отсортированные сначала по возрастанию значений *username*, а затем по убыванию *date*, то есть от более поздних до последних значений даты, например:

db.events.find().sort({ *username*: 1, *date*: -1 })

Или запросы, которые возвращают результаты, отсортированные сначала по убыванию значений *username*, а затем по возрастанию значений *date*, например:

db.events.find().*sort*({ *username*: -1, *date*: 1 })

Следующий индекс может поддерживать обе эти операции сортировки:

db.events.createIndex({ "*username*" : 1, "*date*" : -1 })

Указанный выше индекс не может поддерживать сортировку по возрастанию значений *username*, а затем по возрастанию значений *date*, например следующее:

db.events.find().sort({ *username*: 1, *date*: 1 })

Многоключевой индекс используется для индексации содержимого, хранящегося в массивах. Так, если индексируется поле, содержащее значение массива, *MongoDB* создает отдельные записи индекса для каждого элемента массива. Эти индексы позволяют запросам выбирать документы, содержащие массивы, путем сопоставления элемента или элементов массивов. Многоключевые индексы могут быть

построены как на массивах, содержащих скалярные значения (например, строки, числа), так и на вложенных документах.

MongoDB автоматически определяет, нужно ли создавать многоключевой индекс, если индексируемое поле содержит значение массива. Явно указывать многоключевой тип индекса не нужно.

Например, рассмотрим коллекцию *catalogue*, которая содержит следующие документы:

{ _id: 5, type: "food", item: "aaa", ratings: [*5, 8, 9*] }
{ _id: 6, type: "food", item: "bbb", ratings: [*5, 9*] }
{ _id: 7, type: "food", item: "ccc", ratings: [*9, 5, 8*] }
{ _id: 8, type: "food", item: "ddd", ratings: [*9, 5*] }
{ _id: 9, type: "food", item: "eee", ratings: [5, 9, 5] }

Создадим многоключевой индекс по полю *ratings*:

db.catalogue.createIndex({ *ratings*: 1 })

Следующий запрос найдет документы, в которых поле *ratings* представляет собой массив [5, 9]:

db.catalogue.find({ *ratings*: [5, 9] })

При выполнении запроса будет использоваться многоключевой индекс для поиска документов, которые имеют 5 в любой позиции в массиве *ratings*. Затем *MongoDB* извлекает и фильтрует документы, массив *ratings* которых равен массиву запросов [5, 9].

Геопространственный индекс используется для поддержки эффективных запросов данных геопространственных координат. *MongoDB* предоставляет два специальных индекса: 2*d*-индексы, которые используют плоскую геометрию при возврате результатов, и 2*dsphere*-индексы, которые используют сферическую геометрию для возврата результатов.

Текстовый индекс – поддерживает поиск строкового содержимого в полнотекстовой коллекции. Текстовые индексы могут включать любое поле, значение которого является строкой или массивом строковых элементов Текстовые индексы не хранят стоп-слова, зависящие от языка (например, *the*, *a*, *or*).

Хешированный индекс – индексирует хеш-значения поля. Он используется для поддержки сегментирования на основе хешей. Ин-

дексы этого типа имеют более случайное распределение значений по диапазону и поддерживают только совпадения на равенство, не могут поддерживать запросы на основе диапазона.

Свойства индексов

Уникальность требует отклонять повторяющиеся значения для индексированного поля. Помимо ограничения уникальности, уникальные индексы функционально взаимозаменяемы с другими индексами *MongoDB*.

Частичность, при которой индексируют только те документы в коллекции, которые соответствуют указанному выражению фильтра. Благодаря индексации подмножества документов в коллекции частичные индексы имеют меньшие требования к хранению и снижают затраты на создание и обслуживание индексов. Частичные индексы предлагают расширенный набор функций разреженных индексов, и им следует отдавать предпочтение перед разреженными индексами.

Разреженность (*sparse*) гарантирует, что индекс будет содержать только записи для документов, имеющих индексируемое поле. Индекс пропускает документы, не имеющие индексируемого поля.

Можно комбинировать параметр разреженного индекса с параметром уникального индекса, чтобы предотвратить вставку документов, которые имеют повторяющиеся значения для проиндексированных полей, и могут пропустить индексирование документов, в которых отсутствует индексированное поле (поля).

TTL – специальные индексы, которые *MongoDB* может использовать для автоматического удаления документов из коллекции через определенное время. Это идеально подходит для определенных типов информации, таких как данные о событиях, сгенерированные машиной, журналы и информация о сеансах, которые должны сохраняться в базе данных только в течение ограниченного времени.

Скрытость позволяет сделать так, чтобы планировщику запросов не были видны запросы и чтобы они не могли использоваться для поддержки того или иного запроса. Скрывая индекс от планировщика, пользователи могут оценить потенциальное влияние удаления индекса без его фактического удаления.

За исключением индекса *_id*, можно скрыть любые индексы.

3.13. Репликация

***Репликация* (*Replication*)** – механизм синхронизации содержимого нескольких копий объекта базы данных. Репликация подразумевает процесс копирования данных из одного источника на другой (или на множество других) и наоборот. При репликации изменения, сделанные в одной копии объекта, могут быть распространены на другие копии.

Репликация обеспечивает избыточность и увеличивает доступность данных. При наличии нескольких копий данных на разных серверах баз данных репликация обеспечивает определенный уровень отказоустойчивости от потери одного сервера базы данных.

В некоторых случаях репликация может обеспечить повышенную скорость чтения, поскольку клиенты могут отправлять операции чтения на разные серверы. Хранение копий данных в разных центрах обработки данных может повысить их локальность и доступность для распределенных приложений. Возможно поддерживать дополнительные копии для специальных целей, таких как аварийное восстановление, создание отчетов или резервное копирование.

В *MongoDB* реализованы два варианта репликации: главный–подчиненный и наборы реплик. В обоих случаях единственный первичный узел выполняет все операции записи, после чего вторичные узлы считывают описания этих операций и асинхронно применяют их у себя. Однако рекомендуемая стратегия репликации в производственных системах – это наборы реплик.

Набор реплик – это группа экземпляров *mongod*, которые поддерживают один и тот же набор данных. Набор реплик содержит несколько узлов передачи данных и, возможно, один узел-арбитр. Из узлов, несущих данные, один член считается первичным узлом, тогда как другие узлы считаются вторичными.

На рис. 3.11 представлена схема взаимодействия между экземплярами.

Первичный (*Primary*) узел получает все операции записи. Первичный сервер записывает все изменения своих наборов данных в журнал операций. Вторичные (*Secondary*) реплики реплицируют журнал операций первичного сервера и применяют операции к своим наборам данных таким образом, чтобы наборы данных вторичных серверов отражали набор данных первичного.

Если первичная реплика недоступна, подходящие вторичные реплики проведут выборы, чтобы избрать новый первичный узел.

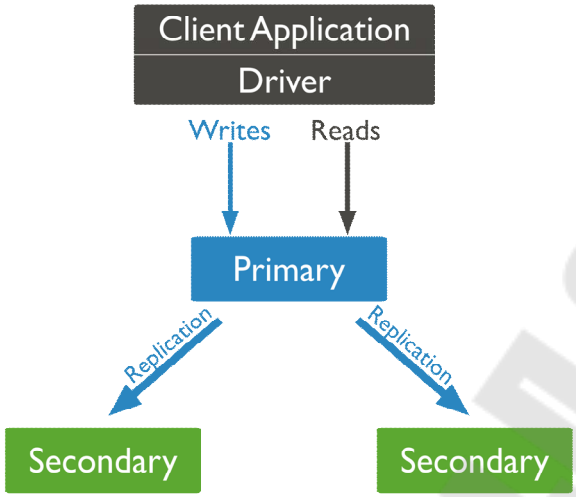

Рис. 3.11. Схема взаимодействия между экземплярами

В некоторых случаях (например, если у вас есть первичный и вторичный сервер, но ограничения по стоимости не позволяют добавлять еще один вторичный), можно добавить экземпляр *mongod* в набор реплик в качестве арбитра. Арбитр участвует в выборах, но не хранит данные (то есть не обеспечивает избыточности данных) (рис. 3.12).

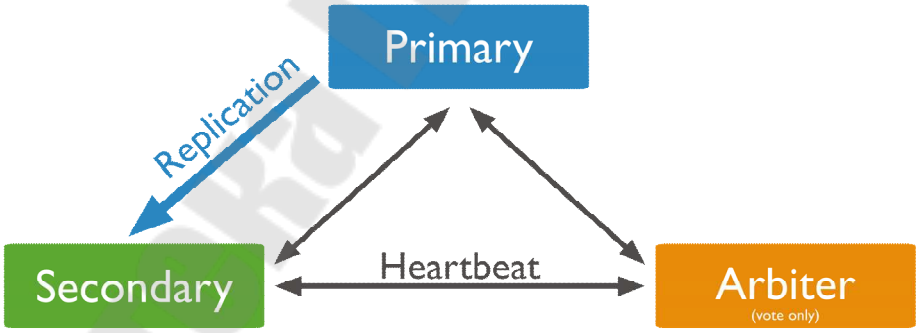

Рис. 3.12. Схема взаимодействия между экземплярами с арбитром

Репликация может быть синхронной или асинхронной. В случае синхронной репликации, если данная реплика обновляется, все другие реплики того же фрагмента данных также должны быть обновлены в одной и той же транзакции. Логически это означает, что существует лишь одна версия данных. В большинстве продуктов синхронная репликация реализуется с помощью триггерных процедур (возможно, скрытых и управляемых системой). Но синхронная репликация имеет

недостаток: она создает дополнительную нагрузку при выполнении всех транзакций, в которых обновляются какие-либо реплики (кроме того, могут возникать проблемы, связанные с доступностью данных).

В случае асинхронной репликации обновление одной реплики распространяется на другие спустя некоторое время, а не в той же транзакции. Таким образом, при асинхронной репликации вводится задержка или время ожидания, в течение которого отдельные реплики могут быть фактически неидентичными (то есть определение «реплика» оказывается не совсем подходящим, поскольку мы не имеем дело с точными и своевременно созданными копиями).

Вторичные серверы реплицируют журнал операций первичного сервера и асинхронно применяют операции к своим наборам данных. Если наборы данных вторичных реплик отражают первичный набор данных, набор реплик может продолжать функционировать, несмотря на отказ одного или нескольких членов.

Минимальная рекомендуемая конфигурация набора реплик состоит из трех узлов, так как в наборе реплик, состоящем только из двух узлов, нельзя получить большинство в случае отказа основного сервера. Ниже мы приведем типовые алгоритмы по настройке набора реплик.

Чтобы включить репликацию, сначала нужно создать набор реплик экземпляров *MongoDB*. Предположим, что для нашего примера у нас есть три сервера: *ServerA*, *ServerB* и *ServerC*. В этой конфигурации *ServerA* будет нашим первичным сервером, а *ServerB* и *ServerC* – нашими вторичными серверами.

Ниже приведены шаги, которые необходимо выполнить для создания набора реплик вместе с добавлением первого и последующих членов в набор.

Создание набора реплик

1. Убедитесь, что все экземпляры *mongod.exe*, которые будут добавлены в набор реплик, установлены на разных серверах.

Для каждого участника запустите экземпляр *mongod*, указав параметр *replSet*. Ниже приведен основной синтаксис – *replSet*:

mongod --port "PORT" --dbpath "YOUR_DB_DATA_PATH" --replSet "REPLICA_SET_INSTANCE_NAME"

Например, по команде

mongod --port 27017 *--dbpath "D:\mongodb\data" --replSet rs*0

будет запущен автономный экземпляр *mongod* на порту 27017 как член нового набора реплик с именем *rs*0. Команда использует существующий путь автономной базы данных *D:\mongodb\data*. Если ваше приложение подключается к более чем одному набору реплик, каждый набор должен иметь собственное имя.

Можно осуществить запуск с настройками, явно связывающими набор реплик и адреса серверов. В следующем примере указывается имя набора реплик и привязка *IP* с помощью параметров командной строки --*replSet* и --*bind_ip*:

mongod --replSet "rs0" --bind_ip localhost,<hostname(s)|ip address(es)> --port 27017

Для <*hostname*(*s*) | *ip address*(*es*)> нужно указать имя (имена) хоста и/или *IP*-адрес (адреса) для вашего экземпляра *mongod*, который удаленные клиенты (включая другие члены набора реплик) могут использовать для подключения.

Если в тестовом варианте вы хотите запускать каждый процесс на одном компьютере, можно сделать следующее.

1. Создайте каталоги данных для каждого члена набора реплик:

mkdir ~/ ServerA
mkdir ~/ ServerB
mkdir ~/ ServerC

Запустите каждый *mongod* в отдельном окне терминала:

mongod --replSet rs0 --dbpath ~ / ServerA --port 40000
mongod --replSet rs0 --dbpath ~ / ServerB --port 40001
mongod --replSet rs0 --dbpath ~ / ServerC --port 40002

Эти команды запускают три отдельных сервера на одном компьютере, при этом они будут прослушивать разные порты; мы сообщаем каждому *mongod*, что он будет членом набора реплик *rs*0.

2. Убедитесь, что все экземпляры *mongo.exe* могут подключаться друг к другу, например, из первичного *ServerA* выполните следующие две команды:

mongo --host ServerB --port 27017
mongo --host ServerC --port 27017

Аналогичным образом проделайте то же самое с остальными серверами.

3. Из первичного *ServerA* выполните инициирование набора реплик с помощью команды *rs.initiate*(). После этого набор реплик готов к работе.

4. Проверьте набор реплик, выполнив команду *rs.conf*(), чтобы убедиться, что реплика настроена правильно.

5. Убедитесь, что в наборе реплик есть первичный сервер, для этого используйте команду *rs.status*().

Добавление дополнительных серверов в набор реплик

Вторичные серверы можно добавить в набор реплик, просто используя команду *rs.add*:

rs.add(*HOST_NAME:PORT*)

Эта команда принимает имя вторичных серверов и добавляет серверы в набор репликации.

Предположим, что у вас есть *ServerA*, *ServerB* и *ServerC*, которые должны быть частью вашего набора реплик, а *ServerA* определен как первичный сервер в наборе реплик.

Чтобы добавить *ServerB* и *ServerC* в набор реплик, выполните следующие команды:

rs.add("*ServerB*")
rs.add("*ServerC*")

Перенастройка или удаление серверов

Чтобы удалить сервер из набора конфигурации, нам нужно использовать команду *rs.remove'*.

1. Сначала выполните выключение экземпляра, который вы хотите удалить. Это можно сделать с помощью команды *db.shutdownserver* из оболочки *mongo*.

2. Подключитесь к основному серверу.

3. Используйте команду *rs.remove*(), чтобы удалить требуемый сервер из набора реплик.

Предположим, что у вас есть набор реплик с *ServerA*, *ServerB* и *ServerC*, и вы хотите удалить *ServerC* из набора реплик, для этого введите следующую команду:

rs.remove("*ServerC*")

Проблемы с использованием наборов реплик

Следующие шаги представляют собой способы устранения возможных неполадок; возникающих проблем с использованием наборов реплик.

1. Убедитесь, что все экземпляры *mongo.exe* могут подключаться друг к другу.

Предположим, у вас есть три сервера с именами *ServerA*, *ServerB* и *ServerC*. С сервера *A* выполните следующие две команды:

mongo –host ServerB –port 27017
mongo –host ServerC –port 27017

2. Запустите команду *rs.status*().

Эта команда показывает статус набора реплик. По умолчанию каждый участник будет отправлять друг другу сообщения, называемые «пульсирующими» сообщениями, которые просто указывают на то, что сервер жив и работает.

Команда *status* получает статус этих сообщений и показывает, есть ли какие-либо проблемы с какими-либо членами в наборе реплик.

3. Проверьте размер коллекции *Oplog*, в которой хранится история записей, выполненных в базу данных *MongoDB*.

MongoDB использует этот журнал операций для репликации записей другим участникам в наборе реплик.

Чтобы проверить журнал операций, подключитесь к необходимому экземпляру члена и выполните команду *rs.printReplicationInfo*. Эта команда покажет размер журнала и то, как долго он может хранить транзакции в своем файле журнала прежде, чем он заполнится.

3.14. Сегментирование

***Сегментирование* (*Sharding*)** – это метод распределения данных между несколькими вычислительными узлами.

По мере того как объем хранимых и обрабатываемых СУБД растет, приложения баз данных начинает предъявлять более высокие требования к пропускной способности вычислительной системы по скорости чтения и записи, мощности стандартных серверов перестает хватать. Например, отдельный сервер может стать не способным адресовать достаточный объем ОЗУ или ему может не хватить процес-

сорных ядер для поддержания эффективной рабочей нагрузки. К тому же, по мере роста размера данных хранение резервных копий на одном диске или *RAID*-массиве может оказаться практически неосуществимым из-за физических ограничений оборудования.

Существует два метода решения проблемы роста производительности вычислительной системы: вертикальное и горизонтальное масштабирование.

Вертикальное масштабирование включает увеличение емкости отдельного сервера, например, использование более мощного центрального процессора, добавление большего объема оперативной памяти или увеличение объема дискового пространства. Ограничения в доступных технологиях могут ограничивать мощность отдельного компьютера, достаточную для данной рабочей нагрузки. Кроме того, у поставщиков облачных услуг есть жесткие ограничения на основе имеющихся и доступных конфигураций оборудования. В результате имеется практический максимум для вертикального масштабирования.

Горизонтальное масштабирование включает разделение набора данных системы и нагрузки на несколько серверов с добавлением дополнительных серверов для увеличения производительности и емкости по мере необходимости. Хотя общая скорость или емкость одного компьютера могут быть невысокими, каждый компьютер обрабатывает подмножество общей рабочей нагрузки, потенциально обеспечивая лучшую эффективность, чем один высокоскоростной сервер большей емкости. Для увеличения емкости развертывания требуется добавлять дополнительные серверы по мере необходимости, что может иметь меньшую общую стоимость, чем высокопроизводительное оборудование для отдельной машины.

MongoDB поддерживает горизонтальное масштабирование через сегментирование для поддержки развертываний с очень большими наборами данных и операций с высокой пропускной способностью.

Сегментированный кластер MongoDB состоит из следующих компонентов:

- *сегмент* (*shard*) содержит подмножество сегментированных данных. Каждый шард можно развернуть как набор реплик;
- *Mongos* действует как маршрутизатор запросов, обеспечивая интерфейс между клиентскими приложениями и сегментированным кластером. Процессы *mongos* потребляют мало ресурсов и не отвечают за сохранение данных. Обычно они запускаются на тех же маши-

нах, где работают серверы приложений. Приложение локально соединяется с *mongos*, а *mongos* уже управляет соединениями с конкретными сегментами;

• *серверы конфигурации* хранят метаданные и параметры конфигурации кластера. В состав этих данных входят: глобальная конфигурация кластера, местоположение каждой базы данных, коллекции и диапазоны хранящихся в них данных, а также журнал изменений, в котором сохраняется история миграции данных между сегментами.

На рис. 3.13 показано взаимодействие компонентов в сегментированном кластере.

MongoDB разбивает данные на уровне коллекции, распределяя их по шардам в кластере. При этом используется ключ сегмента для распределения документов коллекции по шардам.

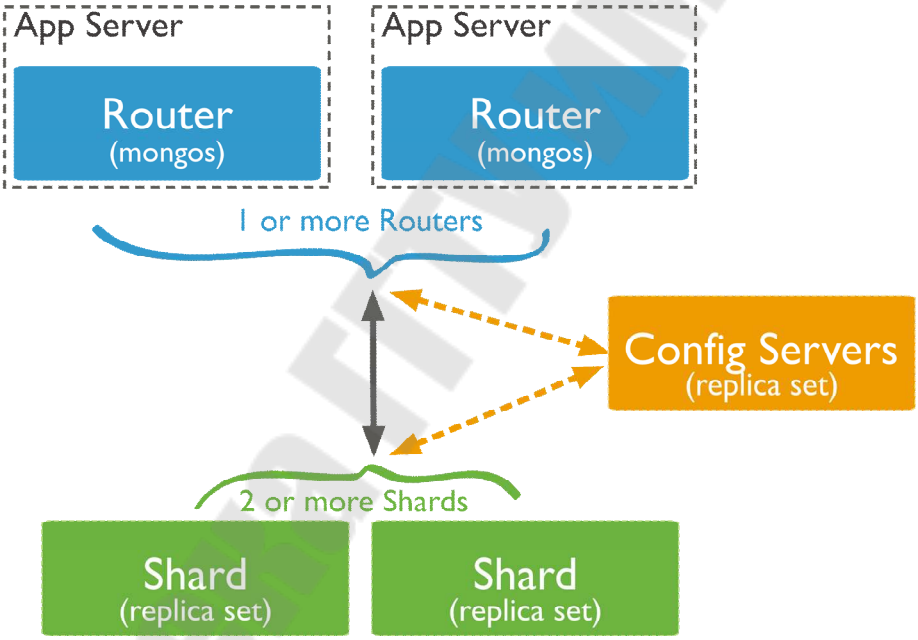

Рис. 3.13. Взаимодействие компонентов в сегментированном кластере

***Ключ сегмента* (*shard key*)** состоит из поля или нескольких полей в документах, выбирается при сегментировании коллекции. Значение ключа сегмента документа определяет его распределение по шардам.

В версии 4.2 и более ранних ключевые поля сегментов должны существовать в каждом документе для сегментированной коллекции. С версии 4.4 в документах в сегментированных коллекциях могут отсутствовать ключевые поля сегментов. Отсутствующие ключевые по-

ля сегментов обрабатываются как имеющие нулевые значения при распределении документов по сегментам, но не при маршрутизации запросов.

В версии *MongoDB* 4.2 можно обновить значение ключа шарда документа, если только поле ключа шарда не является неизменяемым полем *_id*.

В *MongoDB* 4.0 и более ранних версиях значение поля ключа шарда документа является неизменным.

Чтобы сегментировать заполненную коллекцию, нужно сделать так, чтобы она имела индекс, который начинается с ключа сегмента. При сегментировании пустой коллекции *MongoDB* создает поддерживающий индекс, если коллекция еще не имеет соответствующего индекса для указанного ключа сегментирования.

Выбор ключа сегмента влияет на производительность, эффективность и масштабируемость сегментированного кластера. Кластер с наилучшим оборудованием и инфраструктурой может оказаться узким местом из-за ошибочного выбора ключа сегментирования. Выбор ключа сегментирования и его индекса поддержки также может повлиять на стратегию сегментирования, которую может использовать ваш кластер.

MongoDB разбивает сегментированные данные на порции.

***Порция* (*chunk*)** – фрагменты документа одинакового размера, характеризующиеся непрерывным диапазоном значений ключей, находящихся в одном сегменте. Каждая порция имеет включаемый нижний и исключаемый верхний диапазоны на основе ключа сегмента.

MongoDB разбивает порции, когда они превышают настроенный максимальный размер, который по умолчанию составляет 64 мегабайт и переносит порции в другой сегмент, когда текущий сегмент содержит слишком много порций коллекции по сравнению с другими сегментами.

Непосредственно сегментация (шардинг) происходит следующим образом. Данные попадают в порции по определенному диапазону заданного поля, определяемому ключом сегмента. Сначала создается всего одна порция и диапазон значений ключа, которые он принимает, лежит в пределах (–∞, +∞). Когда размер этой порции достигает максимального, *mongos* оценивает значение всех ключей сегмента внутри порции и делит порцию таким образом, чтобы данные были разделены примерно поровну.

Для примера предположим, что у нас есть четыре документа с полями *name*, *age* и *id*, при этом *id* является ключом сегмента:

{“*name*”: “*Max*”, “*age*”: 23, “*id*”: 23}
{“*name*”: “*John*”, “*age*”: 28, “*id*”: 15}
{“*name*”: “*Nick*”, “*age*”: 19, “*id*”:56}
{“*name*”: “*Carl*”, “*age*”: 19, “*id*”: 78}

Предположим, что максимальный размер порции уже достигнут. В данном случае *mongos* разделит диапазон примерно так: (–∞, 45]; (45, +∞). У нас получится две новые порции (рис. 3.14).

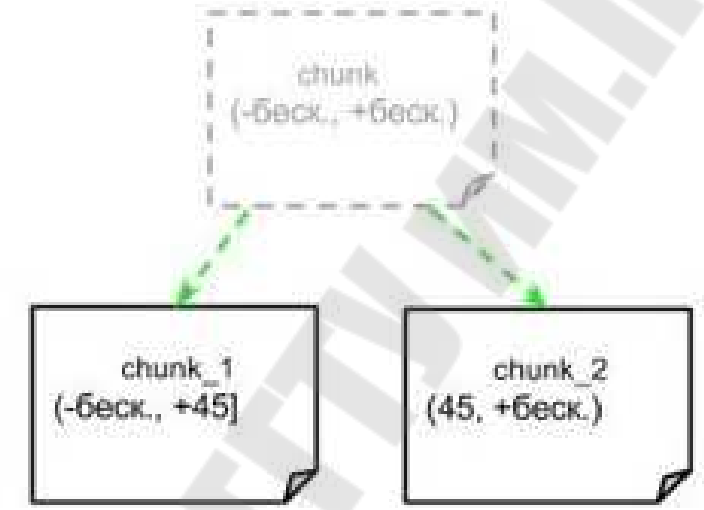

Рис. 3.14. Разделение порции данных

При появлении новых документов они будут записываться в порцию, которая соответствует диапазону ключа. По достижению предельного количества записей в одину порцию разделение произойдет снова, и диапазон будет еще уже. Все порции хранятся в сегментах.

Следует отметить, что при достижении какой-либо порцией неделимого диапазона, например (44, 45], деление происходить не будет, и порция будет расти свыше максимального предела. В связи с этим следует продуманно выбирать сегментный ключ, чтобы это была как можно более случайная величина. Например, если нам надо сформировать хранилище данных всех людей на планете, то удачными выборами сегментного ключа были бы номер телефона, идентификационный налоговый номер, почтовый индекс, а неудачными – имя, город проживания.

Функция балансера, являющегося составной частью *mongos*, заключается в миграции порций из одного сегмента в другой. Процесс происходит примерно так: балансер отсылает команду в сегмент, из которого будет мигрировать порция, а шард, получая эту команду, запускает процесс копирования порции в другой сегмент. После того как все документы скопированы, происходит синхронизация документов между этими двумя порциями, поскольку пока происходила миграция, в исходную порцию могли добавиться новые данные. После окончания синхронизации шард, который принял новую порцию, отправляет адрес конфигурационному серверу. По окончании этого процесса, если в исходной порции нет открытых курсоров, она удаляется.

Преимущества сегментирования

Распределение рабочей нагрузки чтения и записи по шардам в сегментированном кластере позволяет каждому сегменту обрабатывать подмножество кластерных операций. Рабочие нагрузки чтения и записи можно масштабировать по кластеру по горизонтали, добавляя дополнительные сегменты. Операции, которые включают ключ осколка или префикс составного ключа осколка *mongos*, могут нацелить запрос на определенный шард или набор шардов. Эти целевые операции обычно более эффективны, чем широковещательная рассылка на каждый сегмент в кластере.

Распределение данных по шардам в кластере позволяет каждому шарду содержать подмножество общих данных кластера. По мере роста набора данных дополнительные сегменты увеличивают емкость хранилища кластера.

Развертывание серверов конфигурации и сегментов в качестве наборов реплик обеспечивает повышенную доступность. Даже если один или несколько наборов реплик сегментов становятся полностью недоступными, сегментированный кластер может продолжать выполнять частичное чтение и запись.

Пример создания нового сегментированного кластера, который состоит из *mongos*, набора реплик сервера конфигурации и двух наборов реплик сегментов, описан в документации *https://docs.mongodb.com/manual/tutorial/deploy-shard-cluster/*.

3.15. Управление хранением данных

Механизм хранения – это часть системы управления базами данных, которая отвечает за управление хранением данных как в оперативной памяти, так и на диске. Многие СУБД поддерживают не-

сколько механизмов хранения, при этом разные механизмы лучше подходят для конкретных рабочих нагрузок. Например, один механизм хранения может предложить лучшую производительность для рабочих нагрузок с большим количеством операций чтения, а другой может поддерживать более высокую пропускную способность для операций записи.

MongoDB предоставляет множество механизмов хранения, что позволяет выбрать наиболее подходящий для конкретного приложения баз данных. Выбор подходящего механизма хранения может значительно повлиять на производительность приложений.

С версии *MongoDB* 4.2 и более поздних был удален устаревший механизм хранения *MMAPv1* и основным механизмом по умолчанию стал *WiredTiger Storage Engine*.

WiredTiger – это расширяемая платформа *NoSQL* с открытым исходным кодом для управления данными, которая использует архитектуру *MultiVersion Concurrency Control* (*MVCC*). Она хорошо подходит для большинства рабочих нагрузок и рекомендуется для новых развертываний. *WiredTiger*, помимо других функций, предоставляет модель параллелизма на уровне документа, контрольные точки и сжатие. В *MongoDB Enterprise WiredTiger* поддерживает также шифрование в состоянии покоя.

В *MongoDB Enterprise* также дополнительно доступен механизм хранения в памяти ***In-Memory Storage Engine***. Он позволяет сохранять документы в оперативной памяти для более предсказуемой задержки работы с данными.

Остановимся подробнее на возможностях *WiredTiger Storage Engine*.

Параллелизм на уровне документа

Механизм *WiredTiger* использует управление параллелизмом на уровне документа для операций записи. В результате несколько клиентов могут одновременно изменять разные документы коллекции. Для большинства операций чтения и записи используется оптимистичный контроль параллелизма, который предполагает, что несколько транзакций могут часто выполняться, не мешая друг другу. Во время выполнения транзакции используют ресурсы данных без получения блокировок этих ресурсов. Перед фиксацией каждая транзакция убеждается в том, что никакая другая транзакция не изменила прочитанные данные. Если проверка выявляет конфликтующие изменения, фиксирующая транзакция откатывается, и ее можно перезапустить.

WiredTiger использует только блокировки намерения на гло-

бальном уровне, уровне базы данных и коллекции. Когда подсистема хранения обнаруживает конфликты между двумя операциями, в одной из них возникает конфликт записи, в результате чего *MongoDB* прозрачно повторяет эту операцию.

Некоторые глобальные операции (обычно краткосрочные операции с участием нескольких баз данных) по-прежнему требуют глобальной блокировки «на уровне экземпляра». Некоторые другие операции, такие как *collMod*, позволяющие добавлять параметры в коллекцию или изменять определения представлений, по-прежнему требуют исключительной блокировки базы данных.

Снимки и контрольные точки

WiredTiger использует *MVCC* (*MultiVersion Concurrency Control* – управление параллельным доступом посредством многоверсионности) – механизм для обеспечения параллельного доступа к базам данных, заключающийся в предоставлении каждому пользователю так называемого «снимка» базы, обладающего свойством вносить пользователем изменения так, чтобы другие пользователи не видели их до момента фиксации транзакции. Этот способ управления позволяет добиться того, что пишущие транзакции не блокируют читающие, а читающие транзакции не блокируют пишущие.

В начале операции *WiredTiger* предоставляет моментальный снимок данных для операции. Снимок представляет собой согласованное представление данных в памяти. При записи на диск *WiredTiger* записывает все данные моментального снимка на диск согласованным образом во всех файлах данных. Теперь надежные данные действуют как контрольная точка в файлах данных. Контрольная точка обеспечивает согласованность файлов данных до последней контрольной точки включительно, то есть контрольные точки могут действовать как точки восстановления.

MongoDB настраивает *WiredTiger* для создания контрольных точек (записи данных моментального снимка на диск) с интервалом в 60 секунд. Во время записи новой контрольной точки предыдущая контрольная точка все еще действительна. Таким образом, даже если *MongoDB* завершает работу или обнаруживает ошибку при записи новой контрольной точки, после перезапуска СУБД может восстановиться с последней действительной контрольной точки.

Новая контрольная точка становится доступной и постоянной, когда таблица метаданных *WiredTiger* атомарно обновляется для ссылки на новую контрольную точку. Как только новая контрольная

точка станет доступной, *WiredTiger* освобождает страницы из старых контрольных точек.

Журнал

Используя *WiredTiger*, *MongoDB* может восстановиться после последней контрольной точки даже без ведения журнала. Однако чтобы восстановить изменения, сделанные после последней контрольной точки, следует использовать журнализацию.

Механизм *WiredTiger* использует журнал упреждающей записи (или журнал) в сочетании с контрольными точками для обеспечения надежности данных.

Журнал сохраняет все изменения данных между контрольными точками. Если *MongoDB* выходит между контрольными точками, он использует журнал для воспроизведения всех данных, измененных с момента последней контрольной точки.

По умолчанию журнал сжимается с помощью библиотеки мгновенного сжатия. Чтобы указать другой алгоритм сжатия или не использовать сжатие, нужно произвести соответсвующую настройку с помощью параметра *storage.wiredTiger.engineConfig.journal-Compressor.*

Следует отметить, что, если запись журнала меньше или равна 128 байтам, *WiredTiger* не сжимает эту запись.

Можно отключить ведение журнала для автономных экземпляров, установив для *storage.journal.enabled* значение *false*, что может снизить накладные расходы на ведение журнала. Для автономных экземпляров отказ от журнала означает, что при неожиданном выходе из *MongoDB* вы потеряете все изменения данных до последней контрольной точки.

В новых версиях *MongoDB* нельзя указать параметр *--nojournal* или *storage.journal.enabled: false* для членов набора реплик, которые используют механизм хранения *WiredTiger*.

Сжатие

MongoDB поддерживает сжатие для всех коллекций и индексов. Сжатие сводит к минимуму использование хранилища за счет дополнительного процессора. По умолчанию *WiredTiger* использует сжатие блоков с библиотекой мгновенного сжатия для всех коллекций и сжатие префиксов для всех индексов.

Для коллекций также доступны библиотеки сжатия блоков *Zlib* и *Zstd* (доступно с версии *MongoDB* 4.2).

Чтобы указать альтернативный алгоритм сжатия или отсутствие сжатия, используйте параметр *storage.wiredTiger.collection-Config.block Compressor.*

Для индексов, чтобы отключить сжатие префиксов, используйте параметр *storage.wiredTiger.indexConfig.prefixCompression*.

Параметры сжатия также можно настроить для каждой коллекции и индекса во время создания коллекции и индекса.

Для большинства рабочих нагрузок параметры сжатия по умолчанию уравновешивают эффективность хранения и требования к обработке.

Журнал *WiredTiger* также по умолчанию сжимается.

Использование памяти

С *WiredTiger MongoDB* использует как внутренний кэш *WiredTiger*, так и кэш файловой системы. Размер внутреннего кэша *WiredTiger* по умолчанию больше: 50 % (*RAM* – 1 гигабайт), или 256 мегабайтов.

Например, в системе с 4 гигабайтами ОЗУ кэш *WiredTiger* будет использовать 1,5 гигабайта ОЗУ (0,5 · (4 гигабайта – 1 гигабайт) = 1,5 гигабайта). И наоборот, система с общим объемом ОЗУ 1,25 гигабайта будет выделять 256 мегабайтов кэш-памяти *WiredTiger*, потому что это больше половины от общего объема ОЗУ минус один гигабайт (0,5 · (1,25 гигабайт – 1 гигабайт) = 128 мегабайтов < 256 мегабайтов).

В некоторых случаях, например, при работе в контейнере, база данных может иметь ограничения памяти ниже, чем общая системная память. В таких случаях этот предел памяти используется в качестве максимальной доступной оперативной памяти.

По умолчанию *WiredTiger* использует сжатие блоков *Snappy* для всех коллекций и сжатие префиксов для всех индексов. Параметры сжатия по умолчанию настраиваются на глобальном уровне, а также устанавливаются для каждой коллекции и индекса во время их создания.

Для данных во внутреннем кэше *WiredTiger* используют разные представления по сравнению с форматом на диске.

Данные в кэше файловой системы такие же, как и в формате на диске, включая преимущества любого сжатия файлов данных. Кэш файловой системы используется операционной системой для уменьшения дискового ввода-вывода.

Индексы, загруженные во внутренний кеш *WiredTiger*, имеют представление данных, отличное от формата на диске, но все же могут использовать сжатие префиксов индекса для уменьшения использования ОЗУ. Сжатие префиксов индекса дедуплицирует общие префиксы из индексированных полей.

Данные коллекции во внутреннем кэше *WiredTiger* несжаты, используют представление, отличное от формата на диске. Блочное

сжатие может обеспечить значительную экономию на диске, но данные должны быть несжатыми, чтобы сервер мог ими управлять.

Через кеш файловой системы *MongoDB* автоматически использует всю свободную память, которая не используется кешем *WiredTiger* или другими процессами.

Глава 4. ПРАКТИЧЕСКАЯ РАБОТА С СУБД *MongoDB*

MongoDB предлагает как корпоративную (*Enterprise and Community*), так и общественную (*Community*) версию своей СУБД. Эти версии различаются политиками лицензирования и возможностями.

Версия *Community* предлагает гибкую модель документа, а также специальные запросы, индексацию и агрегирование в реальном времени, чтобы обеспечить эффективные способы доступа к данным и способы их анализа. В качестве распределенной системы получает высокую доступность за счет встроенной репликации и аварийного переключения, а также горизонтальную масштабируемость с собственным сегментированием.

MongoDB Enterprise доступен как часть подписки *MongoDB Enterprise Advanced*, которая обеспечивает наиболее полную поддержку со стороны производителя при запуске СУБД *MongoDB* в своей собственной инфраструктуре. *MongoDB Enterprise Advanced* также предоставляет комплексные операционные инструменты, расширенную аналитику и визуализацию данных, интеграцию платформ и сертификацию, а также обучение по запросу для вашего персонала.

MongoDB Enterprise предоставляет различные функции, недоступные в версии *MongoDB Community*, например: механизм хранения в памяти, аудиторскую проверку, проверку подлинности *Kerberos*, аутентификацию и авторизацию *LDAP*, шифрование в состоянии покоя.

Хотя *Enterprise*-версия обладает несколько большими возможностями, доступна она только в пробном режиме или по подписке, которая предполагает внесение определенной платы.

В данной главе будет описана установка программного обеспечения СУБД *MongoDB Community Edition* на различные операционные системы. По состоянию на момент написания данной книги эту версию можно установить на такие операционные системы, как *Linux*, *Windows*, *MacOS*.

4.1. Установка, запуск и использование СУБД *MongoDB* в *Linux*

Для оптимальной установки *MongoDB* предоставляет пакеты для популярных дистрибутивов *Linux*. В официальной документации под-

робно описывается процесс установки для таких систем, как *Red Hat*, *Ubuntu*, *Debian*, *SUSE*, *Amazon*. В настоящий момент *MongoDB* не поддерживает подсистему *Windows* для *Linux* (*WSL*).

Рассмотрим специфику установки *MongoDB Community Edition* в наиболее популярном дистрибутиве *Linux – Ubuntu Linux*.

В настоящий момент *MongoDB* 4.4 *Community Edition* поддерживает следующие выпуски 64-разрядных версий: *Ubuntu LTS* (долгосрочная поддержка) на архитектуре *x*86_64: 20.04 *LTS* ("*Focal*"), 18.04 *LTS* ("*Bionic*"), 16.04 *LTS* ("*Xenial*").

Следует отметить, что *MongoDB* поддерживает только 64-битные версии этих платформ. *MongoDB* 4.4 *Community Edition* на *Ubuntu* также поддерживает архитектуры *ARM*64 и *s*390*x*.

Для установки *MongoDB Community* в *Ubuntu* будет использоваться официальный пакет *mongodb-org*. Официальный пакет *mongodb-org* всегда содержит последнюю версию *MongoDB*, доступен в отдельном репозитории.

Следует отметить, что пакет *mongodb*, предоставляемый *Ubuntu*, не поддерживается *MongoDB Inc*. и конфликтует с официальным пакетом *mongodb-org*. Если вы уже установили пакет *mongodb* в своей системе *Ubuntu*, вы должны сначала удалить пакет *mongodb*, прежде чем продолжить установку согласно описываемой ниже последовательности.

Чтобы установить *MongoDB Community Edition*, следует выполнить следующие действия с помощью диспетчера пакетов *apt* (табл. 4.1).

1. Импорт открытого ключа, используемого системой управления пакетами.

Из терминала следует выполнить следующую команду, чтобы импортировать открытый ключ *GPG MongoDB* из *https://www.mongodb.org/static/pgp/server*-4.4.*asc*:

wget -qO - https://www.mongodb.org/static/pgp/server-4.4.*asc* | *sudo apt-key add -*

Операция должна ответить *OK*.

Таблица 4.1

Пакеты *MongoDB Community Edition*

Имя пакета	Описание
mongodb-org	Метапакет, который автоматически устанавливает пакеты компонентов, перечисленные ниже
mongodb-org-server	Содержит демон (службу) *mongod*, связанный сценарий инициализации и файл конфигурации (*/etc/mongod.conf*)
mongodb-org-mongos	Содержит демон (службу) *mongos*
mongodb-org-shell	Содержит консольную оболочку *mongo*
mongodb-org-tools	Метапакет, который автоматически устанавливает пакеты компонентов: инструменты базы данных *mongodb-database-tools* (*mongodump*, *mongorestore*, *bsondump*, *mongoimport*, *mongoexport*, *mongostat*, *mongotop*, *mongofiles*), скрипт *install_compass mongodb-org-database-tools-extra*

При получении сообщения об ошибке, указывающего, что *gnupg* не установлен, можно установить его и необходимые библиотеки с помощью следующей команды:

sudo apt-get install gnupg

После установки следует повторить попытку импорта ключа:

wget -qO - https://www.mongodb.org/static/pgp/server-4.4.*asc* | *sudo apt-key add –*

2. Создание файла списка.
Для *Ubuntu* 20.04 (*Focal*) команда выглядит следующим образом:

echo "deb [*arch=amd*64,*arm*64] *https://repo.mongodb.org/apt /ubuntu focal/mongodb-org/4.4 multiverse"* | *sudo tee /etc/apt/sources. list.d/mongodb-org*-4.4.*list*

3. Перезагрузка локальной базы данных пакетов.
Следующая команда перезагрузит локальную базу данных пакетов:

sudo apt-get update

4. Установка пакетов *MongoDB*.

Существует возможность установить как последнюю стабильную версию *MongoDB*, так и определенную версию *MongoDB*.

Чтобы установить последнюю стабильную версию, нужно выполнить следующую команду:

sudo apt-get install -y mongodb-org

Чтобы установить конкретный выпуск, необходимо указать каждый пакет компонентов отдельно вместе с номером версии, как в следующем примере:

sudo apt-get install -y mongodb-org = 4.4 *mongodb-org-server* = 4.4 *mongodb-org-shell* = 4.4 *mongodb-org-mongos* = 4.4 *mongodb-org-tools* = 4.4

Если вы устанавливаете только *mongodb-org* = 4.4 и не включаете пакеты компонентов, последняя версия каждого пакета *MongoDB* будет установлена независимо от того, какую версию вы указали.

Хотя можно указать любую доступную версию *MongoDB*, *apt-get* автоматически обновит пакеты, когда станет доступна более новая версия. Чтобы предотвратить непреднамеренные обновления, можно закрепить пакет в текущей установленной версии:

echo "mongodb-org hold" | sudo dpkg --set-selections
echo "mongodb-org-server hold" | sudo dpkg --set-selections
echo "mongodb-org-shell hold" | sudo dpkg --set-selections
echo "mongodb-org-mongos hold" | sudo dpkg --set-selections
echo "mongodb-org-tools hold" | sudo dpkg --set-selections

Большинство *Unix*-подобных операционных систем ограничивают системные ресурсы, которые может использовать процесс. Эти ограничения могут негативно повлиять на работу *MongoDB*, и их можно (следует) изменить. Например, начиная с *MongoDB* 4.4, возникает ошибка запуска, если значение *ulimit* для количества открытых файлов меньше 64000.

Параметры утилиты *UNIX* для вашей платформы указаны по адресу: *https://docs.mongodb.com/manual/ reference/ulimit/*.

Если установка СУБД произведена через диспетчер пакетов, каталог данных */var/lib/mongodb* и каталог журнала */var/log/mongodb* будут созданы во время установки.

По умолчанию *MongoDB* запускается с использованием учетной записи пользователя *mongodb*. Если вы измените пользователя, который запускает процесс *MongoDB*, вы также должны изменить права доступа к каталогам данных и журналов, чтобы предоставить этому пользователю доступ к этим каталогам.

Официальный пакет *MongoDB* включает файл конфигурации (*/etc/mongod.conf*). Эти настройки (например, спецификации каталога данных и каталога журналов) вступают в силу при запуске. То есть, если вы изменяете файл конфигурации во время работы экземпляра *MongoDB*, нужно перезапустить экземпляр, чтобы изменения вступили в силу.

Для запуска и управления процессом *mongod* используют встроеннюю систему инициализации операционной системы. Последние версии *Linux*, как правило, используют *systemd* (который использует команду *systemctl*).

Запустить *MongoDB* можно, запустив процесс *mongod* и введя следующую команду:

sudo systemctl start mongod

При запуске можно получить сообщение об ошибке, подобное приведенному ниже:

Failed to start mongod.service: Unit mongod.service not found (Не удалось запустить *mongod.service*: модуль *mongod.service* не найден).

В этом случае сначала следует выполнить команду:

sudo systemctl daemon-reload

После этого – снова запустить команду *start*, указанную выше.

Убедиться, что *MongoDB* успешно запущена, можно, введя следующую команду:

sudo systemctl status mongod

При желании можно убедиться, что *MongoDB* запустится после перезагрузки системы, выполнив следующую команду:

sudo systemctl enable mongod

При необходимости можно остановить *MongoDB*, остановив процесс *mongod* и введя следующую команду:

sudo systemctl stop mongod

Можно перезапустить *MongoDB*, перезапустив процесс *mongod*, и введя следующую команду:

sudo systemctl restart mongod

Можно следить за состоянием процесса на предмет ошибок или важных сообщений, наблюдая за выводом в */var/log/mongodb/mongod.log* файл.

Чтобы начать использовать *MongoDB*, нужно запустить оболочку *mongo* на том же хосте, что и процесс *mongod*. Можно запустить оболочку *mongo* без каких-либо параметров командной строки, чтобы подключиться к *mongod*, который работает на вашем локальном хосте с портом по умолчанию 27017:

mongo

Чтобы облегчить начало использования *MongoDB*, компания разработчиков предоставляет руководство по началу работы в различных версиях драйверов. Официальную документацию по соответствующему драйверу можно найти в разделе *https://api.mongodb.com/* .

Адрес и порт *TCP*, на котором экземпляр *MongoDB* прослушивает клиентские подключения, задается параметром *bindIp*. По умолчанию *MongoDB* запускается с *bindIp*, установленным на 127.0.0.1, который привязывается к сетевому интерфейсу *localhost*. Это означает, что *mongod* может принимать соединения только от клиентов, работающих на одной машине. Удаленные клиенты не смогут подключиться к *mongod*, и *mongod* не сможет инициализировать набор реплик, если это значение не установлено на допустимый сетевой интерфейс.

Это значение можно настроить либо в файле конфигурации *MongoDB* с помощью *bindIp* или через аргумент командной строки – *bind_ip*.

Перед привязкой к любому *IP*-адресу, отличному от локального хоста (например, общедоступному), нужно убедиться, защищен ли кластер или нет от несанкционированного доступа. Например, как минимум, рассмотреть возможность включения аутентификации и усиления сетевой инфраструктуры.

4.2. Установка, запуск и использование *MongoDB* в *Windows*

MongoDB 4.4 *Community Edition* поддерживает следующие 64-разрядные версии *Windows* на архитектуре *x*86_64: *Windows Server* 2019, *Windows* 10/*Windows Server* 2016. *MongoDB* поддерживает только 64-битные версии этих платформ.

Установить *MongoDB Community Edition* можно с помощью мастера установки *MongoDB*. В процессе установки устанавливаются как двоичные файлы *MongoDB*, так и файл конфигурации по умолчанию, который размещается в *<install directory>\bin\mongod.cfg*.

Для загрузки установщика *MongoDB* перейдите по следующей ссылке: *https://www.mongodb.com/try/download/community?tck= docs_server*. Выберите требуемую версию СУБД, операционную систему *Windows*, тип установщика *msi* и нажмите кнопку *Download* (Загрузить) для загрузки файла на компьютер. Запустите загруженный файл и следуйте указаниям мастера установки.

Мастер проведет вас через установку *MongoDB* и *MongoDB Compass*.

В процессе ответов на вопросы можно выбрать тип установки: «Полная» (рекомендуется для большинства пользователей) или «Выборочная». Параметр «Полная установка» устанавливает *MongoDB* и инструменты *MongoDB* в расположение по умолчанию. Параметр «Выборочная установка» позволяет указать, какие исполняемые файлы устанавливаются и где.

С версии 4.0 *MongoDB* можно запустить в качестве службы *Windows* во время установки или просто установить двоичные файлы.

Работа *MongoDB* в качестве службы *Windows*

Если вы установили *MongoDB* в качестве службы *Windows*, то она запускается после успешной установки. Чтобы начать использовать *MongoDB*, подключите оболочку *mongo.exe* к работающему экземпляру *MongoDB*. Для этого в проводнике *Windows* перейдите в каталог установки, по умолчанию это *C:\Program Files\MongoDB*

Server\4.4\bin, дважды щелкните файл *mongo.exe*. Или откройте интерпретатор команд *Windows* (*cmd.exe*) с правами администратора и запустите:

*C:\Program Files\MongoDB\Server*4.4*\bin\mongo.exe*

Чтобы остановить/приостановить службу *MongoDB*, следует использовать консоль служб (*services.msc*):

1) в консоли служб найдите службу *MongoDB*;

2) щелкните правой кнопкой мыши службу *MongoDB* и выберите «Остановить» (или «Пауза»).

Чтобы удалить службу *MongoDB*, сначала необходимо использовать консоль служб (*services.msc*), чтобы остановить службу, а затем – командную строку/интерпретатор *Windows* (*cmd.exe*) от имени администратора, выполнив следующую команду:

sc.exe delete MongoDB

Работа *MongoDB* как приложения

Если в процессе установки были установлены только исполняемые файлы и *MongoDB* не была установлена как служба *Windows*, необходимо вручную запустить экземпляр *MongoDB* для последующего его использования. Это можно сделать из командной строки/интерпретатора *Windows* (*cmd.exe*). Для этого необходимо открыть командную строку/интерпретатор *Windows* (*cmd.exe*) от имени администратора. Затем создать каталог, в котором *MongoDB* будет хранить данные. Путь к каталогу данных *MongoDB* по умолчанию – это абсолютный путь *\data\db* на диске, с которого вы запускаете *MongoDB*.

Для реализации вышеизложенного в интерпретаторе команд необходимо выполнить следующие команды:

```
cd C:\
md "\data\db"
```

Чтобы запустить *MongoDB*, запустите приложение *mongod.exe*:

"C:\Program Files\MongoDB\Server\4.4\bin\mongod.exe" --dbpath="c:\data\db"

Параметр --*dbpath* указывает на каталог вашей базы данных.

Если сервер *MongoDB* работает правильно, интерпретатор команд должен отображать следующее:

[*initandlisten*] *waiting for connections*

В зависимости от настроек брандмауэра защитника *Windows* на вашем хосте *Windows* операционная система может отображать диалоговое окно «Предупреждение системы безопасности» о блокировании «некоторых функций» *C:\Program Files\MongoDB\ Server*\4.4*bin*\ *mongod.exe* для связи по сети. Чтобы решить эту проблему, нужно дать необходимое разрешение на доступ для частой сети.

Чтобы подключить оболочку *mongo.exe* к работающему экземпляру СУБД *MongoDB*, откройте другой интерпретатор команд с правами администратора и запустите:

C:\Program Files\MongoDB\Server\4.4*bin**mongo.exe*

По умолчанию *MongoDB* запускается с *bindIp*, установленным на 127.0.0.1, который привязывается к сетевому интерфейсу *localhost*. Это означает, что *mongod.exe* может принимать соединения только от клиентов, работающих на одном компьютере. Удаленные клиенты не смогут подключиться к *mongod.exe*, а *mongod.exe* не сможет инициализировать набор реплик, если это значение не будет установлено на допустимый сетевой интерфейс.

Это значение можно настроить либо в файле конфигурации *MongoDB* с помощью *bindIp* или через аргумент командной строки --*bind_ip*.

Перед привязкой к *IP*-адресу, отличному от локального хоста (например, общедоступному), убедитесь, что вы защитили свой кластер от несанкционированного доступа. Как минимум, следует рассмотреть возможность включения аутентификации и усиления сетевой инфраструктуры.

Все примеры командной строки в этом руководстве представлены как абсолютные пути к двоичным файлам *MongoDB*. Вы можете добавить *C:\Program Files\MongoDB\ Server*\ 4.4*bin* в системную перменную *PATH*, а затем можете опустить полный путь к двоичным файлам *MongoDB*.

4.3. Оболочка *mongo shell*

Mongo shell, или оболочка ***mongo*** – это консольная программа, обеспечивающая интерактивный интерфейс, принимающий выражения на языке *JavaScript* для *MongoDB*. Эту оболочку можно использовать для выполнения запросов на получение и обновление данных, а также для выполнения административных операций.

Mongo shell включена в установку сервера *MongoDB*. Если вы уже установили сервер, то программное обеспечение оболочки устанавливается в то же место, что и двоичный файл сервера.

Mongo shell можно установить и как отдельный пакет, выполнив следующие действия:

1) перейти к Центру загрузки *MongoDB https://www.mongodb.-com/try/download/community?tck=docs_server*;

2) выбрать предпочитаемую версию и платформу из раскрывающихся списков;

3) выбрать пакет для загрузки в соответствии с вашей платформой;

4) скопировать *mongo shell* из архива в необходимое место в файловой системе.

После того как вы загрузили *mongo shell*, можно использовать ее для подключения к работающему серверу *MongoDB*. Убедившись, что сервер *mongod* запущен, откройте окно терминала (или командную строку для *Windows*) и перейдите в каталог *<mongo shell installation dir>*:

cd <mongo shell installation dir>

Добавление *<mongo shell installation dir>* в переменную среды *PATH* позволяет вам набирать *mongo* напрямую вместо того, чтобы сначала переходить в каталог *<mongo shell installation dir>* или указывать полный путь к двоичному файлу. В качестве альтернативы можно скопировать оболочку *mongo* в место в файловой системе, которое уже присутствует в *PATH*, например, */usr/bin в Linux*.

Можно запустить оболочку *mongo* без каких-либо параметров командной строки для подключения к экземпляру *MongoDB*, запущенному на локальном хосте с портом по умолчанию 27017:

mongo

Чтобы указать порт, который будет использоваться, нужно добавить параметр командной строки *--port*. Например, чтобы подключиться к экземпляру *MongoDB*, работающему на локальном хосте с портом 28015, отличным от порта по умолчанию, нужно выполнить команду:

mongo --port 28015

Чтобы явно указать имя хоста и/или порт, можно использовать один из перечисленных ниже способов.

Явное указание строки подключения. Чтобы подключиться к экземпляру *MongoDB*, запущенному на удаленном хост-компьютере, нужно выполнить следующую команду:

*mongo "mongodb://mongodb0.example.com:*28015*"*

Использование параметра командной строки *--host <host>:<port>*. Чтобы подключиться к экземпляру *MongoDB*, запущенному на удаленном хост-компьютере, нужно выполнить следующую команду:

*mongo --host mongodb0.example.com:*28015

Использование параметров командной строки *--host <host>* и *--port <port>* . Чтобы подключиться к экземпляру *MongoDB*, запущенному на удаленном хост-компьютере, нужно выполнить следующую команду:

mongo --host mongodb0.example.com --port 28015

Для подключения к экземпляру *MongoDB* требуется прохождение аутентификации. Для этого следует указать имя пользователя, базу данных и, при необходимости, пароль в строке подключения. Например, для подключения и аутентификации к удаленному экземпляру *MongoDB* от имени пользователя *alice*:

*mongo "mongodb://alice@mongodb*0*.examples.com:*28015*/?authSource=admin"*

Можно также использовать параметры командной строки *--username <user>* и *--password*, *--authenticationDatabase <db>*. Например, для подключения и аутентификации к удаленному экземпляру *MongoDB* от имени пользователя *alice*:

*mongo --username alice --password --authenticationDatabase admin --host mongodb*0*.examples.com --port* 28015

Следует отметить, что, если вы не укажете пароль в строке подключения, оболочка сама запросит его.

Работа с *mongo shell*

Чтобы отобразить используемую базу данных, введите *db*:

db

Операция должна вернуть функцию *test*, которая является базой данных по умолчанию.

Чтобы переключить базы данных, введите *use <db>*, как в следующем примере:

use <database>

Чтобы вывести список доступных пользователю баз данных, используйте *show dbs*.

Вы можете переключиться и на несуществующие базы данных. Когда вы впервые сохраняете данные в базе данных, например, при создании коллекции, *MongoDB* предварительно создает базу данных для хранения этой коллекции. Например, следующие команды создадут как базу данных *myNewDatabase*, так и коллекцию *myCollection* во время выполнения операции *insertOne*():

use myNewDatabase
db.myCollection.insertOne({ *x*: 1 });

db.myCollection.insertOne() – один из методов, доступных в оболочке *mongo*, где *db* относится к текущей базе данных, которая установлена первой командой, а *myCollection* – это название коллекции.

Если оболочка *mongo* не принимает имя коллекции по какой-

либо причине, можно использовать альтернативный синтаксис – *db.getCollection*(). Например, если имя коллекции содержит пробел или дефис, начинается с числа или конфликтует со встроенной функцией, можно использовать следующее:

db.getCollection("3 *test*").*find*()
db.getCollection("3-*test*").*find*()
db.getCollection("*stats*").*find*()

Приглашение оболочки *mongo* имеет ограничение в 4095 кодовых точек для каждой строки. Если вы введете строку с более чем 4095 символов, оболочка усечет ее.

Наиболее простой способ получения содержимого базы данных представляет использование метод *db.collection.find*(). Действие этого метода во многом аналогично обычному *SQL*-запросу *SELECT * FROM Table*, который извлекает все строки.

Чтобы отформатировать выводимый результат, можно добавить к операции *.pretty*(), как показано ниже:

db.myCollection.find().*pretty*()

Кроме того, вы можете использовать следующие явные методы форматирования вывода в оболочке *mongo*:

- *print*() для печати без форматирования;
- *print*(*tojson* (<*obj*>)) или *printjson*() для печати с форматированием *JSON*, что эквивалентно *printjson*.

Если вы заканчиваете строку открытой круглой скобкой ('('), открытой фигурной скобкой ('{') или открытой квадратной скобкой ('['), то последующие строки начинаются с многоточия ('...'), пока вы не введете соответствующую закрывающую круглую скобку (')'), закрывающую фигурную скобку ('}') или закрывающую квадратную скобку (']'). Оболочка *mongo* ожидает появления закрывающей круглой скобки, закрывающей фигурной скобки или закрывающей квадратной скобки перед вычислением кода, как в следующем примере:

```
> if ( x > 0 ) {
... count++;
... print (x);
... }
```

Вы сможете выйти из режима продолжения строки, если введете две пустые строки, как в следующем примере:

```
> if (x > 0
...
...
>
```

Оболочка *mongo* поддерживает некоторые полезные сочетания клавиш. Например, используйте следующее:

- клавиши со стрелками вверх/вниз для прокрутки истории команд;
- <*Tab*> для автозаполнения или для перечисления возможностей завершения, как в следующем примере, где <*Tab*> используется для завершения имени метода, начинающегося с буквы *c*:

```
db.myCollection.c<Tab>
```

Поскольку существует много методов сбора, начинающихся с буквы *c*, <*Tab*> перечислит различные методы, которые начинаются с этой буквы.

Полный список сочетаний клавиш можно посмотреть в соответствующей документации.

.mongorc.js File

При запуске *mongo* проверяет домашний каталог пользователя на наличие файла *JavaScript* с именем *.mongorc.js*. Если он обнаружен, *mongo* интерпретирует содержимое *.mongorc.js* перед первым отображением приглашения. Если вы используете оболочку для выполнения файла или выражения *JavaScript*, либо с помощью параметра --*eval* в командной строке, либо путем указания файла *.js* для *mongo*, *mongo* прочитает файл *.mongorc.js* после того, как *JavaScript* завершит обработку. Вы можете предотвратить загрузку *.mongorc.js* с помощью параметра --*norc*.

Чтобы выйти из оболочки, введите *quit* () или используйте сочетание клавиш <*Ctrl-C*>.

Mongosh

Новая оболочка *MongoDB mongosh*, которая на момент написания настоящего пособия доступна в бета-версии, предлагает множество преимуществ по сравнению с оболочкой *mongo*, например:

- улучшена подсветка синтаксиса;

• улучшенная история команд;
• улучшено ведение журнала.

На стадии бета-тестирования *mongosh* поддерживает подмножество методов оболочки *mongo*. Достижение паритета функций между *mongosh* и оболочкой *mongo* – это постоянная работа.

Для обеспечения обратной совместимости методы, поддерживаемые *mongosh*, используют тот же синтаксис, что и соответствующие методы в оболочке *mongo*.

4.4. Графический клиент *Compass*

Для интерактивной работы с СУБД, наряду с консольной оболочкой, которая была описана выше, разработан официальный графический клиент *MongoDB Compass*.

MongoDB Compass – это программа, предоставляющая графический интерфейс для доступа к базам данных *MongoDB*. Она позволяет обращаться к хранимым данным без формального знания синтаксиса запросов *MongoDB*. Помимо работы с данными в удобной визуальной среде, *Compass* можно также использовать для анализа и оптимизации производительности запросов, оптимизации управления индексами, реализации проверки документов и других административных работ.

На момент написания данного учебно-методического пособия для загрузки *Mongo Compass* необходимо перейти по адресу *https://www.mongodb.com/ download-center/compass*. На странице, внешний вид которой приведен на рис. 4.1, необходимо указать параметры для загрузки конкретного экземпляра программного обеспечения – версию *Compass* и целевую операционную систему.

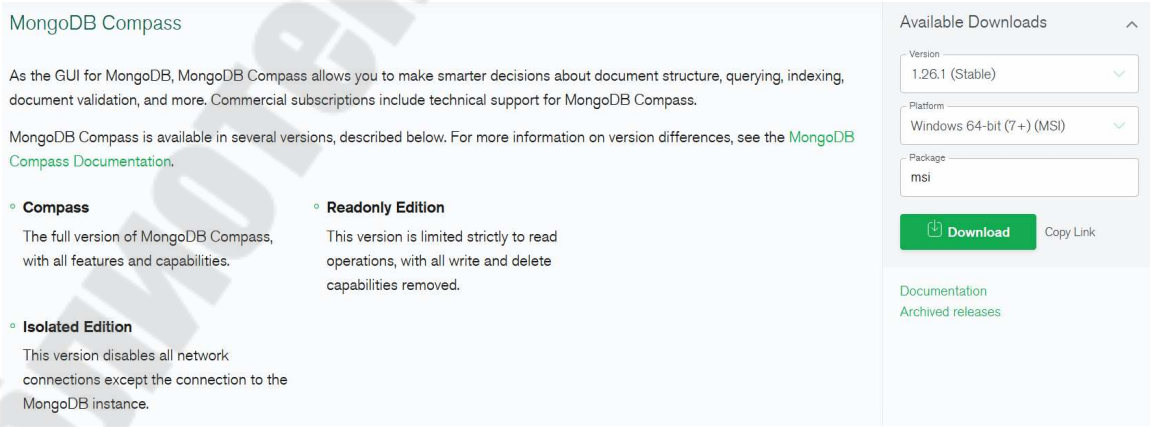

Рис. 4.1. Окно параметров загрузки *Mongo Compass*

После загрузки программы у пользователя должны быть советующие права, которые предоставляются администратором. После запуска программы установки отобразится окно, которое информирует о том, что производится установка.

Как правило, собственно процесс установки происходит достаточно быстро, и после его окончания на рабочем столе и в меню Пуск операционной системы *Windows* отобразится соответствующая иконка. Собственно, *Compass* после этого запустится автоматически (рис. 4.2, 4.3).

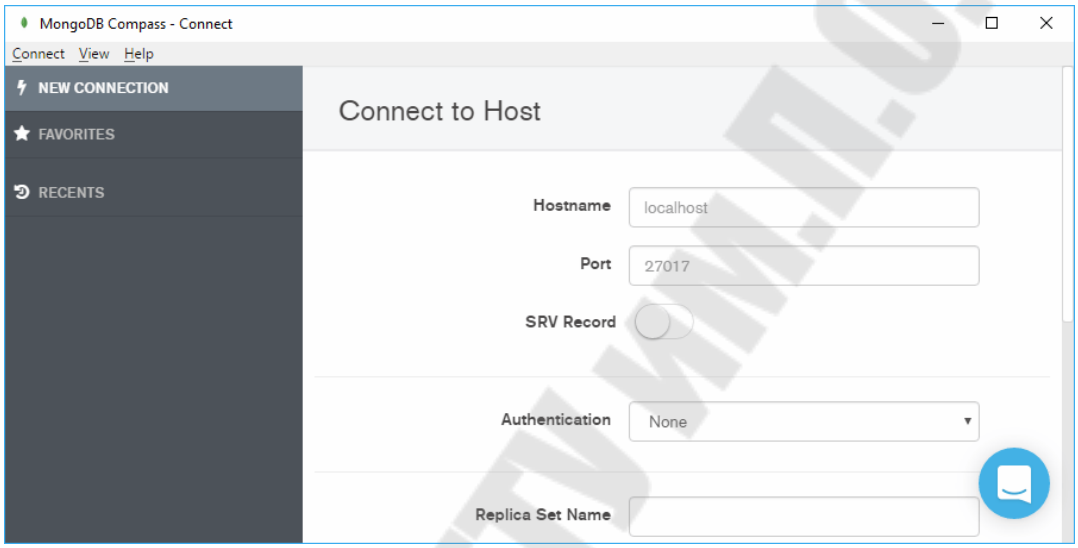

Рис. 4.2. Окно *Mongo Compass*

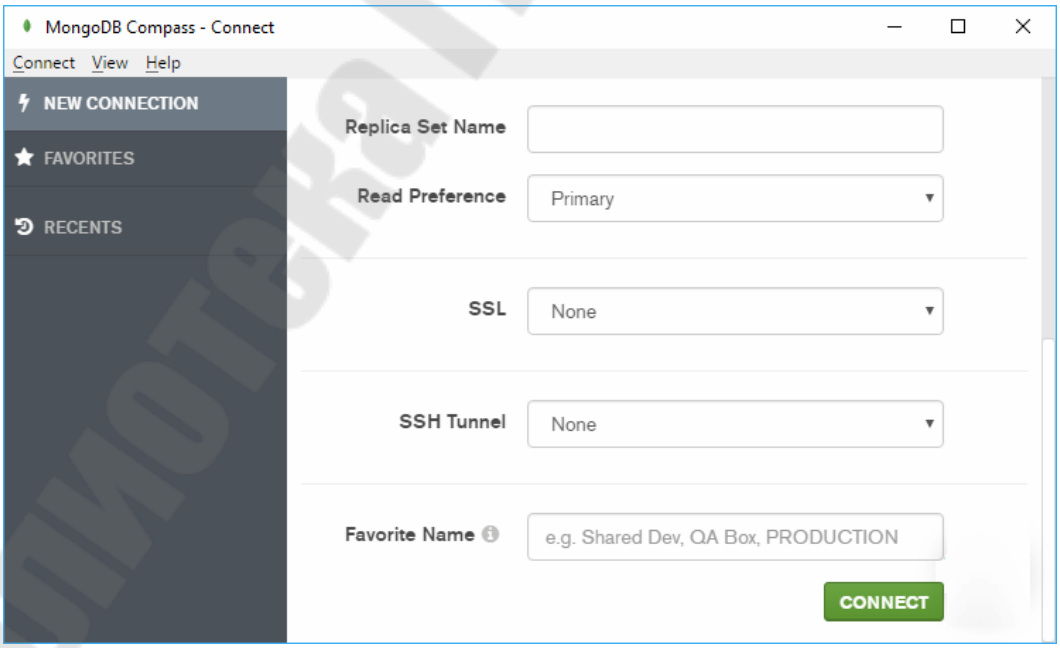

Рис. 4.3. Окно *Mongo Compass*

Поскольку программа *MongoDB Compass* является клиентским приложением, после ее запуска программы необходимо определить ряд параметров, чтобы подключиться к конкретному серверу СУБД *MongoDB*:

• *Hostname*: имя хоста сервера, если вы подключаетесь к серверу, который работает на том же компьютере, что и *Compass*, это значение будет *localhost*;

• *Port*: порт, по которому запущен сервер. Не требуется, если вы используете запись *SRV* для подключения к СУБД *MongoDB*. По умолчанию автономное развертывание работает на порту 27017. Если вы подключаетесь к набору реплик, ваш порт также может быть 27018 или 27019. Если вы не знаете, на каком порту работает ваш сервер, обратитесь к администратору базы данных за информацией;

• *SRV Record*: спецификация, которая определяет расположение сервера (например, адрес и порт). Если этот переключатель включен, то не нужно указывать порт;

• *Authentication*: тип применяемой аутентификации:

– *None;*

– *Username / Password;*

– *X.509;*

– *Kerberos;*

– *LDAP;*

• *Replica Set Name*: название реплики *MongoDB*, к которой происходит подключение. Необходимо указывать, если реплики существуют;

• *Read Preference*: определяет, как *Compass* управляет операциями чтения. Может принимать следующие опции: *Primary, Primary Preferred, Secondary, Secondary Preferred* и *Nearest*;

• *SSL*: позволяет узнать, будет ли использоваться защищенное подключение;

• *SSH tunnel*: позволяет узнать, следует ли подключаться к кластеру *MongoDB* через туннель *SSH*;

• *Favorite Name*: устанавливает имя подключения, которое можно будет использовать для повторных подключений графического клиента.

Приведем пример настройки подключения к локальному серверу. Если сервер *MongoDB* предварительно не запущен, необходимо его запустить. В *Compass* оставим все настройки подключения по умолчанию. Эти настройки и сделаны для подключения к локальному серверу, который запускается по адресу *localhost*:27017.

После нажатия на кнопку *Connect* откроется список баз данных, которые есть на сервере (рис. 4.4).

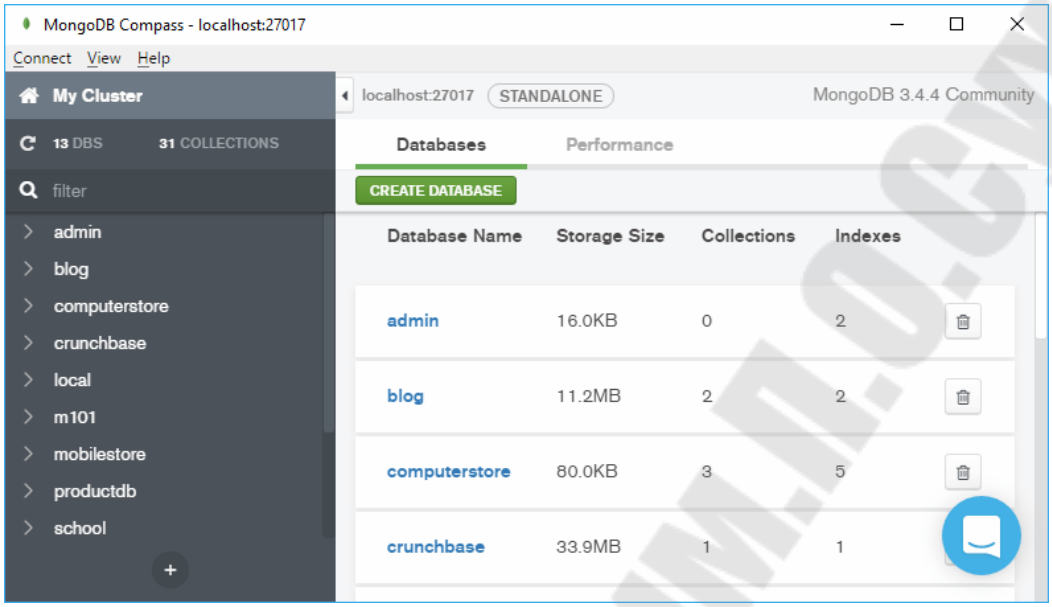

Рис. 4.4. Окно списка баз данных

Используя предоставленный графический интерфейс, можно выбрать определенную базу данных и получить связанную с ней информацию, в частности, увидеть набор коллекций в базе данных и объем, занимаемый данными (рис. 4.5).

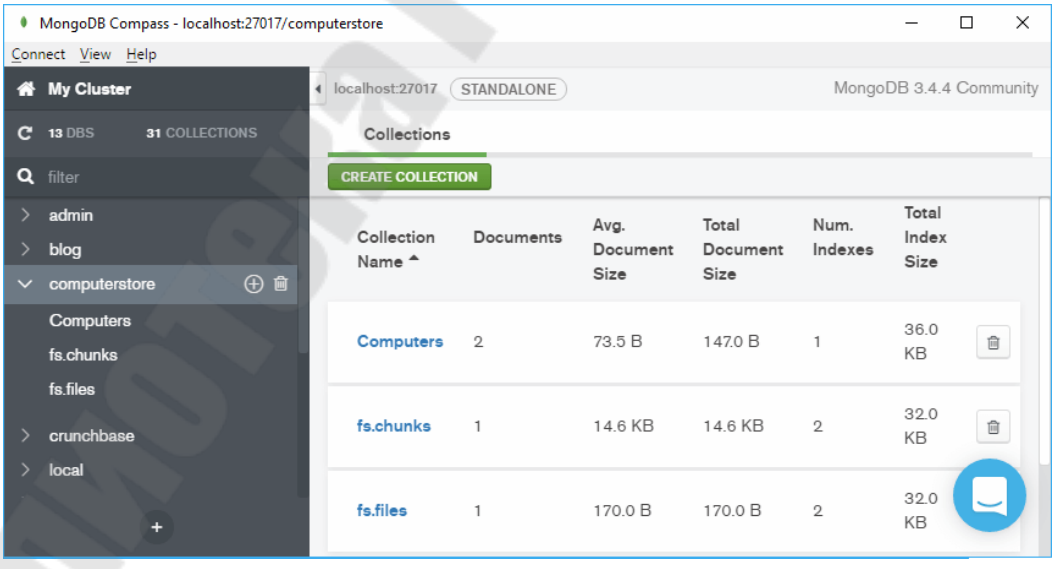

Рис. 4.5. Окно выбора из списка баз данных

Выбрав конкретную коллекцию в базе данных, можно увидеть данные, которые включены в коллекцию (рис. 4.6).

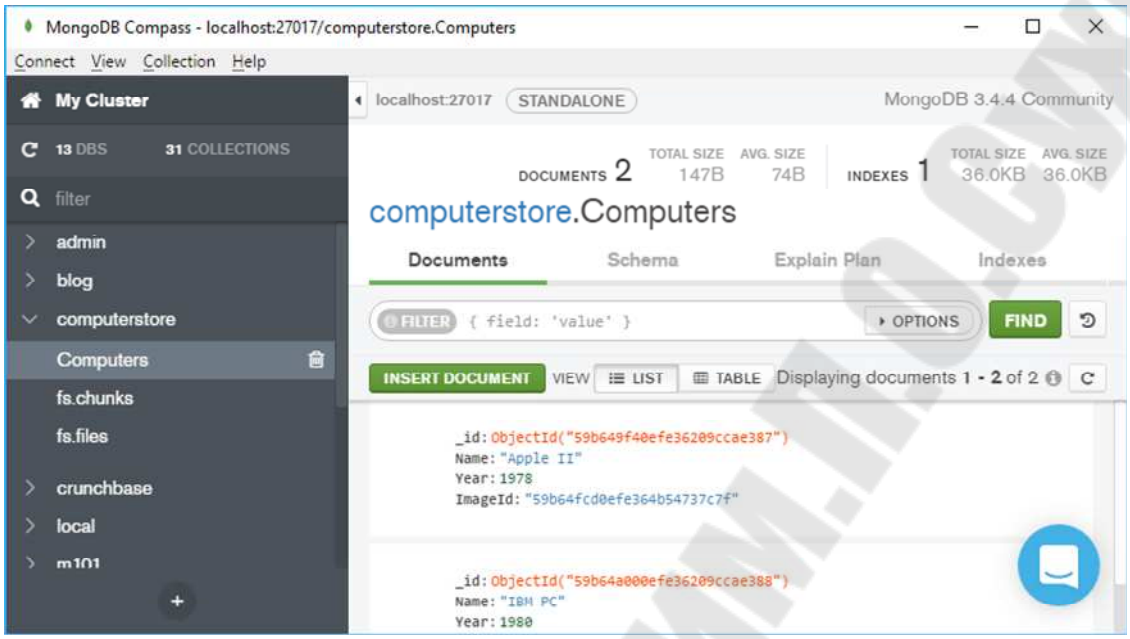

Рис. 4.6. Окно просмотра содержимого баз данных

Используя графический интерфейс программы *Compass*, можно не только просматривать эти данные, но и добавлять, изменять, удалять их.

Глава 5. СУБД *MongoDB* и *C#*

Рассмотрим применения программных средств реализации нереляционных моделей баз данных на примере СУБД *MongoDB* и *C#*. Для создания приложения на языке программирования *C#* требуется установить драйвер *MONGODB .NET*.

Пакеты *NuGet* включают поддержку четырех имен целевой платформы (*TFM*): *net*452, *netstandard*1.5, *netstandard*2.0 и *netstandard*2.1. Целевой объект *net*452 позволяет использовать драйвер с полной версией *.NET Framework* 4.5.2 и новее. *TFM netstandardX.Y* позволяет использовать драйвер с любой реализацией *.NET*, поддерживающей этот *TFM*. Это включает поддержку различных версий *.NET Core*, а также *.NET* 5.0 и выше.

На *nuget* доступно пять пакетов:

1) *MongoDB.Driver*: новый драйвер. В основном он не содержит устаревшего кода и должен использоваться для всех новых проектов;

2) *MongoDB.Driver.Core*: ядро драйвера *MongoDB.Driver*;

3) *MongoDB.Driver.GridFS*: пакет *GridFS*;

4) *MongoDB.Bson*: слой *BSON*, его можно использовать отдельно;

5) *mongocsharpdriver*: уровень совместимости для тех, кто обновляется с нашей серии 1.*x*. Нельзя использовать для новых проектов.

Рассмотрим, как выполнять базовые операции *CRUD* (создание, чтение, обновление, удаление).

В следующем примере показаны три способа подключения к серверу или серверам на локальном компьютере [11]:

```
// для прямого подключения к одному серверу MongoDB
// (это не приведет к автоматическому обнаружению первичного объекта,
//  даже если он является членом набора реплик)
var client = new MongoClient ();
// или используйте строку подключения
var client = new MongoClient ("mongodb: // localhost: 27017");
// или, чтобы подключиться к набору реплик, с автоматическим
// обнаружением основного, предоставить начальный список членов
var client = new MongoClient ("mongodb: // localhost: 27017, localhost: 27018, localhost: 27019");
```

Экземпляр *MongoClient* фактически представляет собой пул подключений к базе данных, понадобится только один экземпляр класса *MongoClient* даже при работе с несколькими потоками.

Чтобы получить базу данных, укажите имя базы данных методу *GetDatabase* на клиенте. Если базы данных еще нет, она будет создана при первом использовании:

var database = client.GetDatabase ("*foo*")

Теперь переменная базы данных содержит ссылку на базу данных *foo*.

Чтобы получить коллекцию для работы, укажите имя коллекции в методе *GetCollection <TDocument>* в базе данных. Если коллекции еще нет, она будет создана при первом использовании:

var collection = database.GetCollection <BsonDocument> ("*bar* ")

Переменная коллекции теперь содержит ссылку на коллекцию *bar* в базе данных *foo*.

При наличии экземпляра коллекции можно вставлять документы в коллекцию. Например, рассмотрим следующий документ *JSON*; документ содержит информацию о поле, эта информация является встроенным документом:

```
{
   "name": "MongoDB",
   "type": "database",
   "count": 1,
   "info": {
      x: 203,
      y: 102
   }
}
```

Для создания документа с помощью драйвера *.NET* используйте класс *BsonDocument*. Данный класс можно использовать и для создания встроенного документа:

var document = new BsonDocument

```
{
   { "name", "MongoDB" },
   { "type", "Database" },
   { "count", 1 },
   { "info", new BsonDocument
      {
         { "x", 203 },
         { "y", 102 }
      }}
};
```

Для вставки документа в коллекцию используйте методы *collection.InsertOne*(*document*) или *await collection.InsertOneAsync* (*document*).

Чтобы вставить несколько документов, можно использовать методы *InsertMany* или *InsertManyAsync*:

```
// генерируем 100 документов со счетчиком от 0 до 99:
var documents = Enumerable.Range(0, 100).Select(i => new BsonDocument("counter", i));
collection.InsertMany(documents);
await collection.InsertManyAsync(documents);
```

Теперь, когда вставлен 101 документ, можно проверить их наличие с помощью методов *CountDocuments* или *CountDocumentsAsync*. Следующий код должен установить значение *count*, равное 101:

```
var count = collection.CountDocuments(new BsonDocument());
```

или

```
var count = await collection.CountDocumentsAsync(new BsonDocument());
```

Используйте метод *Find* для запроса коллекции. Метод *Find* возвращает экземпляр *IFindFluent <TDocument*, *TProjection>*, который предоставляет удобный интерфейс для связывания параметров операции поиска.

Чтобы получить первый документ в коллекции, используйте методы *FirstOrDefault* или *FirstOrDefaultAsync*. *FirstOrDefault* возвраща-

ет первый документ или ноль. Это полезно для запросов, которые должны соответствовать только одному документу, или если вас интересует только первый документ.

В следующих примерах печатается первый найденный в коллекции документ:

```
var document = collection.Find(new BsonDocument()).FirstOrDefault();
Console.WriteLine(document.ToString());
```

или

```
var document = await collection.Find(new BsonDocument()).
FirstOrDefaultAsync();
Console.WriteLine(document.ToString());
```

В примере должен быть напечатан следующий документ:

```
{
  "_id": ObjectId("551582c558c7b4fbacf16735") },
  "name": "MongoDB",
  "type": "database",
  "count": 1,
  "info": { "x" : 203, "y" : 102 }
}
```

Чтобы получить все документы в коллекции, используйте методы *ToList* или *ToListAsync*. Это полезно, когда количество возвращаемых документов невелико:

```
var documents = collection.Find(new BsonDocument()).ToList();
```

или

```
var documents = await collection.Find(new BsonDocument()).
ToListAsync();
```

Если ожидается, что количество документов будет большим или документы будут обрабатываться итеративно, *ForEachAsync* осуществит обратный вызов для каждого возвращенного документа:

```
await collection.Find(new BsonDocument()).ForEachAsync(d => Console.WriteLine(d));
```

Для перебора возвращенных документов с помощью синхронного *API* используйте оператор *C# foreach* с методом адаптера *ToEnumerable*:

```
var cursor = collection.Find(new BsonDocument()).ToCursor();
foreach (var document in cursor.ToEnumerable())
{
   Console.WriteLine(document);
}
```

Можно создать фильтр для передачи методу *Find*, чтобы получить подмножество документов в нашей коллекции. Например, если бы мы хотели найти документ, для которого значение поля *i* равно 71, мы бы сделали следующее:

```
var filter = Builders<BsonDocument>.Filter.Eq("i", 71);
```

или

```
var document = collection.Find(filter).First();
Console.WriteLine(document);
```

или

```
var document = await collection.Find(filter).FirstAsync();
Console.WriteLine(document);
```

и он должен распечатать только один документ:

```
{ "_id" : ObjectId("5515836e58c7b4fbc756320b"), "i" : 71 }
```

Можно также получить комплект документов из нашей коллекции. Например, если мы хотим получить все документы, где $i > 50$, необходимо написать:

```
var filter = Builders<BsonDocument>.Filter.Gt("i", 50);
```

```
…
var cursor = collection.Find(filter).ToCursor();
foreach (var document in cursor.ToEnumerable())
{
    Console.WriteLine(document);
}
…
await collection.Find(filter).ForEachAsync(document => Console.WriteLine(document));
```

Для получения диапазона документов, например, 50 < *i* < = 100, необходимо написать:

```
var filterBuilder = Builders<BsonDocument>.Filter;
var filter = filterBuilder.Gt("i", 50) & filterBuilder.Lte("i", 100);
…
var cursor = collection.Find(filter).ToCursor();
foreach (var document in cursor.ToEnumerable())
{
    Console.WriteLine(document);
}
…
await collection.Find(filter).ForEachAsync(document => Console.WriteLine(document));
```

Можно добавить сортировку в поисковый запрос, используя метод *Sort*. Ниже используется метод построения фильтров *Exists* и метод построения сортировки по убыванию для сортировки документов:

```
var filter = Builders<BsonDocument>.Filter.Exists("i");
var sort = Builders<BsonDocument>.Sort.Descending("i");
…
var document = collection.Find(filter).Sort(sort).First();
…
var document = await collection.Find(filter).Sort(sort).FirstAsync();
```

Часто нам нужно изъять часть данных, содержащихся в документе. Построитель проекции поможет построить параметр проекции для операции поиска. Ниже мы исключим поле *_id* и выведем первый соответствующий документ:

```
var projection = Builders<BsonDocument>.Projection.Exclude-
("_id");
…
var document = collection.Find(new BsonDocument()).Project
(projection).First();
Console.WriteLine(document.ToString());
…
var document = await collection.Find(new BsonDocument()).-
Project(projection).FirstAsync();
Console.WriteLine(document.ToString());
```

MongoDB поддерживает множество операторов обновления. Чтобы обновить не более одного документа, используйте методы *UpdateOne* или *UpdateOneAsync*, чтобы указать фильтр и документ обновления. Здесь мы обновляем первый документ, который соответствует фильтру *i* = 10, и устанавливаем значение *i*, равное 110:

```
var filter = Builders<BsonDocument>.Filter.Eq("i", 10);
var update = Builders<BsonDocument>.Update.Set("i", 110);
…
collection.UpdateOne(filter, update);
…
await collection.UpdateOneAsync(filter, update);
```

Чтобы обновить все документы, соответствующие фильтру, используют методы *UpdateMany* или *UpdateManyAsync*. Например, увеличиваем значение *i* на 100:

```
var filter = Builders<BsonDocument>.Filter.Lt("i", 100);
var update = Builders<BsonDocument>.Update.Inc("i", 100);
…
var result = collection.UpdateMany(filter, update);
if (result.IsModifiedCountAvailable)
{
   Console.WriteLine(result.ModifiedCount);
}
…
var result = await collection.UpdateManyAsync(filter, update);
if (result.IsModifiedCountAvailable)
```

```
{
    Console.WriteLine(result.ModifiedCount);
}
```

Методы обновления возвращают *UpdateResult*, который предоставляет информацию об операции, включая количество документов, измененных обновлением.

Чтобы удалить не более одного документа, используют методы *DeleteOne* или *DeleteOneAsync*:

```
var filter = Builders<BsonDocument>.Filter.Eq("i", 110);
…
collection.DeleteOne(filter);
…
await collection.DeleteOneAsync(filter);
```

Чтобы удалить все документы, соответствующие фильтру, используют методы *DeleteMany* или *DeleteManyAsync*. Например, удаление всех документов, где $i >= 100$:

```
var filter = Builders<BsonDocument>.Filter.Gte("i", 100);
…
var result = collection.DeleteMany(filter);
Console.WriteLine(result.DeletedCount);
…
var result = await collection.DeleteManyAsync(filter);
Console.WriteLine(result.DeletedCount);
```

Методы удаления возвращают *DeleteResult*, который предоставляет информацию об операции, включая количество удаленных документов.

Есть два типа массовых операций:

1) заказанные массовые операции, выполняющие все операции по порядку и выдающие ошибку при первой ошибке;

2) неупорядоченные массовые операции, выполняющие все операции и сообщающие об ошибках, неупорядоченные массовые операции не гарантируют порядок выполнения.

Рассмотрим два простых примера с использованием упорядоченных и неупорядоченных операций:

```
var models = new WriteModel<BsonDocument>[]
{
   new InsertOneModel<BsonDocument>(new BsonDocument("_id",
4)),
   new InsertOneModel<BsonDocument>(new BsonDocument("_id",
5)),
   new InsertOneModel<BsonDocument>(new BsonDocument("_id",
6)),
   new UpdateOneModel<BsonDocument>(
      new BsonDocument("_id", 1),
      new BsonDocument("$set", new BsonDocument("x", 2))),
   new DeleteOneModel<BsonDocument>(new BsonDocument
("_id", 3)),
   new ReplaceOneModel<BsonDocument>(
      new BsonDocument("_id", 3),
      new BsonDocument("_id", 3).Add("x", 4))
};
…
// 1. Заказанная массовая операция – порядок работы гарантирован:
collection.BulkWrite (models);

// 2. Неупорядоченная массовая операция – нет гарантии порядка операций:
collection.BulkWrite (models, новый BulkWriteOptions {IsOrdered = false});
…
// 1. Заказанная массовая операция – порядок работы гарантирован:
await collection.BulkWriteAsync(models);

// 2. Неупорядоченная массовая операция – нет гарантии порядка операции:
await collection.BulkWriteAsync(models, new BulkWriteOptions { IsOrdered = false });
```

Рассмотрим выполнение некоторых административных функций. Для начала быстро подключим и создадим переменные клиента базы данных и коллекции для использования в примерах ниже:

```
var client = new MongoClient();
```

```
var database = client.GetDatabase("foo");
var collection = client.GetCollection<BsonDocument>("bar");
```

Можно вывести список всех баз данных с помощью методов *ListDatabases* или *ListDatabasesAsync*:

```
using (var cursor = client.ListDatabases())
{
   foreach (var document in cursor.ToEnumerable())
   {
      Console.WriteLine(document.ToString()));
   }
}
…
using (var cursor = await client.ListDatabasesAsync())
{
   await cursor.ForEachAsync(document => Console.WriteLine-
(document.ToString()));
}
```

Можно удалить базу данных с помощью методов *DropDatabase* или *DropDatabaseAsync*:

```
client.DropDatabase("foo");
…
await client.DropDatabaseAsync("foo");
```

Коллекция в *MongoDB* создается автоматически, используя методы *CreateCollection* или *CreateCollectionAsync*, можно создать коллекцию, чтобы настроить ее конфигурацию. Например, чтобы создать ограниченную коллекцию размером до 1 мегабайта:

```
var options = new CreateCollectionOptions {Capped = true, Max-
Size=1024*1024};
…
database.CreateCollection("cappedBar", options);
…
await database.CreateCollectionAsync("cappedBar", options);
```

Можно удалить коллекцию с помощью методов *DropCollection* или *DropCollectionAsync*:

```
database.DropCollection("cappedBar");
…
await database.DropCollectionAsync("cappedBar");
```

MongoDB поддерживает вторичные индексы. Чтобы создать индекс, требуется указать поле или комбинацию полей, и для каждого поля укажите направление индекса этого поля; 1 по возрастанию и -1 по убыванию. Следующее создает возрастающий индекс в поле *i*:

```
collection.Indexes.CreateOne(new BsonDocument("i", 1));
```

или

```
var keys = Builders<BsonDocument>.IndexKeys.Ascending("i");
collection.Indexes.CreateOne(keys);
…
await collection.Indexes.CreateOneAsync(new BsonDocument("i", 1));
```

или

```
var keys = Builders<BsonDocument>.IndexKeys.Ascending("i");
await collection.Indexes.CreateOneAsync(keys);
```

Используйте методы *List* или *ListAsync* для вывода списка индексов в коллекции:

```
using (var cursor = collection.Indexes.List())
{
   foreach (var document in cursor.ToEnumerable())
    {
       Console.WriteLine(document.ToString());
    }
}
…
using (var cursor = await collection.Indexes.ListAsync())
{
```

```
        await cursor.ForEachAsync(document => Console.WriteLine
(document.ToString()));
    }
```

В примере должны быть напечатаны следующие индексы:

```
{ "v" : 1, "key" : { "_id" : 1 }, "name" : "_id_", "ns" : "mydb.test" }
{ "v" : 1, "key" : { "i" : 1 }, "name" : "i_1", "ns" : "mydb.test" }
```

MongoDB также предоставляет текстовые индексы для поддержки поиска строкового содержимого. Текстовые индексы могут включать любое поле, значение которого является строкой или массивом строковых элементов. Чтобы создать текстовый индекс, укажите строковый литерал «текст» в индексном документе:

```
collection.Indexes.CreateOne(new BsonDocument("content", "text"));
```

или

```
var keys = Builders<BsonDocument>.IndexKeys.Text("content");
collection.Indexes.CreateOne(keys);
…
await collection.Indexes.CreateOneAsync(new BsonDocument("content",
"text"));
```

или

```
var keys = Builders<BsonDocument>.IndexKeys.Text("content");
await collection.Indexes.CreateOneAsync(keys);
```

С версии *MongoDB* 2.6 текстовые индексы были интегрированы в основной язык запросов и были включены по умолчанию:

```
// вставляем несколько документов:
collection.InsertMany(new []
{
    new BsonDocument("_id", 0).Add("content", "textual content"),
    new BsonDocument("_id", 1).Add("content", "additional content"),
    new BsonDocument("_id", 2).Add("content", "irrelevant content"),
```

```
});
// находим их по текстовому индексу:
var filter = Builders<BsonDocument>.Filter.Text("textual content -irrelevant");
var matchCount = collection.CountDocuments(filter);
Console.WriteLine("Text search matches: {0}", matchCount);
// находим их по текстовому индексу с оператором $language
var englishFilter = Builders<BsonDocument>.Filter.Text("textual content -irrelevant", "english");
var matchCount = collection.CountDocuments(filter);
Console.WriteLine("Text search matches (english): {0}", matchCount);
// находим наиболее результативное совпадение:
var projection = Builders<BsonDocument>.Projection.MetaTextScore("score");
var doc = collection.Find(filter).Project(projection).First();
Console.WriteLine("Highest scoring document: {0}", doc);
…
// вставляем несколько документов:
await collection.InsertManyAsync(new []
{
    new BsonDocument("_id", 0).Add("content", "textual content"),
    new BsonDocument("_id", 1).Add("content", "additional content"),
    new BsonDocument("_id", 2).Add("content", "irrelevant content"),
});
// находим их по текстовому индексу:
var filter = Builders<BsonDocument>.Filter.Text("textual content -irrelevant");
var matchCount = await collection.CountDocumentsAsync(filter);
Console.WriteLine("Text search matches: {0}", matchCount);
// находим их по текстовому индексу с оператором $language
var englishFilter = Builders<BsonDocument>.Filter.Text("textual content -irrelevant", "english");
var matchCount = await collection.CountDocumentsAsync(filter);
Console.WriteLine("Text search matches (english): {0}", matchCount);
// находим наиболее результативное совпадение:
var projection = Builders<BsonDocument>.Projection.MetaTextScore("score");
```

```
var doc = await collection.Find(filter).Project(projection).First
Async();
Console.WriteLine("Highest scoring document: {0}", doc);
```

Он должен напечатать:

```
Text search matches: 2
Text search matches (english): 2
Highest scoring document: {"_id" :1,"content":"additional content",
"score":0.75}
```

Можно запустить любую команду с помощью методов *RunCommand* или *RunCommandAsync*. Например, используя команду *buildInfo*:

```
var buildInfoCommand = new BsonDocument("buildinfo", 1);
…
var result = database.RunCommand<BsonDocument>(buildInfo-
Command);
…
var result = await database.RunCommandAsync<BsonDocument>
(buildInfoCommand);
```

Библиотека *.NET BSON* поддерживает запись документов *JSON* с помощью класса *JsonWriter*.

Приведенные ниже программы экспортируют все документы из коллекции в файл по одному документу в строке. Существует две версии программы: одна использует синхронный *API*, а другая – асинхронный *API*.

Содержание коллекции:

```
db.mydata.find()
{ "_id" : ObjectId("5513306a2dfd32ffd580e323"), "x" : 1.0 }
{ "_id" : ObjectId("5513306c2dfd32ffd580e324"), "x" : 2.0 }
{ "_id" : ObjectId("5513306e2dfd32ffd580e325"), "x" : 3.0 }
{ "_id" : ObjectId("551330712dfd32ffd580e326"), "x" : 4.0 }
```

Синхронная версия программы:

```
using MongoDB.Bson;
using MongoDB.Bson.IO;
```

```
using MongoDB.Bson.Serialization;
using MongoDB.Driver;

// ...
string outputFileName; // initialize to the output file
IMongoCollection<BsonDocument> collection; // initialize to the collection to read from
using (var streamWriter = new StreamWriter(outputFileName))
{
   var cursor = collection.Find(new BsonDocument()).ToCursor();
   foreach (var document in cursor.ToEnumerable())
   {
      using (var stringWriter = new StringWriter())
      using (var jsonWriter = new JsonWriter(stringWriter))
      {
         var context = BsonSerializationContext.CreateRoot(jsonWriter);
         collection.DocumentSerializer.Serialize(context, document);
         var line = stringWriter.ToString();
         streamWriter.WriteLine(line);
      }
   }
}
```

Асинхронная версия программы:

```
using MongoDB.Bson;
using MongoDB.Bson.IO;
using MongoDB.Bson.Serialization;
using MongoDB.Driver;
// ...
string outputFileName;
 // initialize to the output file
IMongoCollection<BsonDocument> collection;
 // initialize to the collection to read from
using (var streamWriter = new StreamWriter(outputFileName))
{
   await collection.Find(new BsonDocument())
      .ForEachAsync(async (document) =>
      {
```

```
            using (var stringWriter = new StringWriter())
            using (var jsonWriter = new JsonWriter(stringWriter))
            {
             var context = BsonSerializationContext.CreateRoot(jsonWriter);
               collection.DocumentSerializer.Serialize(context, docu-
ment);
               var line = stringWriter.ToString();
               await streamWriter.WriteLineAsync(line);
            }
        });
    }
```

Выходной файл должен выглядеть так:

```
{ "_id" : ObjectId("5513306a2dfd32ffd580e323"), "x" : 1.0 }
{ "_id" : ObjectId("5513306c2dfd32ffd580e324"), "x" : 2.0 }
{ "_id" : ObjectId("5513306e2dfd32ffd580e325"), "x" : 3.0 }
{ "_id" : ObjectId("551330712dfd32ffd580e326"), "x" : 4.0 }
```

Библиотека *.NET BSON* поддерживает чтение документов *JSON* с помощью класса *JsonReader*.

Приведенные ниже программы импортируют все документы из файла по одному документу в строке в коллекцию. Существует две версии программы: одна использует синхронный *API*, а другая – асинхронный *API*.

Содержимое входного файла:

```
{ "_id" : ObjectId("5513306a2dfd32ffd580e323"), "x" : 1.0 }
{ "_id" : ObjectId("5513306c2dfd32ffd580e324"), "x" : 2.0 }
{ "_id" : ObjectId("5513306e2dfd32ffd580e325"), "x" : 3.0 }
{ "_id" : ObjectId("551330712dfd32ffd580e326"), "x" : 4.0 }
```

Синхронная версия программы:

```
using MongoDB.Bson;
using MongoDB.Bson.IO;
using MongoDB.Bson.Serialization;
using MongoDB.Driver;
// ...
```

```
string inputFileName; // initialize to the input file
IMongoCollection<BsonDocument> collection; // initialize to the collection to write to.
using (var streamReader = new StreamReader(inputFileName))
{
   string line;
   while ((line = streamReader.ReadLine()) != null)
   {
      using (var jsonReader = new JsonReader(line))
      {
         var context = BsonDeserializationContext.CreateRoot(jsonReader);
         var document = collection.DocumentSerializer.Deserialize(context);
         collection.InsertOne(document);
      }
   }
}
```

Асинхронная версия программы:

```
using MongoDB.Bson;
using MongoDB.Bson.IO;
using MongoDB.Bson.Serialization;
using MongoDB.Driver;
// ...
string inputFileName; // initialize to the input file
IMongoCollection<BsonDocument> collection; // initialize to the collection to write to.
using (var streamReader = new StreamReader(inputFileName))
{
   string line;
   while ((line = await streamReader.ReadLineAsync()) != null)
   {
      using (var jsonReader = new JsonReader(line))
      {
       var context = BsonDeserializationContext.CreateRoot(jsonReader);
         var document = collection.DocumentSerializer.Deserialize(context);
```

```
            await collection.InsertOneAsync(document);
        }
    }
}
```

Содержимое коллекции должно выглядеть так:

```
db.mydata.find()
{ "_id" : ObjectId("5513306a2dfd32ffd580e323"), "x" : 1.0 }
{ "_id" : ObjectId("5513306c2dfd32ffd580e324"), "x" : 2.0 }
{ "_id" : ObjectId("5513306e2dfd32ffd580e325"), "x" : 3.0 }
{ "_id" : ObjectId("551330712dfd32ffd580e326"), "x" : 4.0 }
```

Глава 6. СУБД M*ongoDB* и *JavaScript*, *Java*

6.1. СУБД *MongoDB* и *JavaScript*

Рассмотрим применение программных средств реализации нереляционных моделей баз данных на примере СУБД *MongoDB* и *JavaScript*. Для этого будем использовать *Node.js*, представляющий среду выполнения кода на *JavaScript*, которая построена на основе движка *JavaScript Chrome V*8, позволяющая транслировать вызовы на языке *JavaScript* в машинный код. *Node.js*, прежде всего, предназначен для создания серверных приложений на языке *JavaScript*.

Для работы необходимо установить сервер *MongoDB*. Подробнее как это сделать, описывается в предыдущих главах. Кроме самого сервера *Mongo* для взаимодействия с *Node.js* необходим драйвер.

При подключении и взаимодействии с базой данных в *MongoDB* можно выделить следующие этапы:

1) подключение к серверу;

2) получение объекта базы данных на сервере;

3) получение объекта коллекции в базе данных;

4) взаимодействие с коллекцией (добавление, удаление, получение, изменение данных).

Для создания нового проекта определим новый каталог, который будет называться *mongoapp*. Далее определим в этом каталоге новый файл *package.json*:

```
{
 "name": "mongoapp",
 "version": "1.0.0",
 "dependencies": {
    "express": "^4.17.0",
    "mongodb": "^4.1.0"
  }
}
```

Основным классом для работы с *MongoDB* является класс *MongoClient*: через него будет идти все взаимодействия с хранилищем данных. Соответственно, вначале мы должны получить *MongoClient*:

```
const MongoClient = require("mongodb").MongoClient;
```

Для подключения к серверу *mongodb* применяется метод *connect*():

```
const MongoClient = require("mongodb").MongoClient;
// создаем объект MongoClient и передаем ему строку подключения:
const mongoClient = new MongoClient("mongodb://localhost:27017/");
mongoClient.connect(function(err, client){
    if(err){
      return console.log(err);
    }
    // взаимодействие с базой данных:
    client.close();
});
```

Создадим объект *MongoClient*. Для этого в его конструктор передается два параметра. Первый параметр – это адрес сервера. В качестве протокола адреса устанавливается *mongodb*://. На локальной машине адресом будет *localhost*, после которого указывается номер порта. По умолчанию номер порта 27017.

С помощью метода *connect* происходит подключение к серверу. В качестве параметра метод принимает функцию обратного вызова, который срабатывает при установке подключения. Это функция принимает два параметра: *err* (возникшая ошибка при установке соединения) и *client* (ссылка на подключенный к серверу клиент).

Если при подключении возникла ошибка, то мы можем использовать значение *err* для ее получения. Если же ошибки нет, то мы можем взаимодействовать с сервером через объект *client*. В завершение работы с базой данных нам нужно закрыть соединение с помощью метода *client.close*().

Выше был рассмотрен пример с использованием функций обратного вызова. Но *node.js* также позволяет использовать другой подход – *async/await*:

```
const MongoClient = require("mongodb").MongoClient;
const url = "mongodb://localhost:27017/";
const mongoClient = new MongoClient(url);
async function run() {
```

```
    try {
        // подключаемся к серверу:
        await mongoClient.connect();
        // взаимодействие с базой данных:
    }catch(err) {
        console.log(err);
    } finally {
// закрываем подключение при завершении работы или при ошибке:
        await mongoClient.close();
    }
}
run();
```

Получив объект подлюченного клиента, мы можем обращаться к базе данных на сервере. Для этого используется метод *const db = client.db* («название_бд»). В качестве параметра в метод передается название базы данных, к которой хотим подключиться. Результатом функции является объект базы данных, через который сможем обращаться к данным в этой базе данных.

Для примера выполним *ping* к базе данных *admin*, которая по умолчанию есть на сервере *MongoDB*:

```
const MongoClient = require("mongodb").MongoClient;
    const url = "mongodb://localhost:27017/";
const mongoClient = new MongoClient(url);
  // подключаемся к серверу:
mongoClient.connect(function(err, client){
          // обращаемся к базе данных admin:
    const db = client.db("admin");

    db.command({ping: 1}, function(err, result){
        if(!err){
   console.log("Подключение с сервером успешно установлено");
            console.log(result);
        }
        // закрываем подключение:
        client.close();
        console.log("Подключение закрыто");
```

```
    });
});
```

Для выполнения диагностических команд типа пинга объект базы данных *Db* предоставляет метод *command*, который в качестве первого параметра принимает объект диагностики. Например, объект {*ping*: 1} указывает на то, что нужно выполнить пинг к базе данных, то есть проверить подключение к ней.

В качестве второго параметра передается функция обратного вызова с двумя параметрами. В функции обратного вызова можно проверить наличие ошибки, а при ее отстутствии – вывести информацию об успешном подключении и полученный результат:

```
if(!err){
    console.log("Подключение с сервером успешно установлено");
    console.log(result);
}
```

Все данные в *MongoDB* попадают в коллекции. В рамках *node.js* для взаимодействия с базой данных (добавления, удаления, чтения данных) нам нужно получить объект коллекции, то есть применить метод *db.collection* («название_коллекции»), в который передается название коллекции и который возвращает объект класса *Collection*, позволяющий через него взаимодействовать с этой коллекцией.

В коллекциях *MongoDB* данные хранятся в виде документов. Например, обратимся к коллекции в базе данных. Определим в каталоге проекта следующий файл *app.js*:

```
const MongoClient = require("mongodb").MongoClient;
const url = "mongodb://localhost:27017/";
const mongoClient = new MongoClient(url);
mongoClient.connect(function(err, client){
      const db = client.db("usersdb");
    const collection = db.collection("users");
    collection.countDocuments(function(err, result){
          if(err){
        return console.log(err);
      }
      console.log(`В коллекции users ${result} документов`);
```

```
        client.close();
    });
});
```

В качестве базы данных здесь используется *usersdb*. При этом не важно, что по умолчанию на сервере *MongoDB* нет подобной базы данных. При первом обращении к ней сервер автоматически ее создаст.

Получив коллекцию, мы можем использовать ее методы. В данном случае для простоты применяется метод *countDocuments*(), который получает количество документов коллекции. Этот метод имеет два параметра – сам добавляемый объект и функция обратного вызова, которая выполняется после добавления. В этой функции применяются два параметра: *err* и *result*.

Для добавления одного объекта применяют метод *insertOne*(), для дополнения набора объектов – метод *insertMany*().

Для получения данных из коллекции применяют метод *find*(). Метод *find* возвращает специальный объект *FindCursor*, и, чтобы получить все данные у этого объекта, используют метод *toArray*(). В метод передается функция обратного вызова со стандартными параметрами.

С помощью метода *find*() мы можем дополнительно отфильтровать извлекаемые документы. Метод *findOne*() работает аналогично, только позволяет получить один документ.

Удалять документы в *MongoDB* можно различными способами:

- *deleteMany*(): удаляет все документы, которые соответствуют определенному критерию;
- *deleteOne*(): удаляет один документ, который соответствует определенному критерию;
- *findOneAndDelete*(): получает и удаляет один документ, который соответствует определенному критерию;
- *drop*(): удаляет всю коллекцию.

Node.js использует модульную систему. Модуль представляет собой блок кода, который может использоваться повторно в других модулях. При необходимости можно подключать нужные модули. Какие встроенные модули есть в *node.js* и какую функциональность они предоставляют, можно узнать из [2].

Для загрузки модулей применяется функция *require*(), в которую передается название модуля. Например, для получения и обработки запроса необходим модуль *http*: *const http = require*("*http*");».

Аналогичным способом можно загружать и использовать другие

встроенные модули. Например, модуль *os*, который предоставляет информацию об окружении и операционной системе:

```
const os = require("os");
// получим имя текущего пользователя
let userName = os.userInfo().username;
 console.log(userName);
```

Система *Node.js* не ограничена встроенными модулями, и при необходимости есть возможность создать модули пользователя.

Для работы с сервером и протоколом *http* в *Node.js* используется модуль *http*. Чтобы создать сервер, нужно использовать метод *http.createServer*():

```
const http = require("http");
 http.createServer().listen(3000);
```

Метод *createServer*() возвращает объект *http.Server*. Чтобы сервер мог прослушивать и обрабатывать входящие подключения, у объекта сервера необходимо использовать метод *listen*(), в который в качестве параметра передается номер порта, по которому запускается сервер.

Для обработки подключений в метод *createServer* можно передать специальную функцию:

```
const http = require("http");
 http.createServer(function(request, response){
        response.end("Hello world!");
}).listen(3000);
```

Эта функция принимает два параметра:

- *request*: хранит информацию о запросе;
- *response*: управляет отправкой ответа.

Параметр *request* позволяет получить информацию о запросе и представить объект *http.IncomingMessage*. Основные свойства этого объекта:

- *headers*: возвращает заголовки запроса;
- *method*: тип запроса (*GET*, *POST*, *DELETE*, *PUT*);
- *url*: представляет запрошенный адрес.

Параметр *response* управляет отправкой ответа и представляет объект *http.ServerResponse*. Среди его функциональности можно выделить следующие методы:

- *statusCode*: устанавливает статусный код ответа;
- *statusMessage*: устанавливает сообщение, отправляемое вместе со статусным кодом;
- *setHeader*(*name*, *value*): добавляет в ответ один заголовок;
- *write*: пишет в поток ответа некоторое содержимое;
- *writeHead*: добавляет в ответ статусный код и набор заголовков;
- *end*: сигнализирует серверу, что заголовки и тело ответа установлены, в итоге ответ отсылается клиенту. Данный метод должен вызываться в каждом запросе.

По умолчанию *Node.js* не имеет встроенной системы маршрутизации. Обычно она реализуется с помощью специальных фреймворках типа *Express*. Однако при необходимости разграничить простейшую обработку нескольких маршрутов вполне можно использовать свойство *url* объекта *Request*. Например:

```
const http = require("http");
 http.createServer(function(request, response){
    response.setHeader("Content-Type", "text/html; charset=utf-8;");
       if(request.url === "/home" || request.url === "/"){
      response.write("<h2>Home</h2>");
   }
   else if(request.url == "/about"){
      response.write("<h2>About</h2>");
   }
   else if(request.url == "/contact"){
      response.write("<h2>Contacts</h2>");
   }
   else{
      response.write("<h2>Not found</h2>");
   }
   response.end();
}).listen(3000);
```

В данном случае обрабатываются три маршрута. Если идет обращение к корню сайта или по адресу *localhost:*3000*/home*, то пользователю выводится строка *Home*. Ели обращение идет по адре-

су *localhost:*3000*/about*, то пользователю в браузере отображается строка *About*. Если запрошенный адрес не соответствует ни одному маршруту, то выводится заголовок *Not Found*.

Переадресация предполагает отправку статусного кода 301 (постоянная переадресация) или 302 (временная переадресация) и заголовка *Location*, который указывает на новый адрес. Например, выполним переадресацию с адреса *localhost:*3000*/* на адрес *localhost:*-3000*/newpage*.

```
const http = require("http");
 http.createServer(function(request, response){
   response.setHeader("Content-Type", "text/html; charset=utf-8;");
      if(request.url === "/"){
      response.statusCode = 302; // временная переадресация
      // на адрес localhost:3000/newpage
      response.setHeader("Location", "/newpage")
   }
   else if(request.url == "/newpage"){
      response.write("New address");
   }
   else{
      response.statusCode = 404; // адрес не найден
      response.write("Not Found");
   }
   response.end();
}).listen(3000);
```

Отправка статических файлов – частая задача в построении и функционировании веб-приложеия. Рассмотрим, как отправлять файлы в приложении на *Node.js*.

Пусть в каталоге проекта у нас будут три файла:

1) *app.js*;

2) *about.html*;

3) *index.html*.

Наряду с файлом приложения *app.js* определим два *html*-файла. В файле *index.html* определим следующий код:

```
<!DOCTYPE html>
<html>
<head>
```

```
    <title>Главная</title>
    <meta charset="utf-8" />
</head>
<body>
    <h1>Главная</h1>
</body>
<html>
```

Аналогично определим код в файле *about.html*:

```
<!DOCTYPE html>
<html>
<head>
    <title>О сайте</title>
    <meta charset="utf-8" />
</head>
<body>
    <h1>О сайте</h1>
</body>
<html>
```

Задача состоит в том, чтобы отправить их содержимое пользователю. Для считывания файла может применяться метод *fs.createReadStream*(), который считывает файл в поток; затем с помощью метода *pipe*() можно связать считанные файлы с потоком записи, то есть объектом *response*. Поместим в файл *app.js* следующий код:

```
const fs = require("fs");
  http.createServer(function(request, response){
        console.log(`Запрошенный адрес: ${request.url}`);
    // получаем путь после слеша:
    const filePath = request.url.substr(1);
    // смотрим, есть ли такой файл:
    fs.access(filePath, fs.constants.R_OK, err => {
        // если произошла ошибка - отправляем статусный код 404:
        if(err){
            response.statusCode = 404;
            response.end("Resourse not found!");
```

```
        }
        else{
            fs.createReadStream(filePath).pipe(response);
        }
    });
}).listen(3000, function(){
    console.log("Server started at 3000");
});
```

Вначале получаем запрошенный адрес. Например, запрошенный адрес будет соответствовать напрямую пути к файлу на сервере. Затем с помощью асинхронной функции *fs.access* проверяем доступность файла для чтения. Первый параметр функции – путь к файлу. Второй параметр – опция, относительно которой проверяется доступ. В данном случае значение *fs.constants.R_OK* говорит о том, что проверяются права на чтение из файла. Третий параметр функции – функция обратного вызова, которая получает объект ошибки. Если произошла ошибка, посылается статусный код 404.

Для отправки файла применяется цепочка методов *fs.createReadStream*("*some.doc*").*pipe*(*response*).

Метод *fs.createReadStream*("*some.doc*") создает поток для чтения – объект *fs.ReadStream*. Для получения данных из потока вызывается метод *pipe*(), в который передается объект интерфейса *stream.Writable* или поток для записи: объект *http.ServerResponse*, который реализует этот интерфейс.

Запустим приложение и в браузере обратимся по адресу *http://localhost:*3000/*index.html* (рис. 6.1).

Для дальнейшей работы нужен фреймворк *Express*, который сам использует модуль *http*, но вместе с тем предоставляет ряд готовых абстракций, которые упрощают создание сервера и серверной логики, в частности, обработка отправленных форм, работа с куками, *CORS* и т. д. Исходный код фреймворка *Express* можно посмотреть в репозитории на гитхабе по адресу *https://github.com/expressjs/express*.

Объединим в одном приложении обработки запросов с помощью *Express* и работу с данными в *MongoDB*. Для этого определим следующий файл приложения *app.js* (Приложение 2).

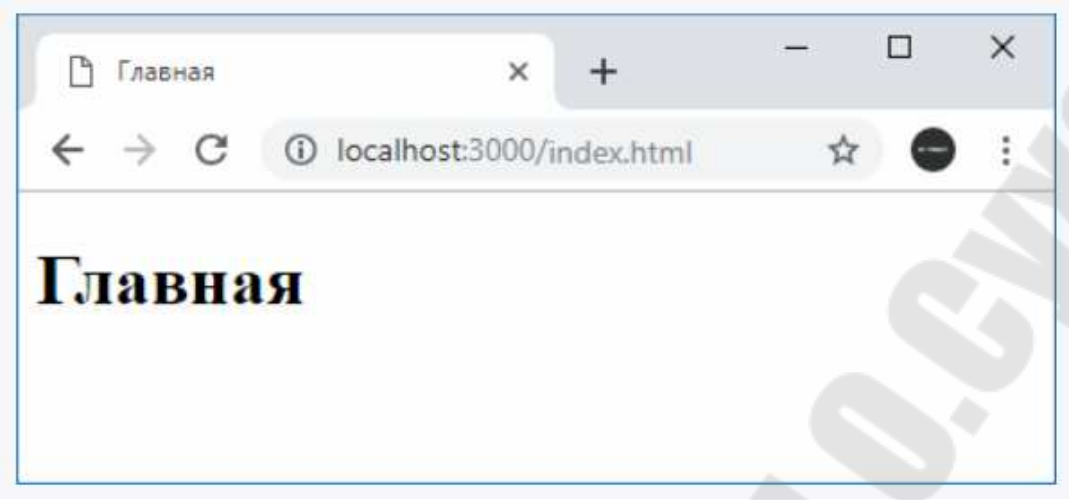

Рис. 6.1. Окно приложения

Для каждого типа запросов здесь определен свой обработчик *Express.* И в каждом из обработчиков мы обращаемся к базе данных. Чтобы не открывать и не закрывать подключение при любом запросе, мы открываем подключение в самом начале и только после открытия подключения запускаем прослушивание входящих запросов:

```
<font style="vertical-align: inherit;"><font style="vertical-align: inherit;">mongoClient.connect (function (err, client) {</font></font><font></font><font style="vertical-align: inherit;"><font style="vertical-align: inherit;">
    if (err) return console.log (err);</font></font><font></font><font style="vertical-align: inherit;"><font style="vertical-align: inherit;">
    dbClient = client;</font></font><font></font><font style="vertical-align: inherit;"><font style="vertical-align: inherit;">
    app.locals.collection = client.db ("usersdb"). collection ("users");</font></font><font></font><font style="vertical-align: inherit;"><font style="vertical-align: inherit;">
    app.listen (3000, function () {</font></font><font></font><font style="vertical-align: inherit;"><font style="vertical-align: inherit;">
        console.log ("Сервер ожидает подключения ...");</font></font><font></font><font style="vertical-align: inherit;"><font style="vertical-align: inherit;">
    });</font></font><font></font><font style="vertical-align: inherit;"><font style="vertical-align: inherit;">
});</font></font><font></font>
```

Взаимодействие будет идти с коллекцией пользователя – получаем ссылку на эту коллекцию в локальную переменную приложения *app.locals.collection*. Далее через эту переменную мы сможем получить доступ к коллекции в любом месте приложения.

В конце работы скрипта мы можем закрыть подключение, сохраненное в переменную *db*:

```
process.on("SIGINT", () => {<font></font>
   dbClient.close();<font></font>
   process.exit();<font></font>
});<font></font>
```

В данном случае мы прослушиваем событие *SIGINT*, которое запускается при нажатии комбинации *CTRL+C* в консоли, что завершает выполнение программы.

Когда приходит *GET*-запрос к приложению, то возвращаем в ответ клиенту все из базы данных:

```
app.get("/api/users", function(req, res){<font></font>
     <font></font>
   const collection = req.app.locals.collection;<font></font>
   collection.find({}).toArray(function(err, users){<font></font>
      <font></font>
      if(err) return console.log(err);<font></font>
      res.send(users)<font></font>
   });<font></font>
   <font></font>
});<font></font>
```

Если в *GET*-запросе передается параметр *id*, то возвращается только один пользователь из базы данных по этому *id*:

```
app.get("/api/users/:id", function(req, res){<font></font>
     <font></font>
   const id = new objectId(req.params.id);<font></font>
   const collection = req.app.locals.collection;<font></font>
   collection.findOne({_id: id}, function(err, user){<font></font>
         <font></font>
      if(err) return console.log(err);<font></font>
```

```
        res.send(user);<font></font>
    });<font></font>
});<font></font>
```

Когда приходит *POST*-запрос, с помощью парсера *jsonParser* получаем отправленные данные и по ним создаем объект, который добавлен в базу данных посредством метода *insertOne*():

```
app.post("/api/users", jsonParser, function (req, res) {<font></font>
    <font></font>
    if(!req.body) return res.sendStatus(400);<font></font>
    <font></font>
    const userName = req.body.name;<font></font>
    const userAge = req.body.age;<font></font>
    const user = {name: userName, age: userAge};<font></font>
    <font></font>
    const collection = req.app.locals.collection;<font></font>
    collection.insertOne(user, function(err, result){<font></font>
        <font></font>
        if(err) return console.log(err);<font></font>
        res.send(user);<font></font>
    });<font></font>
});<font></font>
```

При получении запроса *PUT* также получаем отправленные данные и с помощью метода *findOneAndUpdate*() обновляемые данные в базе данных.

И в методе *app.delete*(), который срабатывает при получении запроса *DELETE*, применяем метод *findOneAndDelete*() для удаления данных.

Теперь создадим в папке проекта новый каталог *public* и определим в этом каталоге файл *index.html* (Приложение 3).

Для упрощения запроса в данном случае применяется библиотека *jquery*.

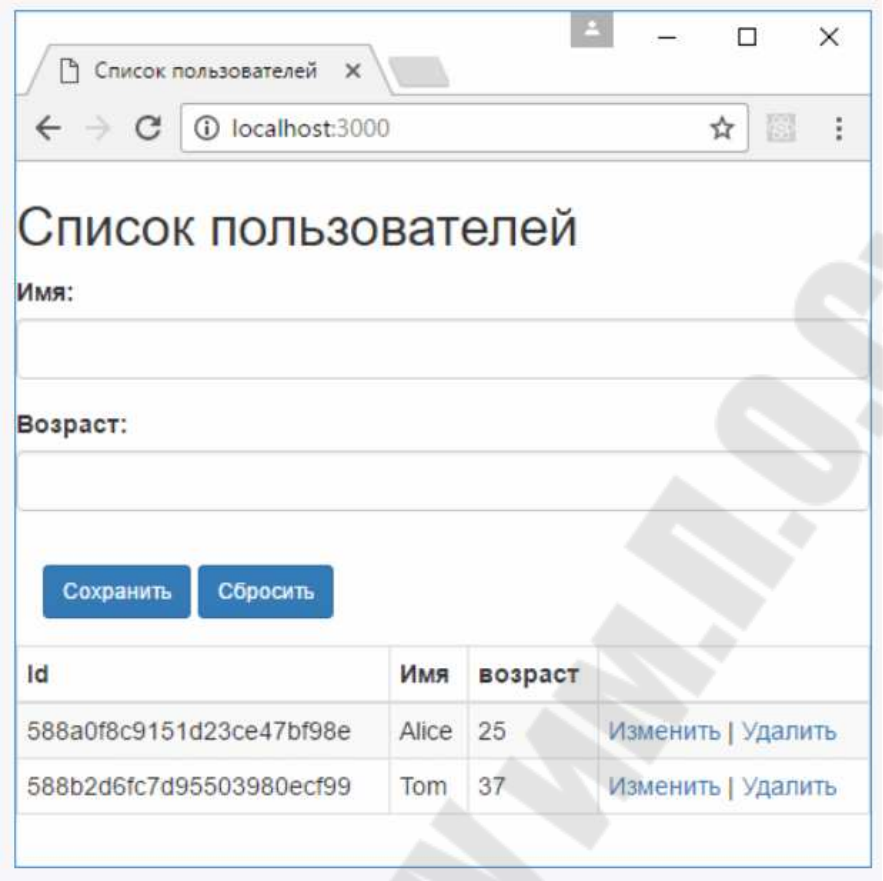

Рис. 6.2. Окно приложения с базой данных

Запустим приложение, обратимся к приложению по адресу *http*: // *localhost*:3000, мы получим возможность управлять пользователями, которые хранятся в базе данных *MongoDB* (рис. 6.2).

6.2. СУБД *MongoDB* и *Java*

Для создания приложения на языке программирования *Java* нужно установить драйвер *MongoDB Java*, который предназначен для синхронной обработки потоков. Если требуется приложение с асинхронной обработкой потоков и реактивными потоками, необходимо использовать драйвер реактивных потоков.

Покажем, как создать приложение, использующее драйвер *Java* для подключения к кластеру *MongoDB Atlas*. Драйвер *Java* позволяет подключаться и взаимодействовать с кластерами *MongoDB* из приложения *Java*.

MongoDB Atlas – это полностью управляемая облачная служба баз данных, в которой данные размещаются в кластерах *MongoDB*.

Выполним следующие действия, чтобы подключить приложение *Java* к кластеру *MongoDB Atlas*.

Для настройки разрабатываемого проекта необходимо установить *Java Development Kit* (*JDK*). Желательно, чтобы в системе был установлен *JDK 8* или более поздняя версия.

Рассмотрим, как добавить зависимости драйвера *Java MongoDB* с помощью *Maven* или *Gradle*. Для разработки проекта можно использовать интегрированную среду разработки (*IDE*), такую как *Intellij IDEA* или *Eclipse IDE*, чтобы было удобнее настраивать *Maven* или *Gradle* для сборки и запуска вашего проекта.

Если вы используете *Maven*, добавьте следующее в список зависимостей *pom.xml*:

<dependencees> <dependency>
<groupId>org.mongodb</groupId> <artifactId>mongodb-driver-sync</artifactld> <version>4.3.2</version> </dependency> </dependencies>

Если вы используете *Gradle*, добавьте в список зависимостей *build.gradle* следующее:

dependencies {
compile 'org.mongodblmongodb-driver-sync:4.3.2' }

После настройки зависимостей убедитесь, что они доступны для вашего проекта, это может потребовать запуск вашего диспетчера зависимостей и обновление проекта в вашей среде *IDE*.

После настройки зависимостей проекта *Java* необходимо создать кластер *MongoDB* для хранения и управления данными. Выполните руководство по началу работы с *Atlas*, чтобы настроить новую учетную запись *Atlas*, создать и запустить кластер *MongoDB* бесплатного уровня, загрузить наборы данных и взаимодействовать с данными.

После выполнения шагов, описанных в руководстве по Атласу, вы увидите новый кластер *MongoDB*, развернутый в Атласе, нового пользователя базы данных и образцы наборов данных, загруженные в ваш кластер.

Необходимо передать инструкции драйверу о том, как подключиться к кластеру *MongoDB* в строке, называемой строкой подключения. Эта строка включает информацию об имени хоста или *IP*-адресе и порте вашего кластера, механизме аутентификации, учетных данных пользователя, если применимо, и других вариантах подключения.

Для получения строки подключения для кластера и пользователя, созданного на предыдущем шаге, войдите в свою учетную запись *Atlas*, перейдите в раздел «Кластеры» и нажмите кнопку «Подключить» для кластера, к которому вы хотите подключиться, как показано на рис. 6.3.

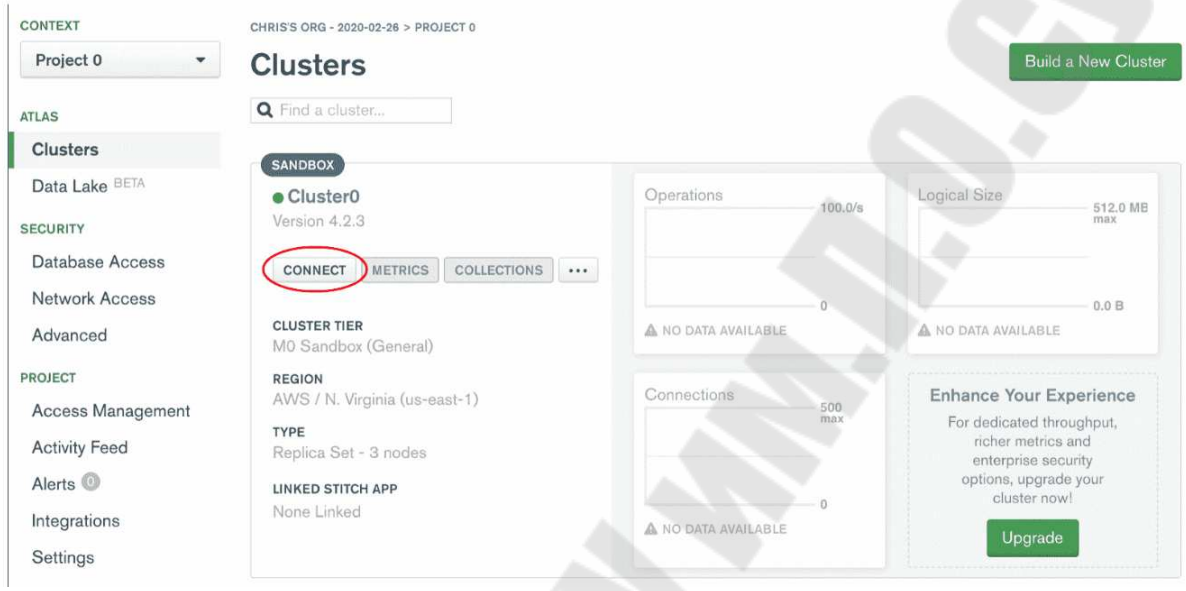

Рис. 6.3. Окно подключения кластера

Далее следует перейти к этапу «Подключить приложение» и выбрать драйвер *Java*, версию «4.1 или новее»; щелкнуть значок «Копировать», чтобы скопировать строку подключения в буфер обмена, как показано на рис. 6.4.

Создайте файл *QuickStart.java*, содержащий код приложения в каталоге базового пакета вашего проекта. Используйте следующий пример кода, чтобы выполнить запрос к вашему набору данных в *MongoDB Atlas*, заменив значение переменной *uri* строкой подключения *MongoDB Atlas*.

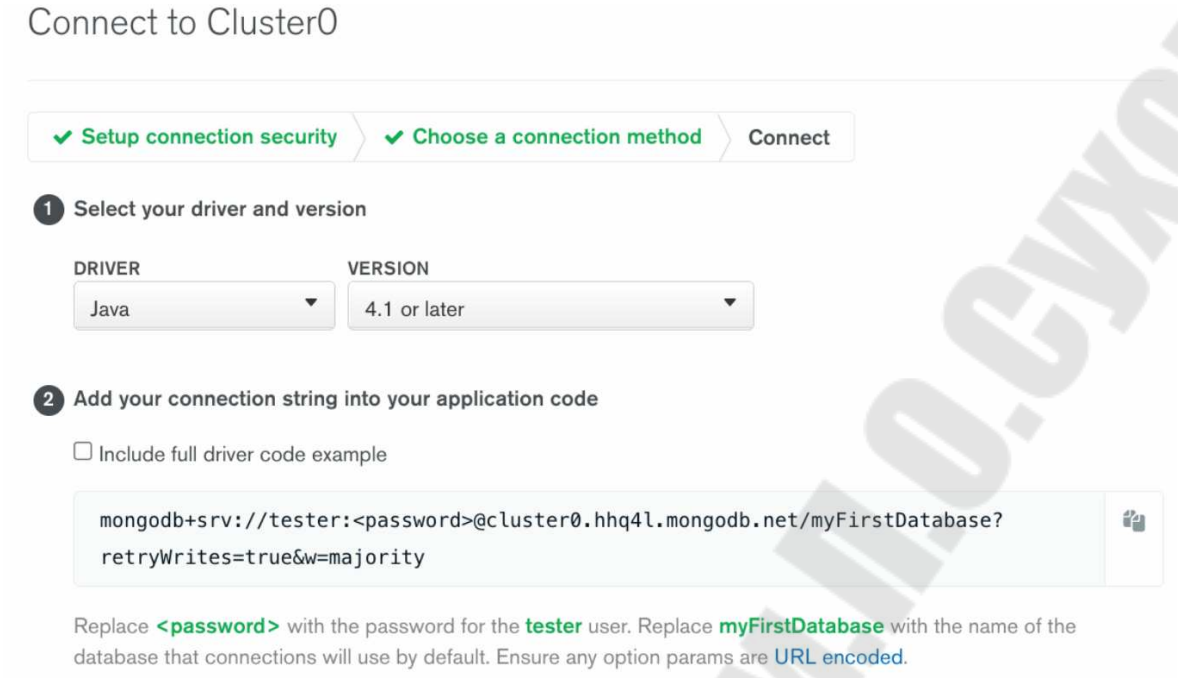

Рис. 6.4. Окно подключения кластера с изменением строки подключения

Необходимо заменить раздел *<password>* в строке подключения паролем, который вы создали для своего пользователя с разрешениями *atlasAdmin*:

```
import static com.mongodb.client.model.Filters.eq;
import org.bson.Document;
import com.mongodb.client.MongoCT.ient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
public static void main( String[] args ) {
// Замените строку uri на строку подключения вашего развертывания MongoDB:
String uri = "<connection string uri>";
try (MongoClient mongoClient = MongoClients.create(uri)) {
MongoDatabase database = mongoClient.getDatabase ("sample_mflix"); MongoCollection<Oocument> collection = database.getCollection("movies");
Document doc = collection.find(eq("title", "Back to the Future")).first(); System.out.println(doc.toJsonQ);
}
}
```

Когда вы запустите класс *QuickStart*, он должен вывести детали фильма из образца набора данных, которые будут выглядеть примерно так:

```
{
_id: ...,
plot: 'A young man is accidentally sent 30 years into the past...',
genres: [ 'Adventure', 'Comedy', 'Sci-Fi' ],
...
title: 'Back to the Future',
...
}
```

Если нет отклика в виде выходных данных или появилось сообщение об ошибке, необходимо проверить, правильно ли была включена строка подключения в свой класс *Java*, был ли загружен образец набора данных в указанный кластер *MongoDB Atlas*.

После выполнения этого шага должно быть рабочее приложение, которое использует драйвер *Java*, для подключения к вашему кластеру *MongoDB*, для выполнения запроса к образцам данных и распечатки результата.

Можно научиться использовать свой собственный простой старый объект *Java* (*POJO*) для хранения и извлечения данных из *MongoDB*. Необходимо создать файл с именем *Movie.java* в каталоге базового пакета проекта и добавить следующий код для класса, который включает следующие поля, сеттеры и геттеры:

```
public class Movie {
    String plot;
    List<String> genres;
    String title;
    public String getPlot() {
        return plot;
    }
    public void setPlot(String plot) {
        this.plot = plot;
    }
    public List<String> getGenres() {
        return genres;
```

```
    }
    public void setGenres(List<String> genres) {
        this.genres = genres;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    @Override
    public String toString() {
        return "Movie [\n  plot=" + plot + ",\n  genres=" + genres +
",\n  title=" + title + "\n]";
    }
}
```

Необходимо создать новый файл *QuickStartPojoExample.java* в том же каталоге пакета, что и файл фильма в проекте. Используйте следующий пример кода, чтобы выполнить запрос к набору данных в *MongoDB Atlas*, заменив значение переменной *uri* строкой подключения *MongoDB Atlas*. Убедитесь, что вы заменили раздел *<password>* в строке подключения паролем, который вы создали для своего пользователя с разрешениями *atlasAdmin*:

```
import static com.mongodb.MongoClientSettings.getDefaultCodec
Registry;
import static com.mongodb.client.model.Filters.eq;
import static org.bson.codecs.configuration.CodecRegistries.from
Providers;
import static org.bson.codecs.configuration.CodecRegistries.from
Registries;
import org.bson.codecs.configuration.CodecProvider;
import org.bson.codecs.configuration.CodecRegistry;
import org.bson.codecs.pojo.PojoCodecProvider;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
```

```
public class QuickStartPojoExample {
   public static void main(String[] args) {
      CodecProvider pojoCodecProvider = PojoCodecProvider.-
builder().automatic(true).build();
      CodecRegistry pojoCodecRegistry = fromRegistries(getDefault-
CodecRegistry(), fromProviders(pojoCodecProvider));
```

// Замените строку *uri* на строку подключения вашего развертывания *MongoDB*:

```
      String uri = "<connection string uri>";
      try (MongoClient mongoClient = MongoClients.create(uri)) {
         MongoDatabase database = mongoClient.getDatabase
("sample_mflix").withCodecRegistry(pojoCodecRegistry);
         MongoCollection<Movie> collection = database.getCollection-
("movies", Movie.class);

         Movie movie = collection.find(eq("title", "Back to the Fu-
ture")).first();
         System.out.println(movie);
      }
   }
}
```

Когда вы запустите класс *QuickStartPojoExample*, он должен вывести детали фильма из образца набора данных, которые должны выглядеть примерно так:

```
Movie [
  plot=A young man is accidentally sent 30 years into the past...,
  genres=[Adventure, Comedy, Sci-Fi],
  title=Back to the Future
]
```

Можно получить отдельный документ в коллекции, объединив методы *find* () и *first* () в объекте *MongoCollection*. Возможно передать фильтр запроса методу *find*() для запроса и возврата документов, соответствующих фильтру в коллекции. Без подключения фильтра *MongoDB* вернет все документы в коллекции. Метод *first*() возвращает первый соответствующий документ или *null*, если результатов нет.

Можно связать другие методы с методом *find*(), например *sort*(), который организует сопоставленные документы в указанном порядке,

и *projection*(), который настраивает поля, включенные в возвращенные документы.

Метод *find*() возвращает экземпляр класса *FindIterable*, который предлагает несколько методов для доступа, организации и просмотра результатов. *FindIterable* также наследует методы своего родительского класса *MongoIterable*, такие как *first*().

Следующий фрагмент кода находит один документ из коллекции фильмов. Он использует следующие объекты и методы:

- фильтр запроса, который передается методу *find*(). Фильтр *eq* находит только фильмы, название которых точно соответствует тексту *The Room*;
- сортировка, которая упорядочивает совпадающие документы в порядке убывания рейтинга, поэтому если наш запрос соответствует нескольким документам, возвращаемый документ имеет наивысший рейтинг;
- проекция, которая включает объекты в полях *title* и *imdb* и исключает поле *_id* с помощью вспомогательного метода *excludeId*():

```
package usage.examples;
import static com.mongodb.client.model.Filters.eq;
import org.bson.Document;
import org.bson.conversions.Bson;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Projections;
import com.mongodb.client.model.Sorts;
public class FindOne {
   public static void main( String[] args ) {
// Замените строку uri на строку подключения вашего развертывания MongoDB:
        String uri = "<connection string uri>";
        try (MongoClient mongoClient = MongoClients.create(uri)) {
          MongoDatabase database = mongoClient.getDatabase("sample_mflix");
          MongoCollection<Document> collection = database.getCollection("movies");
          Bson projectionFields = Projections.fields(
```

```
                Projections.include("title", "imdb"),
                Projections.excludeId());
        Document doc = collection.find(eq("title", "The Room"))
                .projection(projectionFields)
                .sort(Sorts.descending("imdb.rating"))
                .first();
        if (doc == null) {
            System.out.println("No results found.");
        } else {
            System.out.println(doc.toJson());
        }
    }
  }
}
```

Для запроса нескольких документов в коллекции необходимо использовать метод *find*() для объекта *MongoCollection*. Передать фильтр запроса методу *find*() для запроса и возврата документов, соответствующих фильтру в коллекции.

Метод *find*() возвращает экземпляр класса *FindIterable*, который предлагает несколько методов для доступа, организации и просмотра результатов. *FindIterable* также наследует методы от своего родительского класса *MongoIterable*, который реализует основной интерфейс *Java Iterable*.

Можно использовать метод *iterator*() для *MongoIterable*, который возвращает экземпляр *MongoCursor*, который можно использовать для просмотра результатов. Возможно использовать методы *MongoCursor*, такие как *hasNext*(), чтобы проверить, существуют ли дополнительные результаты, или *next*(), чтобы вернуть следующий документ в коллекции. Если нет документов, соответствующих запросу, вызов *hasNext*() возвращает *false*, поэтому вызов *next*() и вызывает исключение.

При вызове *next*() для итератора либо после того, как он вернул окончательный результат, либо когда результатов не существует, он генерирует исключение типа *java.util.NoSuchElementException*. Всегда используйте *hasNext*() для проверки наличия дополнительных результатов перед вызовом *next*().

Следующий фрагмент кода находит и распечатывает все документы, соответствующие запросу в коллекции фильмов. Он использует следующие объекты и методы:

• фильтр запроса, который передается методу *find*(). Фильтр *lt*() находит только фильмы продолжительностью менее 15 минут;

• сортировка, которая упорядочивает возвращенные документы в порядке убывания по названию;

• проекция, которая включает объекты в полях *title* и *imdb* и исключает поле *_id* с помощью вспомогательного метода *excludeId* ():

```
package usage.examples;
import static com.mongodb.client.model.Filters.lt;
import org.bson.Document;
import org.bson.conversions.Bson;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoCursor;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Projections;
import com.mongodb.client.model.Sorts;
public class Find {
   public static void main( String[] args ) {
      // Замените строку uri на строку подключения вашего развертывания MongoDB:
      String uri = "<connection string uri>";
      try (MongoClient mongoClient = MongoClients.create(uri)) {
         MongoDatabase database = mongoClient.getDatabase("sample_mflix");
         MongoCollection<Document> collection = database.getCollection("movies");
         Bson projectionFields = Projections.fields(
            Projections.include("title", "imdb"),
            Projections.excludeId());
         MongoCursor<Document> cursor = collection.find(lt("runtime", 15))
            .projection(projectionFields)
            .sort(Sorts.descending("title")).iterator();
         try {
            while(cursor.hasNext()) {
               System.out.println(cursor.next().toJson());
            }
```

```
            } finally {
                cursor.close();
            }
        }
    }
}
```

Можно вставить отдельный документ в коллекцию, используя метод *insertOne*() для объекта *MongoCollection*. Чтобы вставить документ, необходимо создать объект *Document*, содержащий поля и значения. Если при вызове метода *insertOne*() коллекции еще не существует, сервер автоматически создаст ее.

После успешной вставки *insertOne*() возвращает экземпляр *InsertOneResult*. Для получения такой информации, как поле *_id* вставленного документа, используют *getInsertedId*() в экземпляре *InsertOneResult*. При неуспешной операции вставки драйвер вызывает исключение.

Следующий фрагмент вставляет один документ в коллекцию фильмов:

```
package usage.examples;
import java.util.Arrays;
import org.bson.Document;
import org.bson.types.ObjectId;
import com.mongodb.MongoException;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.result.InsertOneResult;
public class InsertOne {
   public static void main(String[] args) {
// Замените строку uri на строку подключения вашего развертывания MongoDB:
        String uri = "<connection string uri>";
        try (MongoClient mongoClient = MongoClients.create(uri)) {
            MongoDatabase database = mongoClient.getDatabase("sample_mflix");
            MongoCollection<Document> collection = database.getCollection("movies");
```

```
            try {
              InsertOneResult result = collection.insertOne(new Document()
                      .append("_id", new ObjectId())
                      .append("title", "Ski Bloopers")
                      .append("genres",        Arrays.asList("Documentary", "Comedy")));
               System.out.println("Success! Inserted document id: " + result.getInsertedId());
            } catch (MongoException me) {
              System.err.println("Unable to insert due to an error: " + me);
            }
          }
        }
    }
```

Можно вставить несколько документов в коллекцию за одну операцию, использовав метод *insertMany*() для объекта *MongoCollection*. Чтобы вставить их, добавьте объекты *Document* в список и передайте этот список в качестве аргумента функции *insertMany*(). Если вы вызываете метод *insertMany*() для коллекции, которая еще не существует, сервер создает ее для вас.

После успешной вставки *insertMany*() возвращает экземпляр *InsertManyResult*. Можно получить такую информацию, как поля *_id* вставленных вами документов, использовав метод *getInsertedIds*() в экземпляре *InsertManyResult*.

Следующий фрагмент кода вставляет несколько документов в коллекцию фильмов:

```
package usage.examples;
import java.util.Arrays;
import java.util.List;
import org.bson.Document;
import com.mongodb.MongoException;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.result.InsertManyResult;
public class InsertMany {
```

```
    public static void main(String[] args) {
      // Замените строку uri на строку подключения вашего развертывания MongoDB
      String uri = "<connection string uri>";
      try (MongoClient mongoClient = MongoClients.create(uri)) {
        MongoDatabase database = mongoClient.getDatabase("sample_mflix");
        MongoCollection<Document> collection = database.getCollection("movies");
        List<Document> movieList = Arrays.asList(
            new Document().append("title", "Short Circuit 3"),
            new Document().append("title", "The Lego Frozen Movie"));
        try {
          InsertManyResult result = collection.insertMany(movieList);
          System.out.println("Inserted document ids: " + result.getInsertedIds());
        } catch (MongoException me) {
          System.err.println("Unable to insert due to an error: " + me);
        }
      }
    }
```

Можно обновить отдельный документ, используя метод *updateOne*() для объекта *MongoCollection*. Метод принимает фильтр, соответствующий документу, который вы хотите обновить, и инструкцию обновления, которая инструктирует драйвер, как изменить соответствующий документ. Метод *updateOne*() обновляет только первый документ, соответствующий фильтру.

Чтобы выполнить обновление с помощью метода *updateOne*(), нужно передать фильтр запроса и документ обновления. Фильтр запроса определяет критерии, по которым следует выполнить обновление, а документ обновления содержит инструкции о том, какие изменения следует внести в него.

При желании можно передать экземпляр *UpdateOptions* методу *updateOne*(), чтобы указать поведение метода. Например, если установить для поля *upsert* объекта *UpdateOptions* значение *true*, операция вставит новый документ из полей как в запросе, так и в документе

обновления, если ни один из документов не соответствует фильтру запроса.

После успешного выполнения метод *updateOne*() возвращает экземпляр *UpdateResult*. Можно получить такую информацию, как количество измененных документов, использовав метод *getModified Count*(), или значение поля *_id*, вызвав метод *getUpsertedId*(), если вы указали *upsert*(*true*) в экземпляре *UpdateOptions*.

Если операция обновления завершилась неудачно, драйвер выдаст исключение. Например, если вы попытаетесь установить значение для неизменяемого поля *_id* в документе обновления, метод вызовет исключение *MongoWriteException* с сообщением.

Если ваш документ обновления содержит изменение, которое нарушает правила уникального индекса, метод генерирует исключение *MongoWriteException* с сообщением об ошибке.

В этом примере обновляется первое совпадение для нашего запроса в коллекции фильмов базы данных *sample_mflix*. Выполним следующие обновления соответствующего документа:

1) установите значение времени выполнения на 99;

2) добавьте спорт в массив жанров, только если его еще не существует;

3) установите значение *lastUpdated* на текущее время.

В примере используется конструктор *Updates*, фабричный класс, содержащий статические вспомогательные методы для создания документа обновления:

```
package usage.examples;
import org.bson.Document;
import org.bson.conversions.Bson;
import com.mongodb.MongoException;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.UpdateOptions;
import com.mongodb.client.model.Updates;
import com.mongodb.client.result.UpdateResult;
public class UpdateOne {
   public static void main(String[] args) {
      // Замените строку uri на строку подключения вашего развертывания MongoDB
```

```
        String uri = "<connection string uri>";
        try (MongoClient mongoClient = MongoClients.create(uri)) {
            MongoDatabase database = mongoClient.getDatabase("sample_mflix");
            MongoCollection<Document> collection = database.getCollection("movies");
            Document query = new Document().append("title", "Cool Runnings 2");
            Bson updates = Updates.combine(
                    Updates.set("runtime", 99),
                    Updates.addToSet("genres", "Sports"),
                    Updates.currentTimestamp("lastUpdated"));
            UpdateOptions options = new UpdateOptions().upsert(true);
            try {
                UpdateResult result = collection.updateOne(query, updates, options);
                System.out.println("Modified document count: " + result.getModifiedCount());
                System.out.println("Upserted id: " + result.getUpsertedId()); // only contains a value when an upsert is performed
            } catch (MongoException me) {
                System.err.println("Unable to update due to an error: " + me);
            }
        }
    }
```

После запуска примера должен можно увидеть результат, который выглядит примерно так:

Modified document count: *1*
Upserted id: null

Или так:

Modified document count: 0
Upserted id: BsonObjectId{value=...}

При запросе обновленного документа результат должен выглядеть примерно так:

Document {
{ *_id=...*,
plot=...,
genres=[*Adventure*, *Comedy*, *Family*, *Sports*],
runtime=99,
...
lastUpdated=Timestamp{...}
}
}

Можно обновить несколько документов, используя метод *updateMany*() для объекта *MongoCollection*. Метод принимает фильтр, соответствующий документу, который вы хотите обновить, и инструкцию обновления, которая инструктирует драйвер, как изменить соответствующий документ. Метод *updateMany*() обновляет все документы в коллекции, соответствующие фильтру.

Чтобы выполнить обновление с помощью метода *updateMany*(), нужно передать фильтр запроса и документ обновления. Фильтр запроса указывает на то, какие документы в коллекции должны соответствовать, а документ обновления предоставляет инструкции о том, какие изменения нужно внести в них.

При желании можно передать экземпляр *UpdateOptions* методу *updateMany*(), чтобы изменить поведение вызова. Например, если при установке для поля *upsert* объекта *UpdateOptions* значения *true* ни один документ не соответствует указанному фильтру запроса, операция вставит новый документ, состоящий из полей, как из запроса, так и из документа обновления.

После успешного выполнения метод *updateMany*() возвращает экземпляр *UpdateResult*. Вы можете получить такую информацию, как количество измененных документов, вызвав метод *getModifiedCount* (). Если вы указали *upsert* (*true*) в объекте *UpdateOptions* и операция приводит к вставке, вы можете получить поле *_id* нового документа, вызвав метод *getUpsertedId* () в экземпляре *UpdateResult*.

При неуспешной операции обновления драйвер вызывает исключение и не обновляет ни один из документов, соответствующих фильтру. Например, если драйвер попытается установить значение для неизменяемого поля *_id* в документе обновления, метод *updateMany*() не обновит никакие документы и выдаст исключение *MongoWriteException*.

Если ваш документ обновления содержит изменение, которое нарушает правила уникального индекса, метод генерирует исключение *MongoWriteException* с сообщением об ошибке.

В этом примере обновляются документы, соответствующие нашему запросу в коллекции фильмов базы данных *sample_mflix*. Выполним следующие обновления соответствующих документов:

1) добавьте «часто обсуждаемые» в массив жанров, только если он еще не существует;

2) установите значение *lastUpdated* на текущее время.

Используем конструктор *Updates*, фабричный класс, который содержит статические вспомогательные методы для создания документа обновления:

```
package usage.examples;
import static com.mongodb.client.model.Filters.gt;
import org.bson.Document;
import org.bson.conversions.Bson;
import com.mongodb.MongoException;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Updates;
import com.mongodb.client.result.UpdateResult;
public class UpdateMany {
   public static void main(String[] args) {
```

// Замените строку *uri* на строку подключения вашего развертывания *MongoDB*:

```
        String uri = "<connection string uri>";
        try (MongoClient mongoClient = MongoClients.create(uri)) {
          MongoDatabase database = mongoClient.getDatabase
("sample_mflix");
          MongoCollection<Document> collection = database.get
Collection("movies");
          Bson query = gt("num_mflix_comments", 50);
          Bson updates = Updates.combine(
                Updates.addToSet("genres", "Frequently Discussed"),
                Updates.currentTimestamp("lastUpdated"));
          try {
           UpdateResult result = collection.updateMany(query, updates);
```

```
                System.out.println("Modified document count: " + result.getModifiedCount());
            } catch (MongoException me) {
                System.err.println("Unable to update due to an error: " + me);
            }
        }
    }
}
```

После запуска примера вы должны увидеть результат, который выглядит примерно так:

Modified document count: 53

Если вы запросите обновленный документ или документы, они должны выглядеть примерно так:

```
[
Document {
{ _id=...,
plot=...,
genres=[..., Frequently Discussed, ...],
...
lastUpdated=Timestamp{...}
}
},
...
]
```

Можно заменить один документ с помощью метода *replaceOne*() объекта *MongoCollection*. Этот метод удаляет все существующие поля и значения из документа (кроме поля *_id*) и заменяет его вашим документом.

Метод *replaceOne*() принимает фильтр запроса, соответствующий документу, который вы хотите заменить, и заменяющий документ, содержащий данные, которые вы хотите сохранить, вместо совпадающего документа. Метод *replaceOne*() заменяет только первый документ, соответствующий фильтру.

При желании можно передать экземпляр *ReplaceOptions* методу *replaceOne*(), чтобы указать поведение метода. Например, при уста-

новке для поля *upsert* объекта *ReplaceOptions* значения *true* операция вставит новый документ из полей в заменяющем документе, при условии, что ни один из документов не соответствует фильтру запроса.

После успешного выполнения метод *replaceOne*() возвращает экземпляр *UpdateResult*. Можно получить информацию о количестве измененных документов, использовав метод *getModifiedCount*(). Можно также получить значение поля *_id* документа, использовав метод *getUpsertedId*(), если было установлено *upsert*(*true*) в экземпляре *ReplaceOptions* и операция привела к вставке нового документа.

Если операция замены завершилась неудачно, драйвер вызывает исключение. Например, если вы попытаетесь указать значение неизменяемого поля *_id* в замещающем документе, которое отличается от исходного документа, метод генерирует исключение *MongoWriteException*.

Если замещающий документ содержит изменение, которое нарушает правила уникального индекса, метод генерирует исключение *MongoWriteException*.

В этом примере мы заменяем первое совпадение нашего фильтра запроса в коллекции фильмов базы данных *sample_mflix* на заменяющий документ. Все поля, кроме поля *_id*, удаляются из исходного документа и заменяются замещающим документом.

Перед запуском операции *replaceOne*() исходный документ содержит несколько полей, описывающих фильм. После операции итоговый документ будет содержать только поля, указанные в замещающем документе (заголовок и полный график), и поле *_id*.

В приведенном ниже фрагменте кода используются следующие объекты и методы:

- фильтр запроса, который передается методу *replaceOne*(). Фильтр находит только фильмы, название которых точно соответствует тексту *Music of the Heart*;
- документ замены, содержащий документ, заменяющий соответствующий документ, если он существует;
- объект *ReplaceOptions* с параметром *upsert*, установленным в значение *true*. Этот параметр указывает, что метод должен вставлять данные, содержащиеся в заменяющем документе, если фильтр запроса не соответствует ни одному документу:

```
package usage.examples;
import static com.mongodb.client.model.Filters.eq;
```

```
import org.bson.Document;
import org.bson.conversions.Bson;
import com.mongodb.MongoException;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.ReplaceOptions;
import com.mongodb.client.result.UpdateResult;
public class ReplaceOne {
   public static void main(String[] args) {
```

// Замените строку *uri* на строку подключения вашего развертывания *MongoDB*:

```
        String uri = "<connection string uri>";
        try (MongoClient mongoClient = MongoClients.create(uri)) {
          MongoDatabase database = mongoClient.getDatabase
("sample_mflix");
          MongoCollection<Document> collection = database.get
Collection("movies");
          Bson query = eq("title", "Music of the Heart");
          Document replaceDocument = new Document().
               append("title", "50 Violins").
               append("fullplot", " A dramatization of the true story of
Roberta Guaspari who co-founded the Opus 118 Harlem School of Music");
          ReplaceOptions opts = new ReplaceOptions().upsert(true);
          UpdateResult result = collection.replaceOne(query, replace-
Document, opts);
          System.out.println("Modified document count: " + re-
sult.getModifiedCount());
          System.out.println("Upserted id: " + result.getUpsertedId());
         // only contains a value when an upsert is performed
        }
     catch (MongoException me) {
          System.err.println("Unable to replace due to an error: " + me);
        }
    }
}
```

После запуска примера вы должны увидеть результат, который выглядит примерно так:

Modified document count: 1
Upserted id: null

Или, если пример привел к опровержению:

Modified document count: 0
Upserted id: BsonObjectId{value=...}

Если вы запросите замененный документ, он должен выглядеть примерно так:

Document {
{ _id=...,
title=50 Violins,
fullplot=A dramatization of the true story of Roberta Guaspari who co-founded the Opus 118 Harlem School of Music
}
}

Можно удалить отдельный документ из коллекции, используя метод *deleteOne*() для объекта *MongoCollection*. Метод принимает фильтр запроса, соответствующий документу, который вы хотите удалить. Если вы не укажете фильтр, *MongoDB* будет соответствовать первому документу в коллекции. Метод *deleteOne*() удаляет только первый совпавший документ.

Этот метод возвращает экземпляр *DeleteResult*, который содержит информацию, в том числе и о том, сколько документов было удалено в результате операции. Если операция удаления завершилась неудачно, драйвер выдает исключение.

Следующий фрагмент кода удаляет один документ из коллекции фильмов в базе данных *sample_mflix*. В примере используется фильтр *eq*() для сопоставления фильмов с заголовком, точно соответствующим тексту *The Garbage Pail Kids Movie*:

package usage.examples;
import static com.mongodb.client.model.Filters.eq;
import org.bson.Document;
import org.bson.conversions.Bson;
import com.mongodb.MongoException;
import com.mongodb.client.MongoClient;

```
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.result.DeleteResult;
public class DeleteOne {
   public static void main(String[] args) {
```

// Замените строку *uri* на строку подключения вашего развертывания *MongoDB*:

```
        String uri = "<connection string uri>";
        try (MongoClient mongoClient = MongoClients.create(uri)) {
          MongoDatabase database = mongoClient.getDatabase
("sample_mflix");
          MongoCollection<Document> collection = database.get
Collection("movies");
          Bson query = eq("title", "The Garbage Pail Kids Movie");
          try {
            DeleteResult result = collection.deleteOne(query);
            System.out.println("Deleted document count: " + result.
getDeletedCount());
          } catch (MongoException me) {
            System.err.println("Unable to delete due to an error: " + me);
          }
        }
    }
}
```

Если при запуске примера фильтр запроса, который передали в вызове *deleteOne*(), соответствует документу и удаляет его, можно увидеть результат, который выглядит примерно так: *Deleted document count*: 1.

Если фильтр запроса не соответствует документу в коллекции, вызов *deleteOne*() не удаляет никаких документов и возвращает следующее: *Deleted document count*: 0.

Можно удалить несколько документов из коллекции за одну операцию, использовав метод *deleteMany*() для объекта *MongoCollection*.

Чтобы указать, какие документы нужно удалить, передайте фильтр запроса, соответствующий документам, которые хотите удалить. Если предоставлен пустой документ, *MongoDB* сопоставляет все

документы в коллекции и удаляет их. Вы можете использовать *deleteMany*() для удаления всех документов в коллекции, однако рассмотрите возможность использования вместо этого метода *drop*() для повышения производительности.

После успешного удаления этот метод возвращает экземпляр *DeleteResult*. Можно получить такую информацию, как количество удаленных документов, вызвав метод *getDeletedCount*() для экземпляра *DeleteResult*. Если операция удаления завершилась неудачно, драйвер выдает исключение.

Следующий фрагмент удаляет несколько документов из коллекции фильмов в базе данных *sample_mflix*. Фильтр запроса, переданный методу *deleteMany*(), соответствует всем документам фильмов, которые содержат рейтинг ниже 1,9 во вложенном документе *imdb*:

```
package usage.examples;
import static com.mongodb.client.model.Filters.lt;
import org.bson.Document;
import org.bson.conversions.Bson;
import com.mongodb.MongoException;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.result.DeleteResult;
public class DeleteMany {
   public static void main(String[] args) {
// Замените строку uri на строку подключения вашего развертывания MongoDB:
        String uri = "<connection string uri>";
        try (MongoClient mongoClient = MongoClients.create(uri)) {
          MongoDatabase database = mongoClient.getDatabase("sample_mflix");
          MongoCollection<Document> collection = database.getCollection("movies");
          Bson query = lt("imdb.rating", 1.9);
          try {
            DeleteResult result = collection.deleteMany(query);
            System.out.println("Deleted document count: " + result.getDeletedCount());
```

```
            } catch (MongoException me) {
              System.err.println("Unable to delete due to an error: " + me);
            }
          }
        }
      }
```

Когда запустите пример, увидите результат, в котором указано количество документов, удаленных при обращении к *deleteMany*(): *Deleted document count*: 4.

Метод *bulkWrite*() выполняет операции пакетной записи для одной коллекции. Этот метод сокращает количество сетевых циклов от приложения к экземпляру *MongoDB*, что увеличивает производительность приложения.

Можно указать одну или несколько из следующих операций записи в *bulkWrite*():

1) *insertOne;*
2) *updateOne;*
3) *updateMany;*
4) *deleteOne;*
5) *deleteMany;*
6) *replaceOne.*

Метод *bulkWrite* () принимает следующие параметры:

- список объектов, реализующих *WriteModel*: классы, реализующие *WriteModel*, соответствуют вышеупомянутым операциям записи;
- *BulkWriteOptions*: необязательный объект, который указывает параметры, например, следует ли гарантировать, что экземпляр *MongoDB* упорядочивает ваши операции записи.

Если одна или несколько ваших операций попытаются установить значение, которое нарушает уникальный индекс вашей коллекции, возникнет исключение.

Точно так же, если попытаться выполнить массовую запись в коллекцию, которая использует проверку схемы, и одна или несколько ваших операций записи предоставят неожиданный формат, вы столкнетесь с исключениями.

В следующем примере кода выполняется упорядоченная операция массовой записи для коллекции фильмов в базе данных *sample_mflix*. Пример вызова *bulkWrite*() включает примеры *InsertOne Model, UpdateOneModel* и *DeleteOneModel*:

```
import java.util.Arrays;
import org.bson.Document;
import com.mongodb.MongoException;
import com.mongodb.bulk.BulkWriteResult;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.DeleteOneModel;
import com.mongodb.client.model.InsertOneModel;
import com.mongodb.client.model.ReplaceOneModel;
import com.mongodb.client.model.UpdateOneModel;
import com.mongodb.client.model.UpdateOptions;
public class BulkWrite {
   public static void main(String[] args) {
```

// Замените строку *uri* на строку подключения вашего развертывания *MongoDB*:

```
        String uri = "<connection string uri>";
        try (MongoClient mongoClient = MongoClients.create(uri)) {
          MongoDatabase database = mongoClient.getDatabase("sample_mflix");
          MongoCollection<Document> collection = database.getCollection("movies");
          try {
            BulkWriteResult result = collection.bulkWrite(
                Arrays.asList(
                  new InsertOneModel<>(new Document("name", "A Sample Movie")),
                  new InsertOneModel<>(new Document("name", "Another Sample Movie")),
                  new InsertOneModel<>(new Document("name", "Yet Another Sample Movie")),
                  new UpdateOneModel<>(new Document("name", "A Sample Movie"),
                  new Document("$set", new Document("name", "An Old Sample Movie")),
                  new UpdateOptions().upsert(true)),
                  new DeleteOneModel<>(new Document("name", "Yet Another Sample Movie")),
```

```
                    new ReplaceOneModel<>(new Document("name",
"Yet Another Sample Movie"),
                    new Document("name", "The Other Sample
Movie").append("runtime", "42"))
                            ));
                System.out.println("Result statistics:" +
                        "\ninserted: " + result.getInsertedCount() +
                        "\nupdated: " + result.getModifiedCount() +
                        "\ndeleted: " + result.getDeletedCount());
            } catch (MongoException me) {
                System.err.println("The bulk write operation failed due to
an error: " + me);
            }
        }
    }
}
```

Результат должен выглядеть примерно так:

```
Result statistics:
inserted: 3
updated: 2
deleted: 1
```

Можно отслеживать изменения данных в *MongoDB*, такие как изменения в коллекции, базе данных или развертывании, открыв поток изменений. Поток изменений позволяет приложениям отслеживать изменения данных и реагировать на них. Поток изменений возвращает документы событий изменений, когда происходят изменения. Вы можете открыть поток изменений, вызвав метод *watch*() для объекта *MongoCollection*, *MongoDatabase* или *MongoClient*:

```
ChangeStreamIterable < Document > changeStream = data-
base.watch ();
```

Метод *watch*() дополнительно принимает конвейер агрегации, который состоит из массива этапов, в качестве первого параметра для фильтрации и преобразования вывода события изменения следующим образом:

List <Bson> pipeline = Arrays.asList (
Aggregates.match (
Filters.lt ("fullDocument.runtime", 15)));
ChangeStreamIterable <Document> changeStream = database.watch (конвейер);

Метод *watch*() возвращает экземпляр класса *ChangeStreamIterable*, который предлагает несколько методов для доступа, организации и просмотра результатов. *ChangeStreamIterable* также наследует методы от своего родительского класса *MongoIterable*, который реализует основной интерфейс *Java Iterable*.

Можно использовать метод *forEach*() в *ChangeStreamIterable* для обработки событий по мере их возникновения или использовать метод *iterator*(), который возвращает экземпляр *MongoCursor*, который можно использовать для просмотра результатов.

Можно вызвать методы *MongoCursor*, такие как *hasNext*(), чтобы проверить, существуют ли дополнительные результаты, *next*(), чтобы вернуть следующий документ в коллекции, или *tryNext*(), чтобы немедленно вернуть либо следующий доступный элемент в потоке изменений, либо *null*. В отличие от *MongoCursor*, возвращаемого другими запросами, *MongoCursor*, связанный с потоком изменений, ожидает прибытия события изменения перед возвратом результата из *next*(). В результате вызовы *next*() с использованием *MongoCursor* потока изменений никогда не вызывают исключение *java.util.NoSuch ElementException*.

Чтобы настроить параметры обработки документов, возвращаемых из потока изменений, используйте методы-члены объекта *ChangeStreamIterable*, возвращаемого функцией *watch*().

Чтобы захватить события из потока изменений, вызовите метод *forEach*() с функцией обратного вызова, как показано ниже:

changeStream.forEach (событие -> System.out.println ("Наблюдаемое изменение:" + событие));

Функция обратного вызова срабатывает при возникновении события изменения. Можно указать логику в обратном вызове для обработки документа события при его получении.

В следующем примере используются два отдельных приложения, чтобы продемонстрировать, как отслеживать изменения с помощью потока изменений:

1) первое приложение под названием *Watch* открывает поток изменений в коллекции фильмов в базе данных *sample_mflix*. *Watch* использует конвейер агрегации для фильтрации изменений на основе *operationType*, чтобы получать только события вставки и обновления (удаления исключаются по пропуску). *Watch* использует обратный вызов для получения и печати отфильтрованных событий изменения, происходящих в коллекции:

```
package usage.examples;
import org.bson.Document;
import org.bson.conversions.Bson;
import com.mongodb.client.ChangeStreamIterable;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
import com.mongodb.client.model.changestream.FullDocument;
public class Watch {
   public static void main( String[] args ) {
```

// Замените строку *uri* на строку подключения вашего развертывания *MongoDB*:

```
        String uri = "<connection string uri>";
        try (MongoClient mongoClient = MongoClients.create(uri)) {
          MongoDatabase database = mongoClient.getDatabase("sample_mflix");
          MongoCollection<Document> collection = database.getCollection("movies");
          List<Bson> pipeline = Arrays.asList(
            Aggregates.match(
                Filters.in("operationType",
                    Arrays.asList("insert", "update"))));
          ChangeStreamIterable<Document> changeStream = database.watch(pipeline)
              .fullDocument(FullDocument.UPDATE_LOOKUP);
          // variables referenced in a lambda must be final; final array gives us a mutable integer
          final int[] numberOfEvents = {0};
          changeStream.forEach(event -> {
```

```
            System.out.println("Received a change to the collection: " +
event);
                if (++numberOfEvents[0] >= 2) {
                  System.exit(0);
                }
              });
            }
          }
        }
```

2) второе приложение под названием *WatchCompanion* вставляет один документ в коллекцию фильмов в базе данных *sample_mflix*. Затем *WatchCompanion* обновляет документ новым значением поля. Наконец, *WatchCompanion* удаляет документ:

```
package usage.examples;
import java.util.Arrays;
import org.bson.Document;
import org.bson.types.ObjectId;
import com.mongodb.MongoException;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.result.InsertOneResult;
public class WatchCompanion {
   public static void main(String[] args) {
```

// Замените строку *uri* на строку подключения вашего развертывания *MongoDB*:

```
        String uri = "<connection string uri>";
        try (MongoClient mongoClient = MongoClients.create(uri)) {
          MongoDatabase database = mongoCli-
ent.getDatabase("sample_mflix");
          MongoCollection<Document> collection = data-
base.getCollection("movies");
          try {
            InsertOneResult insertResult = collection.insertOne(new
Document("test", "sample movie document"));
            System.out.println("Success! Inserted document id: " + in-
sertResult.getInsertedId());
```

```
                UpdateResult updateResult = collection.updateOne(new
Document("test", "sample movie document"), Updates.set("field2", "sam-
ple movie document update"));
                System.out.println("Updated " + updateResult.getMod-
ifiedCount() + " document.");
                DeleteResult deleteResult = collection.deleteOne(new
Document("field2", "sample movie document update"));
                System.out.println("Deleted " + deleteResult.getDeleted
Count()+ " document.");
            } catch (MongoException me) {
                System.err.println("Unable to insert, update, or replace
due to an error: " + me);
            }
        }
    }
}
```

Сначала запустите *Watch*, чтобы открыть поток изменений в коллекции и определить обратный вызов в потоке изменений с помощью метода *forEach*(). Во время работы *Watch* запустите *WatchCompanion*, чтобы сгенерировать события изменения путем внесения изменений в коллекцию.

Если вы последовательно запустите предыдущие приложения, то увидите выходные данные приложения *Watch*, аналогичные приведенным ниже. Распечатываются только операции вставки и обновления, поскольку конвейер агрегации отфильтровывает операцию удаления:

```
Received a change to the collection: ChangeStreamDocument{
  operationType=OperationType{value='insert'},
  resumeToken={"_data": "825EC..."},
  namespace=sample_mflix.movies,
  destinationNamespace=null,
  fullDocument=Document{{_id=5ec3..., test=sample movie docu-
ment}},
  documentKey={"_id": {"$oid": "5ec3..."}},
  clusterTime=Timestamp{...},
  updateDescription=null,
  txnNumber=null,
  lsid=null
```

```
}
Received a change to the collection: ChangeStreamDocument{
  operationType=OperationType{value='update'},
  resumeToken={"_data": "825EC..."},
  namespace=sample_mflix.movies,
  destinationNamespace=null,
  fullDocument=Document{{_id=5ec3..., test=sample movie document, field2=sample movie document update}},
  documentKey={"_id": {"$oid": "5ec3..."}},
  clusterTime=Timestamp{...},
  updateDescription=UpdateDescription{removedFields=[], updatedFields={"field2": "sample movie document update"}},
  txnNumber=null,
  lsid=null
}
```

Вы также должны увидеть вывод приложения *WatchCompanion*, подобный следующему:

```
Success! Inserted document id: BsonObjectId{value=5ec3...}
Updated 1 document.
Deleted 1 document.
```

В классе *MongoCollection* есть два метода экземпляра, которые можно вызвать для подсчета количества документов в коллекции:

1) *countDocuments*() возвращает количество документов в коллекции, соответствующих указанному запросу. Если укажете пустой фильтр запроса, метод вернет общее количество документов в коллекции;

2) *EstimatedDocumentCount*() возвращает оценку количества документов в коллекции на основе метаданных коллекции. При использовании этого метода нельзя применять запрос.

Метод *EstimatedDocumentCount*() возвращается быстрее, чем метод *countDocuments*(), потому что он использует метаданные коллекции, а не сканирует всю коллекцию. Метод *countDocuments*() возвращает точное количество документов и поддерживает указание фильтра.

В следующем примере оценивается количество документов в коллекции фильмов в базе данных *sample_mflix*, а затем возвращается точное количество документов в коллекции фильмов *Spain* в поле *countries*:

```
package usage.examples;
import static com.mongodb.client.model.Filters.eq;
import org.bson.Document;
import org.bson.conversions.Bson;
import com.mongodb.MongoException;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
public class CountDocuments {
   public static void main(String[] args) {
```

// Замените строку *uri* на строку подключения вашего развертывания *MongoDB*:

```
        String uri = "<connection string uri>";
        try (MongoClient mongoClient = MongoClients.create(uri)) {
          MongoDatabase database = mongoClient.getDatabase
("sample_mflix");
          MongoCollection<Document> collection = database.get-
Collection("movies");
          Bson query = eq("countries", "Spain");
          try {
            long estimatedCount = collection.estimatedDocument
Count();
            System.out.println("Estimated number of documents in the
movies collection: " + estimatedCount);
            long matchingCount = collection.countDocuments(query);
            System.out.println("Number of movies from Spain: " +
matchingCount);
          } catch (MongoException me) {
            System.err.println("An error occurred: " + me);
          }
        }
    }
}
```

Если вы запустите предыдущий пример кода, увидите результат, который выглядит примерно так (точные числа могут отличаться в зависимости от ваших данных):

Estimated number of documents in the movies collection: 23541
Number of movies from Spain: 755

Можно получить список различных значений для поля в коллекции, использовав метод *unique*() для объекта *MongoCollection*. Передайте имя поля документа в качестве первого параметра и класс, к которому вы хотите привести результаты, в качестве второго параметра, как показано ниже: *collection.distinct* (страны, *String.class*).

Можно указать поле в документе или поле во встроенном документе, используя точечную нотацию. Следующий вызов метода возвращает каждое отдельное значение поля побед во встроенном документе *awards*: *collection.distinct* (*awards.wins*, *Integer.class*).

При желании можно передать в метод фильтр запроса, чтобы ограничить набор документов, из которых ваш экземпляр *MongoDB* извлекает отдельные значения следующим образом: *collection.distinct* («тип», *Filters*.eq («языки», «французский»), *String.class*).

Метод *Different*() возвращает объект, реализующий интерфейс *DistinctIterable*. Этот интерфейс содержит методы для доступа, организации и просмотра результатов. Он также наследует от своего родительского интерфейса *MongoIterable* такие методы, как *first*(), который возвращает первый результат, и *cursor*(), который возвращает экземпляр *MongoCursor*.

Следующий фрагмент кода извлекает список различных значений для поля документа года из коллекции фильмов. Он использует фильтр запроса для сопоставления фильмов, которые включают *Carl Franklin* в качестве одного из значений в массиве *directors*:

```
package usage.examples;
import org.bson.Document;
import com.mongodb.MongoException;
import com.mongodb.client.DistinctIterable;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoCursor;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
public class Distinct {
   public static void main(String[] args) {
```

// Замените строку *uri* на строку подключения вашего развертывания *MongoDB*:

```
String uri = "<connection string uri>";
try (MongoClient mongoClient = MongoClients.create(uri)) {
  MongoDatabase database = mongoClient.getDatabase-
("sample_mflix");
  MongoCollection<Document> collection = database.get
Collection("movies");
  try {
    DistinctIterable<Integer> docs = collection.distinct-
("year", Filters.eq("directors", "Carl Franklin"), Integer.class);
    MongoCursor<Integer> results = docs.iterator();
    while(results.hasNext()) {
      System.out.println(results.next());
    }
  } catch (MongoException me) {
    System.err.println("An error occurred: " + me);
  }
 }
}
```

Когда вы запустите пример, увидите выходные данные, в которых сообщается каждый отдельный год для всех фильмов, когда Карл Франклин был включен в качестве режиссера; это должно выглядеть примерно так:

```
1992
1995
1998
...
```

Можно запускать все необработанные операции с базой данных с помощью метода *MongoDatabase.runCommand*(). Необработанная операция с базой данных – это команда, которую вы можете выполнить непосредственно в интерфейсе командной строки сервера *MongoDB*. Эти команды включают административные и диагностические задачи, такие как получение статистики сервера или инициализация набора реплик. Вызовите метод *runCommand*() с объектом ко-

манды *Bson* в экземпляре *MongoDatabase*, чтобы запустить операцию с исходной базой данных.

Метод *runCommand*() принимает команду в виде объекта *Bson*. По умолчанию *runCommand* возвращает объект типа *org.bson. Document*, содержащий выходные данные команды базы данных. Вы можете указать тип возвращаемого значения для *runCommand*() как необязательный второй параметр.

В следующем примере кода мы отправляем команду *dbStats* для запроса статистики из конкретной базы данных *MongoDB*:

```
package usage.examples;
import org.bson.BsonDocument;
import org.bson.BsonInt64;
import org.bson.Document;
import com.mongodb.ConnectionString;
import com.mongodb.MongoClientSettings;
import com.mongodb.MongoException;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoDatabase;
public class RunCommand {
   public static void main(String[] args) {
```

// Замените строку *uri* на строку подключения вашего развертывания *MongoDB*:

```
        String uri = "<connection string uri>";
        try (MongoClient mongoClient = MongoClients.create(uri)) {
          MongoDatabase database = mongoClient.getDatabase
("sample_mflix");
            try {
              Bson command = new BsonDocument("dbStats", new Bson
Int64(1));
               Document commandResult = database.runCommand
(command);
               System.out.println("dbStats: " + commandResult.toJson());
            } catch (MongoException me) {
               System.err.println("An error occurred: " + me);
            }
        }
    }
```

}

Когда вы запустите предыдущую команду, увидите результат, подобный следующему:

dbStats: {"db": "sample_mflix", "collections": 5, "views": 0, "objects": 75595, "avgObjSize": 692.1003770090614, "dataSize": 52319328, "storageSize": 29831168, "numExtents": 0, "indexes": 9, "indexSize": 14430208, "fileSize": 0, "nsSizeMB": 0, "ok": 1}

Глава 7. РАБОТА С ДАННЫМИ *NoSQL* в *Azure Cosmos DB*

Преимущество *Azure Cosmos DB*

В настоящее время приложения должны быть очень быстрыми и иметь подключение к Интернету. Для обеспечения высокой скорости и доступности копии этих приложений должны быть развернуты в центрах обработки данных, которые находятся рядом с их пользователями. Приложениям необходимо в режиме реального времени реагировать на реальные изменения в использовании в часы пик, сохранять огромные объемы данных и делать эти данные доступными для пользователей за миллисекунды.

Azure Cosmos DB – это полностью управляемая база данных *NoSQL* для разработки современных приложений. Время отклика в миллисекундах, а также автоматическая и мгновенная масштабируемость гарантируют скорость в любом масштабе. Непрерывность бизнеса обеспечивается доступностью на основе *SLA* и безопасностью корпоративного уровня. Разработка приложений происходит быстрее и продуктивнее благодаря распределению данных по нескольким регионам под ключ в любой точке мира, *API* с открытым исходным кодом и *SDK* для популярных языков. Как полностью управляемая служба, *Azure Cosmos DB* поддерживает администрирование базы данных за счет автоматического управления, обновлений и исправлений. *Azure Cosmos DB* обеспечивает управление емкостью с помощью экономичных бессерверных опций и автоматического масштабирования, которые отвечают требованиям приложений.

Возможна быстрая и упрощенная разработка приложений с помощью *API*-интерфейсов с открытым исходным кодом, нескольких *SDK*, данных без схемы и анализа операционных данных без использования *ETL*:

- глубокая интеграция с ключевыми службами *Azure*, используемыми при разработке современных (облачных) приложений, включая функции *Azure, AKS* (служба *Azure Kubernetes*), службу приложений и многое другое;
- выбор из нескольких *API* баз данных, включая собственный *Core* (*SQL*) *API*, *API* для *MongoDB*, *Cassandra API*, *Gremlin API* и *Table API*;

• создание приложения на основе *Core* (*SQL*) *API*, используя выбранные языки с помощью пакетов *SDK* для *.NET*, *Java*, *Node.js* и *Python*. А также можно выбрать драйверы для любого другого *API* базы данных;

• выполнение аналитики без *ETL* для рабочих данных, хранящихся в *Azure Cosmos DB* в режиме реального времени, с помощью *Azure Synapse Analytics*;

• канал изменений позволяет легко отслеживать изменения в контейнерах базы данных и управлять ими, а также создавать инициируемые события с помощью функций *Azure*;

• служба *Azure Cosmos DB* без схемы автоматически индексирует все ваши данные независимо от модели данных для доставки невероятно быстрых запросов.

Сквозное управление базой данных с бессерверным и автоматическим масштабированием в соответствии с потребностями вашего приложения:

• полностью управляемая служба базы данных, что помогает сэкономить время и деньги разработчиков;

• экономичные варианты для непредсказуемых или спорадических рабочих нагрузок любого размера и масштаба, позволяющие разработчикам легко приступить к работе без необходимости планирования мощности;

• бессерверная модель предлагает резкие рабочие нагрузки, автоматический и оперативный сервис для управления всплесками трафика по запросу;

• масштабирование выделенной пропускной способности автоматически и мгновенно масштабирует емкость для непредсказуемых рабочих нагрузок, сохраняя при этом *SLA*.

Веб-приложение, мобильное приложение или игровое приложение, которым необходимо обрабатывать огромные объемы данных, считывать и записывать в глобальном масштабе с почти реальным временем отклика для различных данных, выигрывают от гарантированной высокой доступности, высокой пропускной способности *Cosmos DB*, низкой задержки и настраиваемой согласованности. *Azure Cosmos DB* можно использовать для розничной торговли и маркетинга, игр, веб-приложений и мобильных приложений.

Сценарии использования Интернета вещей обычно имеют общие закономерности в том, как они получают, обрабатывают и хранят данные. Этим системам необходимо получать пакеты данных от дат-

чиков устройств в различных регионах. Затем эти системы обрабатывают и анализируют потоковые данные для получения информации в реальном времени. Затем данные архивируются в «холодное» хранилище для пакетной аналитики. *Microsoft Azure* предлагает разнообразные службы, которые можно применять для сценариев использования Интернета вещей, включая *Azure Cosmos DB*, концентраторы событий *Azure*, *Azure Stream Analytics*, Центр уведомлений *Azure*, Машинное обучение *Azure*, *Azure HDInsight* и *Power BI* (рис. 7.1) [10].

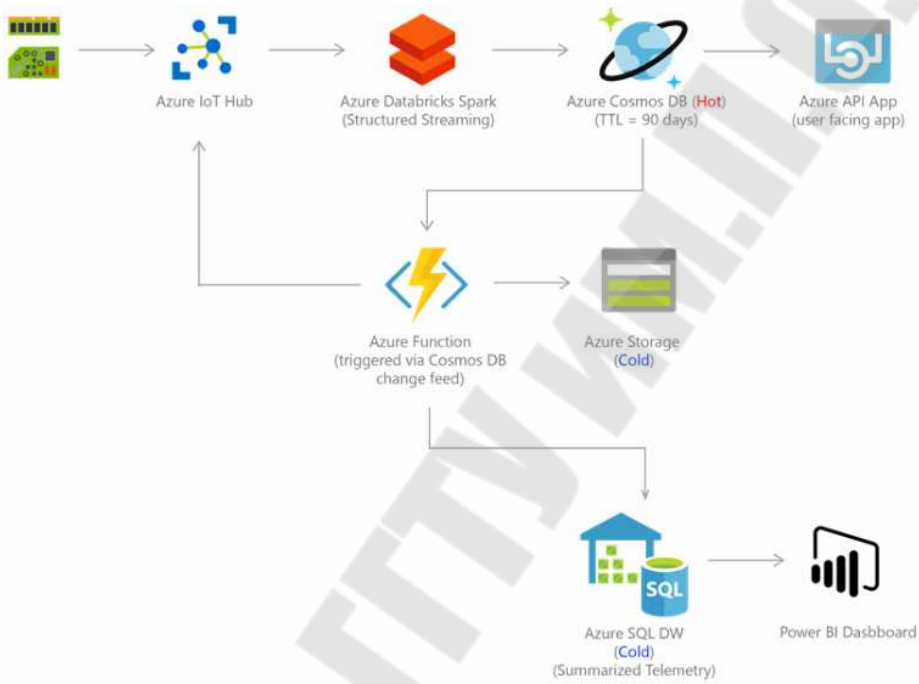

Рис. 7.1. Службы *Microsoft Azure*

Пакеты данных могут приниматься концентраторами событий *Azure*, поскольку они обеспечивают прием данных с высокой пропускной способностью и малой задержкой. Полученные данные, которые необходимо обработать для анализа в реальном времени, можно направить в *Azure Stream Analytics*. Данные могут быть загружены в *Azure Cosmos DB* для специальных запросов. Загруженные в *Azure Cosmos DB* данные готовы к запросу. Кроме того, в ленте изменений можно прочитать новые данные и изменения уже существующих данных. Лента изменений – это постоянный журнал, в котором в последовательном порядке сохраняются изменения в контейнерах *Cosmos*. Затем все данные или только изменения в данных в *Azure Cosmos DB* можно использовать в качестве справочных в рамках аналитики в реальном времени. Кроме того, данные можно уточнять и обрабатывать,

подключив данные *Azure Cosmos DB* к *HDInsight* для заданий *Pig*, *Hive* или *MapReduce*. Затем уточненные данные загружаются обратно в *Azure Cosmos DB* для создания отчетов.

Azure Cosmos DB широко используется на собственных платформах электронной коммерции *Microsoft*, на которых работают *Windows Store* и *XBox Live*. Используется также в розничной торговле для хранения данных каталога и для поиска событий в конвейерах обработки заказов.

Сценарии использования данных каталога включают хранение и запрос набора атрибутов для таких сущностей, как люди, места и продукты. Некоторыми примерами данных каталога являются учетные записи пользователей, каталоги продуктов, реестры устройств *IoT* и системы ведомостей материалов. Атрибуты этих данных могут различаться и могут изменяться со временем в соответствии с требованиями приложения.

Рассмотрим пример каталога продукции поставщика автомобильных запчастей. Каждая часть может иметь свои собственные атрибуты в дополнение к общим для всех частей атрибутам. Кроме того, атрибуты конкретной детали могут измениться в следующем году, когда будет выпущена новая модель. *Azure Cosmos DB* поддерживает гибкие схемы и иерархические данные, поэтому хорошо подходит для хранения данных каталога продуктов (рис. 7.2) [10].

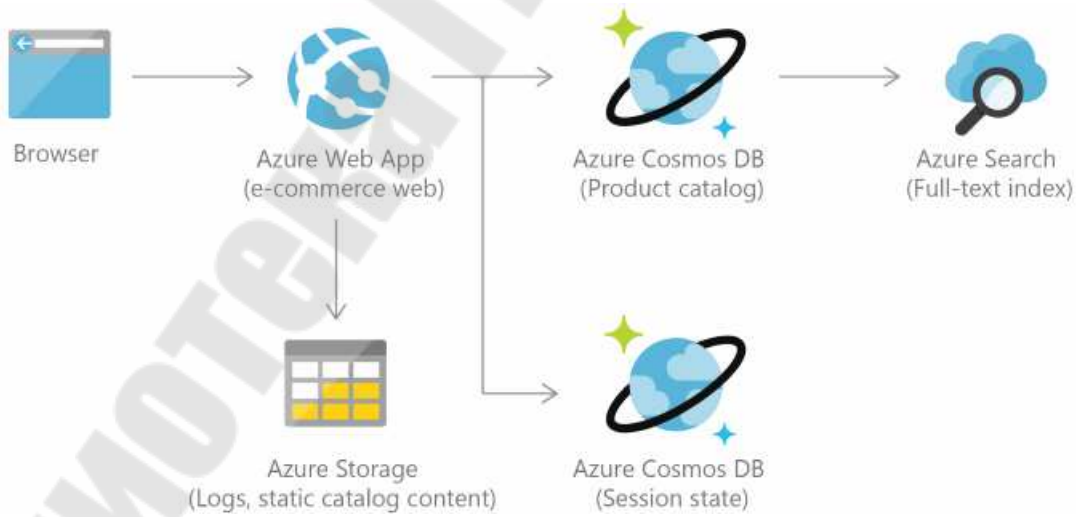

Рис. 7.2. Схема хранения данных каталога продуктов

Azure Cosmos DB часто используется для источников событий, для поддержки архитектур, управляемых событиями, с помощью функции канала изменений. Канал изменений предоставляет ниже-

стоящим микрослужбам возможность надежно и постепенно считывать вставки и обновления (например, события заказа), сделанные в *Azure Cosmos DB*. Эту функциональность можно использовать для предоставления постоянного хранилища событий в качестве брокера сообщений для событий, изменяющих состояние, и управления рабочим процессом обработки порядка между многими микросервисами (которые могут быть реализованы как бессерверные функции *Azure*) (рис. 7.3) [10].

Кроме того, данные, хранящиеся в *Azure Cosmos DB*, можно интегрировать с *HDInsight* для анализа больших данных с помощью заданий *Apache Spark*.

Уровень базы данных – важнейший компонент игровых приложений. Современные игры выполняют графическую обработку на мобильных или консольных клиентах, но полагаются на облако для доставки настроенного и персонализированного контента, такого как внутриигровая статистика, интеграция с социальными сетями и таблицы лидеров. В играх часто требуется задержка в одну миллисекунду для чтения и записи, чтобы обеспечить увлекательный игровой процесс. База данных игр должна быть быстрой, способной справляться с резкими скачками частоты запросов во время запуска новых игр и обновлений функций.

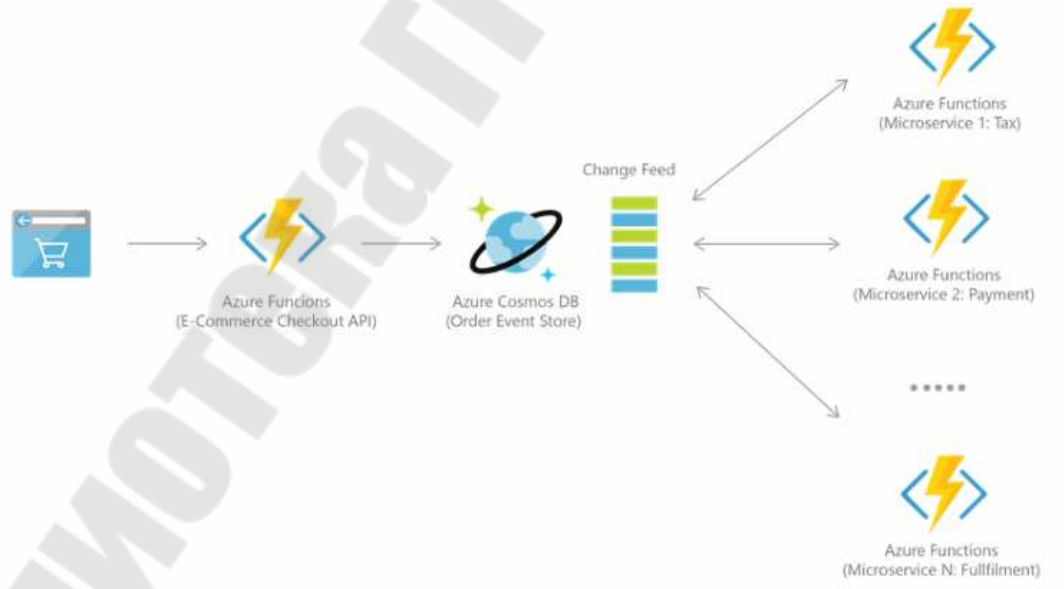

Рис. 7.3. Схема рабочего процесса обработки порядка

Azure Cosmos DB предоставляет разработчикам игр следующие преимущества (рис. 7.4) [10]:

• *Azure Cosmos DB* дает возможность эластично увеличивать или уменьшать производительность, что позволяет играм обрабатывать обновление профиля и статистики от десятков до миллионов игроков одновременно, выполняя один вызов *API*;

• *Azure Cosmos DB* поддерживает чтение и запись в миллисекундах, чтобы избежать задержек во время игры;

• автоматическая индексация *Azure Cosmos DB* позволяет выполнять фильтрацию по нескольким различным свойствам в режиме реального времени, например, находить игроков по их внутренним идентификаторам или запросам на основе членства игрока в гильдии, что возможно без создания сложной инфраструктуры индексации или сегментирования;

• социальные функции, включая сообщения в игровом чате, членство в гильдиях игроков, завершенные задачи, таблицы лидеров и социальные графики, легче реализовать с помощью гибкой схемы;

• *Azure Cosmos DB* как управляемая платформа как услуга (*PaaS*) требует минимальных усилий по настройке и управлению, чтобы обеспечить быструю итерацию и сократить время вывода на рынок.

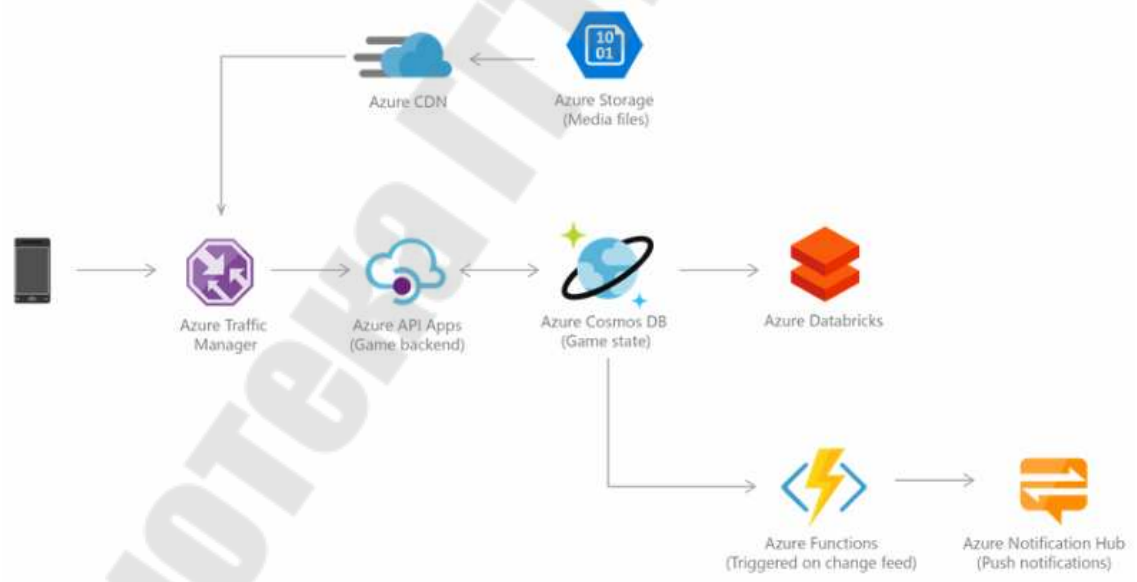

Рис. 7.4. Преимущества *Azure Cosmos DB*

Azure Cosmos DB обычно используется в веб-приложениях и мобильных приложениях и хорошо подходит для моделирования социальных взаимодействий, интеграции со сторонними службами и для создания богатого персонализированного опыта. Пакеты *SDK Cosmos*

DB можно использовать для создания многофункциональных приложений для *iOS* и *Android* с помощью популярной платформы *Xamarin*.

Распространенный вариант использования *Azure Cosmos DB* – хранить и запрашивать пользовательский контент (*UGC*) для веб-приложений, мобильных приложений и приложений социальных сетей. Некоторыми примерами пользовательского контента являются сеансы чата, твиты, сообщения в блогах, рейтинги и комментарии. Часто пользовательский контент в приложениях социальных сетей представляет собой смесь текста произвольной формы, свойств, тегов и отношений, неограниченных жесткой структурой. Такой контент, как чаты, комментарии и сообщения, можно хранить в *Cosmos DB* без необходимости преобразований или сложных объектов в слои реляционного сопоставления. Свойства данных могут быть легко добавлены или изменены в соответствии с требованиями по мере того как разработчики повторяют код приложения, что способствует быстрой разработке.

Приложения, которые интегрируются со сторонними социальными сетями, должны реагировать на изменение схем из этих сетей. Поскольку данные по умолчанию автоматически индексируются в *Cosmos DB*, они готовы к запросу в любое время. Следовательно, эти приложения обладают гибкостью для получения прогнозов в соответствии с их потребностями.

Многие социальные приложения работают в глобальном масштабе и могут демонстрировать непредсказуемые шаблоны использования. Гибкость масштабирования хранилища данных важна, поскольку уровень приложения масштабируется в соответствии с потребностями пользователей. Можно масштабировать, добавляя дополнительные разделы данных в учетной записи *Cosmos DB*. Возможно создание дополнительных учетных записей *Cosmos DB* в нескольких регионах (рис. 7.5) [10].

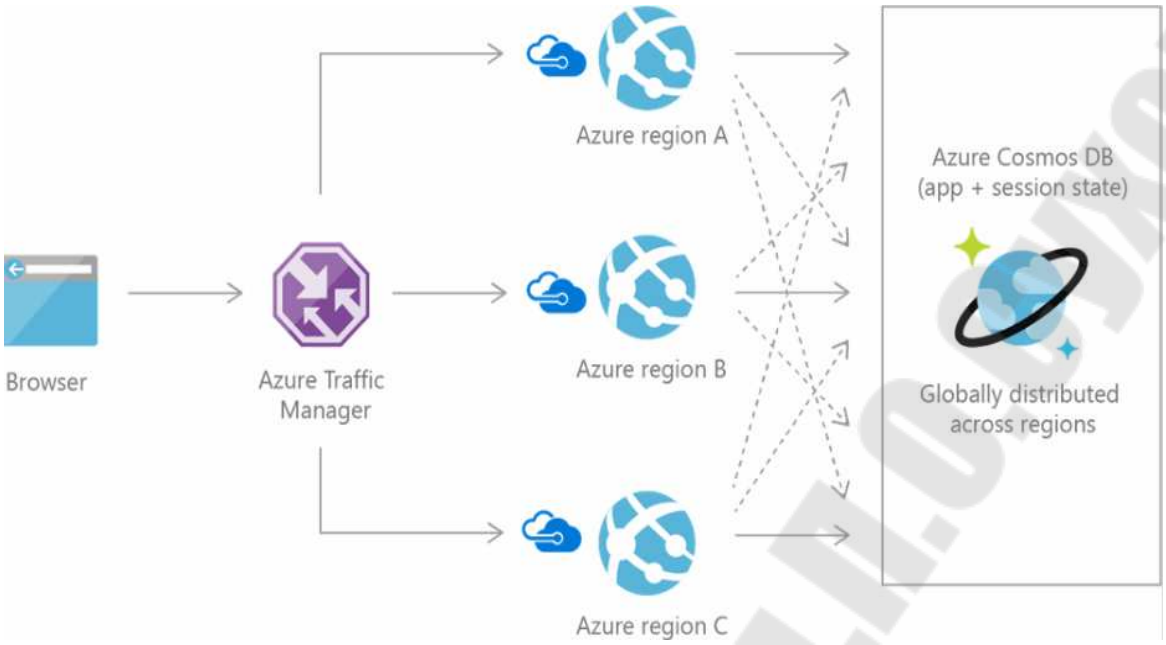

Рис. 7.5. Гибкость масштабирования хранилища *Azure Cosmos DB*

Модель ресурсов *Azure Cosmos DB*

Контейнер *Azure Cosmos* – это фундаментальная единица масштабируемости. Вы можете виртуально иметь неограниченную выделенную пропускную способность и хранилище в контейнере. *Azure Cosmos DB* прозрачно разделяет контейнер с помощью указанного вами логического ключа раздела, чтобы эластично масштабировать подготовленную пропускную способность и хранилище.

В настоящее время вы можете создать максимум 50 учетных записей *Azure Cosmos* в рамках подписки *Azure*. Одна учетная запись *Azure Cosmos* может виртуально управлять неограниченным объемом данных и подготовленной пропускной способностью. Для управления данными и подготовленной пропускной способностью вы можете создать одну или несколько баз данных Azure Cosmos под своей учетной записью, а в этой базе данных есть возможность создать один или несколько контейнеров. На рис. 7.6 [10] показана иерархия элементов в учетной записи *Azure Cosmos*.

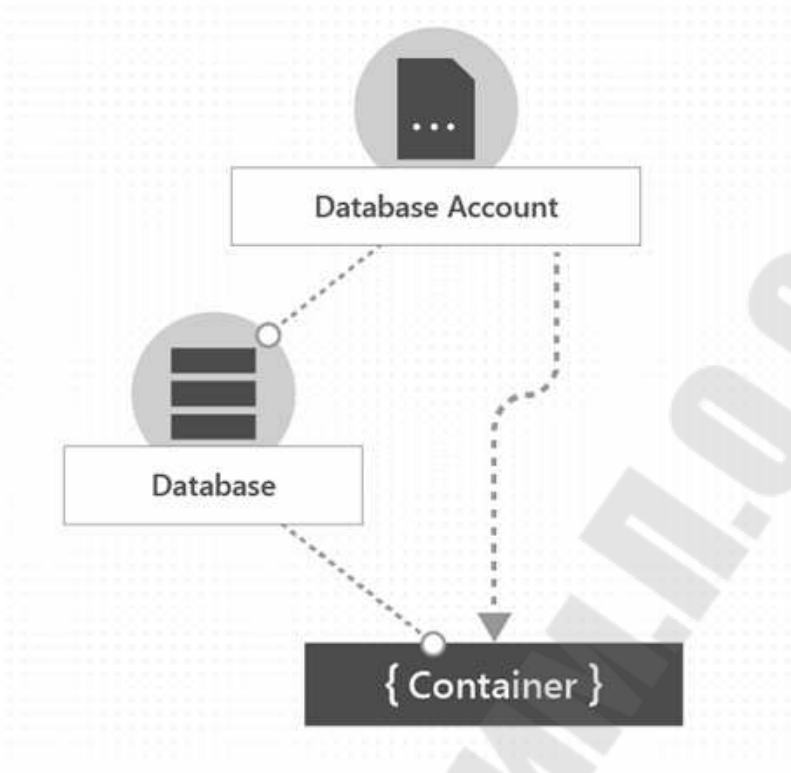

Рис. 7.6. Иерархия элементов в учетной записи *Azure Cosmos*

После создания учетной записи в рамках подписки *Azure* вы можете управлять данными в своей учетной записи, создавая базы данных, контейнеры и элементы.

На рис. 7.7 [10] показана иерархия различных сущностей в учетной записи *Azure Cosmos DB*.

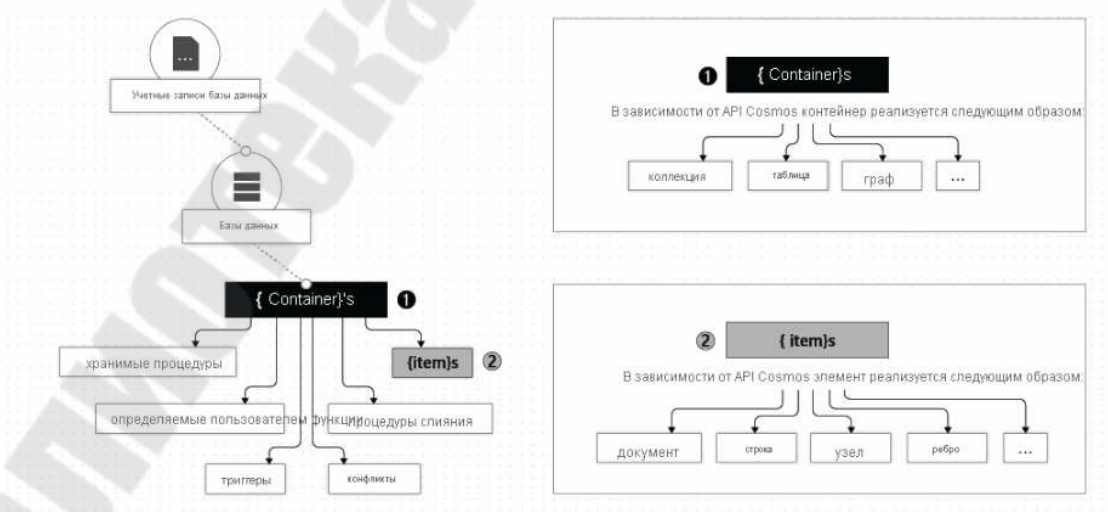

Рис. 7.7. Иерархия иерархия различных сущностей в учетной записи *Azure Cosmos DB*

Можно создать одну или несколько баз данных *Azure Cosmos* под своей учетной записью. База данных аналогична пространству имен. База данных – это единица управления для набора контейнеров *Azure Cosmos*. В табл. 7.1 показано, как база данных сопоставляется с различными объектами, зависящими от *API*.

Таблица 7.1

Сущность *Azure Cosmos DB*	***API SQL***	***API Cassandra***	***API Azure Cosmos DB* для *MongoDB***	***API Gremlin***	***API* таблиц**
База данных *Azure Cosmos*	База данных	Пространство ключей	База данных	База данных	Н/Д

Вы можете взаимодействовать с базой данных *Azure Cosmos* с помощью *API*-интерфейсов *Azure Cosmos*, как описано в табл. 7.2.

Таблица 7.2

API*-интерфейсы *Azure Cosmos

Операция	***Azure CLI***	***API SQL***	***API Cassandra***	***API Azure Cosmos DB* для *MongoDB***	***API Gremlin***	***API* таблиц**
Перечисление всех баз данных	Да	Да	Да (база данных соответствует пространству имен)	Да	Н/Д	Н/Д
Чтение базы данных	Да	Да	Да (база данных соответствует пространству имен)	Да	Н/Д	Н/Д
Создание базы данных	Да	Да	Да (база данных соответствует пространству имен)	Да	Н/Д	Н/Д
Обновление базы данных	Да	Да	Да (база данных соответствует пространству имен)	Да	Н/Д	Н/Д

Контейнер Azure Cosmos – это единица масштабируемости как для подготовленной пропускной способности, так и для хранилища. Контейнер разделяется по горизонтали, а затем реплицируется в нескольких регионах. Элементы, которые вы добавляете в контейнер, автоматически группируются в логические разделы, распределяю-

щиеся по физическим разделам в зависимости от ключа раздела. Пропускная способность контейнера равномерно распределяется по физическим разделам.

При создании контейнера вы настраиваете пропускную способность в одном из следующих режимов:

1) режим выделенной подготовленной пропускной способности: пропускная способность, предоставленная для контейнера, зарезервирована исключительно для этого контейнера и поддерживается соглашениями об уровне обслуживания;

2) режим общей подготовленной пропускной способности: эти контейнеры совместно используют подготовленную пропускную способность с другими контейнерами в той же базе данных, то есть предоставленная пропускная способность в базе данных распределяется между всеми контейнерами «общей пропускной способности».

Контейнер *Azure Cosmos* может эластично масштабироваться независимо от того, создаете ли вы контейнеры с использованием выделенных или совместно подготовленных режимов пропускной способности или нет.

Контейнер – это не зависящий от схемы контейнер элементов. Элементы в контейнере могут иметь произвольные схемы. Например, предмет, представляющий человека, и предмет, представляющий автомобиль, могут быть помещены в один и тот же контейнер. По умолчанию все элементы, которые вы добавляете в контейнер, автоматически индексируются без необходимости явного управления индексом или схемой. Вы можете настроить поведение индексации, настроив политику индексации в контейнере.

Можно установить время жизни для выбранных элементов в контейнере или для всего контейнера, чтобы корректно удалить эти элементы из системы. *Azure Cosmos DB* автоматически удаляет элементы по истечении срока их действия. Это также гарантирует, что запрос, выполненный в контейнере, не вернет просроченные элементы в пределах фиксированной границы.

Можно использовать канал изменений, чтобы подписаться на журнал операций, управляемый для каждого логического раздела контейнера. Лента изменений предоставляет журнал всех обновлений, выполненных в контейнере, а также предоставляет изображения элементов до и после. Возможно настроить продолжительность хранения для канала изменений с помощью политики канала изменений в контейнере.

Имеется возможность регистрировать хранимые процедуры, триггеры, пользовательские функции (*UDF*) и процедуры слияния для своего контейнера.

Можно указать уникальное ограничение ключа для своего контейнера *Azure Cosmos*. Создавая политику уникального ключа, вы гарантируете уникальность одного или нескольких значений для каждого логического ключа раздела. При создании контейнера с использованием политики уникального ключа новые или обновленные элементы со значениями, которые дублируют значения, указанные в ограничении уникального ключа, не могут быть созданы.

Контейнер специализируется на объектах, специфичных для *API*, как показано в табл. 7.3.

Таблица 7.3

Сущность *Azure Cosmos DB*	***API SQL***	***API Cassandra***	***API Azure Cosmos DB* для *MongoDB***	***API Gremlin***	***API* таблиц**
Контейнер *Azure Cosmos*	Контейнер	Таблица	Коллекция	График	Таблица

Контейнер *Azure Cosmos* имеет набор свойств, определенных системой. В зависимости от того, какой *API* используется, некоторые свойства могут не отображаться напрямую. В табл. 7.4 описан список свойств, определенных системой.

Таблица 7.4

Свойства контейнера *Azure Cosmos*

Системные свойства	**Генерируемые системой или настраиваемые пользователем**	**Цель**	***API SQL***	***API Cassandra***	**API Azure Cosmos DB для *MongoDB***	***API Gremlin***	***API* таблиц**
_rid	Генерируемые системой	Уникальный идентификатор контейнера	Да	Нет	Нет	Нет	Нет
_etag	Генерируемые системой	Тег сущности, используемый для управления оптимистической блокировкой	Да	Нет	Нет	Нет	Нет
_ts	Генерируемые системой	Метка времени последнего обновления контейнера	Да	Нет	Нет	Нет	Нет

Окончание табл. 7.4

Системные свойства	**Генерируемые системой или настраиваемые пользователем**	**Цель**	***API SQL***	***API Cassandra***	**API Azure Cosmos DB для *MongoDB***	***API Gremlin***	***API* таблиц**
_self	Генерируемые системой	Адресуемый URI контейнера	Да	Нет	Нет	Нет	Нет
идентификатор	Настраиваемые пользователем	Определяемое пользователем уникальное имя контейнера	Да	Да	Да	Да	Да
IndexingPolicy	Настраиваемые пользователем	Позволяет изменять путь, тип и режим ин-декса	Да	Нет	Нет	Нет	Да
timeToLive	Настраиваемые пользователем	Позволяет автоматически удалять элементы из контейнера через заданное время	Да	Нет	Нет	Нет	Да
ChangeFeedPolicy	Настраиваемые пользователем	Используется для считывания изменений, внесенных в элементы в контейнере	Да	Нет	Нет	Нет	Да
UniqueKeyPolicy	Настраиваемые пользователем	Гарантируют уникальность одного или нескольких значений в пределах логической секции	Да	Нет	Нет	Нет	Да

Контейнер *Azure Cosmos* поддерживает следующие операции при использовании любого из *API Azure Cosmos* (табл. 7.5).

Таблица 7.5

Операция	***Azure Ш***	***API SQL***	***API Cassandra***	***API Azure Cosmos DB* для *MongoDB***	***API Gremlin***	***API* таблиц**
Перечисление контейнеров в базе данных	Да	Да	Да	Да	Н/Д	Н/Д
Считывание контейнера	Да	Да	Да	Да	Н/Д	Н/Д
Создает контейнер.	Да	Да	Да	Да	Н/Д	Н/Д
Обновлениеконтейнера	Да	Да	Да	Да	Н/Д	Н/Д
Удаление контейнера	Да	Да	Да	Да	Н/Д	Н/Д

В зависимости от того, какой *API* вы используете, элемент *Azure Cosmos* может представлять либо документ в коллекции, либо строку в таблице, либо узел или край в графе. В табл. 7.6 показано сопоставление конкретных сущностей *API* с элементом *Azure Cosmos*.

Таблица 7.6

Сущность *Azure Cosmos DB*	***API SQL***	***API Cassandra***	***API Azure Cosmos DB* для *MongoDB***	***API Gremlin***	***API* таблиц**
Элемент *Azure Cosmos*	*Item*	Строка	Документ	Узел или ребро	*Item*

Каждый элемент *Azure Cosmos* имеет следующие системные свойства. В зависимости от того, какой *API* вы используете, некоторые из них могут быть недоступны напрямую (табл. 7.7).

Таблица 7.7

Системные свойства элементов *Azure Cosmos*

Системные свойства	**Генерируемые системой или настраиваемые пользователем**	**Цель**	***API SQL***	***API Cassandra***	***API Azure Cosmos DB* для *MongoDB***	***API Gremlin***	***API* таблиц**
_rid	Генерируемые системой	Уникальный идентификатор элемента	Да	Нет	Нет	Нет	Нет
_etag	Генерируемые системой	Тег сущности, используемый для управления оптимистической блокировкой	Да	Нет	Нет	Нет	Нет
_ts	Генерируемые системой	Метка времени последнего обновления элемента	Да	Нет	Нет	Нет	Нет
идентификатор	Можно использовать	Определяемое пользователем уникальное имя в пределах логической секции	Да	Да	Да	Да	Да
определяемые пользователем свойства	Определяемые пользователем маршруты	Определяемые пользователем свойства в собственном формате *API* (включая *JSON, BSON* и *CQL*)	Да	Да	Да	Да	Да

Элементы *Azure Cosmos* поддерживают приведенные операции: вставка, замена, удаление, обновление или вставка, чтение. Для выполнения этих операций можно использовать любой из интерфейсов *API Azure Cosmos*, кроме *Azure CLI*.

Глобальное распределение данных с помощью *Azure Cosmos DB*

Современные приложения должны быть очень отзывчивыми и всегда подключены к Интернету. Чтобы добиться высокой скорости и доступности, экземпляры этих приложений нужно развернуть в центрах обработки данных, которые находятся рядом с их пользователями. Эти приложения называются глобально распределенными. Глобально распределенным приложениям требуется глобально распределенная база данных, которая может прозрачно реплицировать данные в любой точке мира, чтобы позволить приложениям работать с копией данных, близкой к их пользователям.

Azure Cosmos DB это глобально распределенная система баз данных, которая позволяет вам читать и записывать данные из локальных реплик вашей базы данных. *Azure Cosmos DB* прозрачно реплицирует данные во все регионы, связанные с вашей учетной записью *Cosmos*.

Azure Cosmos DB также предназначена для обеспечения высокой скорости, эластичной масштабируемости пропускной способности, четко определенной семантики для согласованности данных и высокой доступности. Если приложению требуется быстрое время отклика в любой точке мира, если ему требуется всегда быть в сети и требуется неограниченная и гибкая масштабируемость пропускной способности и хранилища, нужно создать приложение на базе *Azure Cosmos DB*.

Имеется возможность настройки своих баз данных для глобального распределения и доступности в любом из регионов *Azure*. Чтобы уменьшить задержку, разместите данные рядом с местом, где находятся ваши пользователи. Выбор необходимых регионов зависит от глобального охвата вашего приложения и от того, где находятся ваши пользователи. *Cosmos DB* прозрачно реплицирует данные во все регионы, связанные с вашей учетной записью *Cosmos*: предоставляет единый системный образ вашей глобально распределенной базы данных и контейнеров *Azure Cosmos*, который ваше приложение может читать и записывать локально.

С помощью *Azure Cosmos DB* вы можете добавлять или удалять регионы, связанные с вашей учетной записью, в любое время. Ваше приложение не нужно приостанавливать или повторно развертывать, чтобы добавить или удалить регион.

На рис. 7.8. изображен пример глобального распределения данных с помощью *Azure Cosmos DB*.

Создавайте глобальные активные приложения. Благодаря новому протоколу репликации записи с несколькими регионами, каждый регион поддерживает как запись, так и чтение. Возможность записи в нескольких регионах также предусматривает:

- неограниченную эластичную масштабируемость записи и чтения;
- высокую доступность для чтения и записи по всему миру;
- гарантированные операции чтения и записей, выполненные с высокой скоростью.

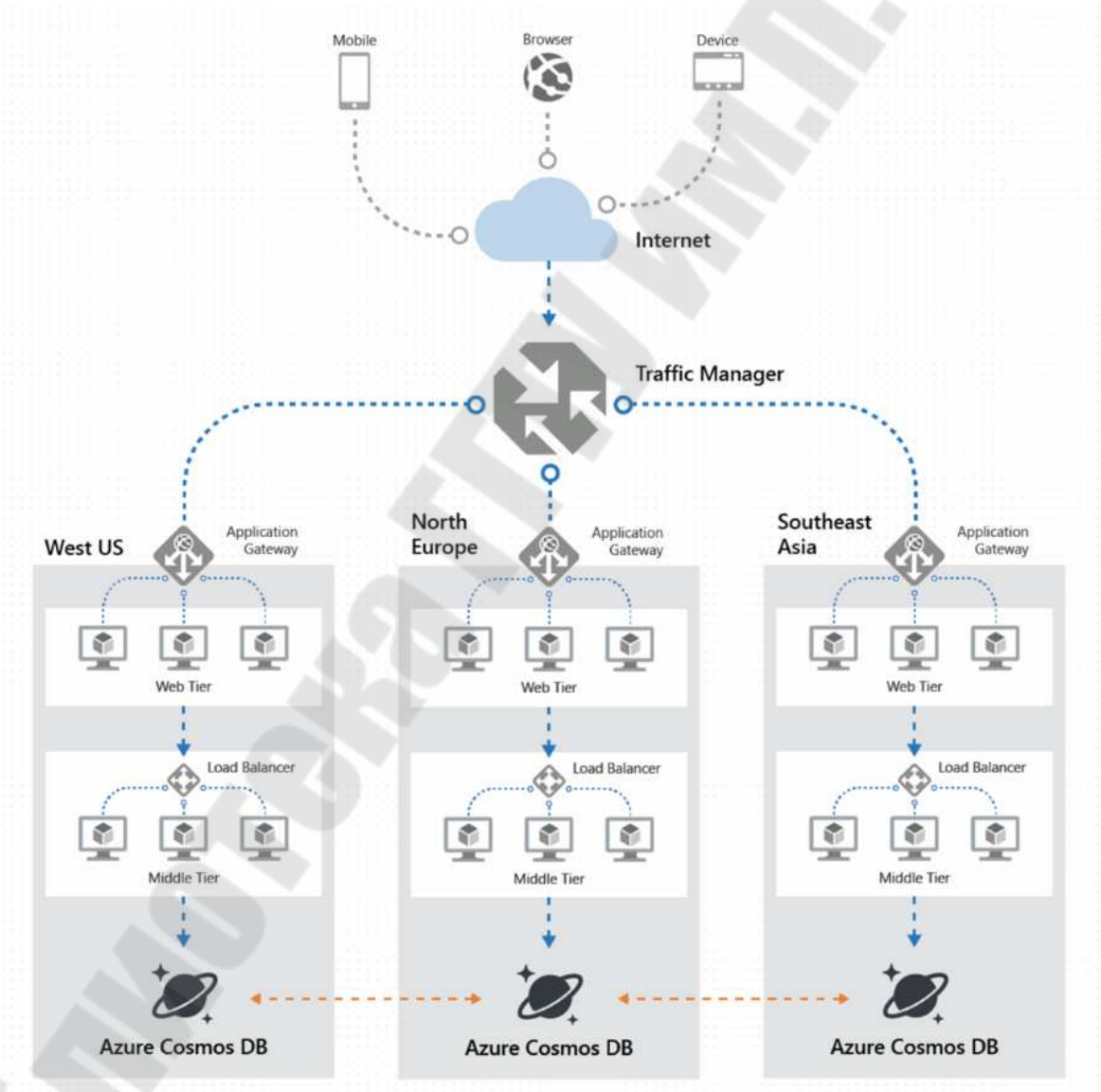

Рис. 7.8. Глобальное распределение данных с помощью *Azure Cosmos DB*

При добавлении и удалении региона в своей учетной записи *Azure Cosmos* и из нее приложение не нужно повторно развертывать или приостанавливать, оно всегда остается высокодоступным.

Есть возможность создавать приложения с высокой степенью реагирования. Приложение может выполнять чтение и запись почти в реальном времени для всех регионов, которые вы выбрали для своей базы данных. *Azure Cosmos DB* внутренне обрабатывает репликацию данных между регионами с гарантиями уровня согласованности выбранного уровня.

Запуск базы данных в нескольких регионах по всему миру увеличивает доступность базы данных. Если один регион недоступен, другие регионы автоматически обрабатывают запросы приложений.

Поддерживается непрерывность бизнеса во время региональных отключений. *Azure Cosmos DB* поддерживает автоматическое переключение во время регионального сбоя. Во время регионального сбоя *Azure Cosmos DB* продолжает поддерживать соглашения об уровне обслуживания, связанные с задержкой, доступностью, согласованностью и пропускной способностью. Чтобы обеспечить высокую доступность всего вашего приложения, *Cosmos DB* предлагает *API* ручной отработки отказа для имитации сбоя в регионе.

Можно сделать каждый регион доступным для записи и эластично масштабировать чтение и запись по всему миру. Пропускная способность, которую ваше приложение настраивает в базе данных или контейнере *Azure Cosmos*, предоставляется во всех регионах, связанных с вашей учетной записью *Azure Cosmos*. Предоставленная пропускная способность гарантируется финансовыми соглашениями об уровне обслуживания.

Есть возможность выбрать одну из нескольких четко определенных моделей согласованности. Протокол репликации *Azure Cosmos DB* предлагает пять четко определенных, практичных и интуитивно понятных моделей согласованности. В каждой модели есть компромисс между согласованностью и производительностью. Используйте эти модели для легкого создания глобально распределенных приложений.

Уровни согласованности в *Azure Cosmos DB*

Распределенные базы данных, которые полагаются на репликацию для обеспечения высокой доступности, низкой задержки или того и другого, должны найти фундаментальный компромисс между согласованностью чтения, доступностью, задержкой и пропускной спо-

собностью. Линеаризуемость модели строгой согласованности – золотой стандарт программируемости данных. Однако это устанавливает высокую цену из-за более высоких задержек записи по причине необходимости реплицирования и фиксирования данных на больших расстояниях. Сильная согласованность также может пострадать из-за снижения доступности (во время сбоев), поскольку данные не могут реплицироваться и фиксироваться в каждом регионе. Конечная согласованность обеспечивает более высокую доступность и лучшую производительность, но ее сложнее программировать приложения, поскольку данные могут не быть полностью согласованными во всех регионах.

Большинство коммерчески доступных распределенных баз данных *NoSQL*, доступных сегодня на рынке, обеспечивают только надежную и конечную согласованность. *Azure Cosmos DB* предлагает пять четко определенных уровней. Уровни бывают от самого сильного до самого слабого:

- уровень согласованности «Строгая»;
- ограниченное устаревание;
- сеанс;
- постоянный префикс;
- итоговая согласованность.

Каждый уровень обеспечивает компромисс между доступностью и производительностью. На рис. 7.9 показаны различные уровни согласованности в виде спектра.

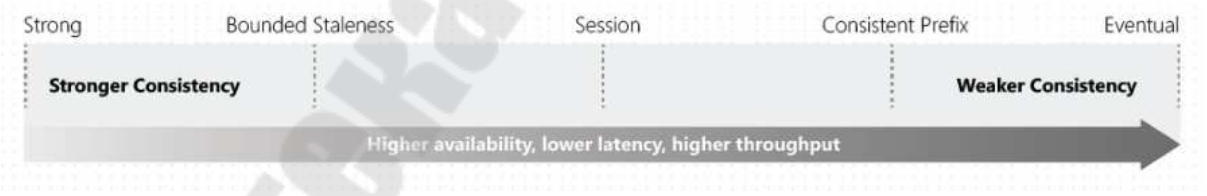

Рис. 7.9. Уровни согласованности в виде спектра

Уровни согласованности гарантируются для всех операций независимо от региона, из которого выполняются операции чтения и записи, количества регионов, связанных с вашей учетной записью *Azure Cosmos*, или от того, настроена ли ваша учетная запись с одной или несколькими регионами записи.

Azure Cosmos DB обеспечивает встроенную поддержку *API*-интерфейсов, совместимых с проводным протоколом, для популяр-

ных баз данных. К ним относятся *MongoDB*, *Apache Cassandra*, *Gremlin* и хранилище таблиц *Azure*. При использовании *API*-интерфейса *Gremlin* и *API*-интерфейса таблиц используется уровень согласованности по умолчанию, настроенный в учетной записи *Azure Cosmos*. Рассмотрим подробнее сведения о сопоставлении уровней согласованности между *Cassandra API* или *API* для *MongoDB* и уровнями согласованности *Azure Cosmos DB*.

Согласованность чтения применяется к одной операции чтения в пределах логического раздела. Операция чтения может выполняться удаленным клиентом или хранимой процедурой.

Можно настроить уровень согласованности по умолчанию для своей учетной записи *Azure Cosmos* в любое время. Уровень согласованности по умолчанию, настроенный для вашей учетной записи, применяется ко всем базам данных и контейнерам *Azure Cosmos* в этой учетной записи. Все операции чтения и запросы к контейнеру или базе данных по умолчанию используют указанный уровень согласованности.

Azure Cosmos DB гарантирует, что 100 % запросов на чтение соответствуют гарантии согласованности для выбранного уровня согласованности. Точные определения пяти уровней согласованности в *Azure Cosmos DB* с использованием языка спецификаций *TLA+* представлены в репозитории *azure-cosmos-tla* на *GitHub*.

Сильная согласованность дает гарантию линеаризуемости. Линеаризуемость относится к одновременному обслуживанию запросов. При чтении гарантированно будет возвращена самая последняя зафиксированная версия элемента. Клиент никогда не видит незафиксированной или частичной записи. Пользователи всегда гарантированно прочитают последнюю совершенную запись.

В случае согласованности с ограниченным устареванием при чтении соблюдается гарантия согласованного префикса. Чтения могут отставать от записи не более чем на *K* версий (то есть «обновлений») элемента или на временной интервал *T*, в зависимости от того, какая из них будет достигнута раньше. Другими словами, когда вы выбираете ограниченное устаревание, его можно настроить двумя способами:

1) количество версий (*K*) элемента;

2) чтение временного интервала (*T*) может отставать от записи.

Для учетной записи с одним регионом минимальное значение *K* и *T* составляет 10 операций записи или 5 секунд. Для мультирегио-

нальных учетных записей минимальное значение *K* и *T* составляет 100 000 операций записи или 300 секунд.

Ограниченное устаревание предлагает полный, глобальный порядок за пределами «окна устаревания». Когда клиент выполняет операции чтения в области, которая принимает записи, гарантии, обеспечиваемые согласованностью ограниченного устаревания, идентичны гарантиям строгой согласованности. По мере приближения окна устаревания либо времени, либо обновлений, в зависимости от того, что ближе, служба будет ограничивать новые записи, чтобы позволить репликации наверстать упущенное и соблюдать гарантию согласованности.

Внутри окна устаревания *Bounded Staleness* обеспечивает следующие гарантии согласованности:

- согласованность для клиентов в одном регионе для учетной записи с одним регионом записи = *Strong*;
- согласованность для клиентов в разных регионах для учетной записи с одной областью записи = согласованный префикс;
- согласованность для клиентов, выполняющих запись в один регион для учетной записи с несколькими регионами записи = согласованный префикс;
- согласованность для клиентов, осуществляющих запись в разные регионы для учетной записи с несколькими регионами записи = возможная.

Ограничение устаревания часто выбирается глобально распределенными приложениями, которые ожидают низких задержек записи, но требуют полной гарантии глобального порядка. Ограниченное устаревание отлично подходит для приложений, включающих групповую совместную работу и совместное использование, биржевой тикер, публикацию-подписку, постановку в очередь и т. д.

При согласованности сеанса в рамках одного сеанса клиентского сеанса чтения соблюдают гарантии согласованного префикса, монотонного чтения, монотонной записи, чтения собственных записей и записей после чтения. Это предполагает один сеанс «записи» или совместное использование маркера сеанса для нескольких записывающих.

Клиенты вне сеанса, выполняющие запись, увидят следующие гарантии:

- согласованность для клиентов в одном регионе для учетной записи с одним регионом записи, то есть согласованный префикс;
- согласованность для клиентов в разных регионах для учетной записи с одной областью записи, то есть согласованный префикс;

• согласованность для клиентов, выполняющих запись в один регион для учетной записи с несколькими регионами записи, то есть согласованный префикс;

• согласованность для клиентов, осуществляющих запись в несколько регионов для учетной записи с несколькими регионами записи, то есть итоговая;

• согласованность для клиентов, использующих интегрированный кеш *Azure Cosmos DB*, то есть итоговая.

Согласованность сеанса – это наиболее широко используемый уровень согласованности как для отдельных регионов, так и для глобально распределенных приложений. Он обеспечивает задержки записи, доступность и пропускную способность чтения, сопоставимые с конечной согласованностью, также обеспечивает гарантии согласованности, которые соответствуют потребностям приложений, написанных для работы в контексте пользователя.

В варианте согласованного префикса возвращаемые обновления содержат некоторый префикс всех обновлений без пробелов. Постоянный уровень согласованности префикса гарантирует, что при чтении все будет записываться по порядку.

Если записи выполнялись в порядке *A*, *B*, *C*, то клиент видит либо *A*, *A*, *B*, либо *A*, *B*, *C*, но никогда не будет перестановок вне порядка, таких как *A*, *C* или *B*, *A*, *C*. Согласованный префикс обеспечивает задержки записи, доступность и пропускную способность чтения, сопоставимые с конечной согласованностью, также обеспечивает гарантии порядка, которые соответствуют потребностям сценариев, где порядок важен.

Ниже приведены гарантии согласованности для согласованного префикса:

• согласованность для клиентов в одном регионе для учетной записи с одним регионом записи, то есть согласованный префикс;

• согласованность для клиентов в разных регионах для учетной записи с одной областью записи, то есть согласованный префикс;

• согласованность для клиентов, выполняющих запись в один регион для учетной записи с несколькими регионами записи, то есть согласованный префикс;

• согласованность для клиентов, осуществляющих запись в несколько регионов для учетной записи с несколькими регионами записи, то есть итоговая.

В случае возможной согласованности нет никакой гарантии упорядочивания чтений. При отсутствии дальнейших операций записи реплики в конечном итоге сходятся.

Конечная согласованность – это самая слабая форма согласованности, потому что клиент может читать значения, которые старше чем те, которые он читал раньше. Конечная согласованность идеальна, когда приложение не требует каких-либо гарантий заказа. Примеры включают количество ретвитов, лайков или комментариев без цепочки.

На практике часто можно получить более строгие гарантии согласованности. Гарантии согласованности для операции чтения соответствуют актуальности и упорядоченности состояния базы данных, которое вы запрашиваете. Согласованность чтения связана с порядком и распространением операций записи и обновления.

Если в базе данных нет операций записи, операция чтения с конечным, сеансовым или согласованным уровнем согласованности префикса, скорее всего, даст те же результаты, что и операция чтения с высоким уровнем согласованности.

Если учетная запись *Azure Cosmos* настроена с уровнем согласованности, отличным от строгой согласованности, вы можете узнать вероятность того, что ваши клиенты могут получить надежные и согласованные чтения для ваших рабочих нагрузок, просмотрев метрику вероятностно ограниченной устойчивости (*PBS*).

Вероятностная ограниченная устаревшая информация показывает, насколько вероятна конечная согласованность. Эта метрика дает представление о том, как часто вы можете добиваться большей согласованности, чем уровень согласованности, на который настроена учетная запись *Azure Cosmos*. Другими словами, вы можете видеть вероятность (измеряемую в миллисекундах) получения строго согласованного чтения для комбинации областей записи и чтения.

Задержка чтения для всех уровней согласованности гарантированно составляет менее 10 миллисекунд на 99-м процентиле. Средняя задержка чтения на 50-м процентиле обычно составляет 4 миллисекунды или меньше.

Задержка записи для всех уровней согласованности гарантированно составляет менее 10 миллисекунд на 99-м процентиле. Средняя задержка записи на 50-м процентиле обычно составляет 5 миллисекунд или меньше. Учетные записи *Azure Cosmos*, охватывающие несколько регионов и настроенные с высокой согласованностью, являются исключением из этой гарантии.

Для учетных записей *Azure Cosmos*, настроенных на строгую согласованность с более чем одним регионом, задержка записи равна двукратному времени приема-передачи (*RTT*) между любым из двух

самых дальних регионов плюс 10 миллисекунд. Высокий сетевой *RTT* между регионами приведет к более высокой задержке для запросов *Cosmos DB*, поскольку при строгой согласованности операция завершается только после того, как она была зафиксирована для всех регионов в учетной записи.

Для сильного и ограниченного устаревания чтения выполняются против двух реплик в наборе из четырех реплик (меньшинство) для обеспечения гарантии согласованности. В результате для того же количества единиц запроса пропускная способность чтения для строгого и ограниченного устаревания составляет половину от других уровней согласованности.

Для таких типов операции записи, как вставка, замена, вставка и удаление, пропускная способность записи для блоков запроса идентична для всех уровней согласованности. Для обеспечения высокой согласованности нужно фиксировать изменения в каждом регионе (глобальное большинство), для всех же других уровней согласованности используется локальное большинство (три реплики в наборе из четырех реплик) (табл. 7.8).

Таблица 7.8

Реплики и уровни создания согласованности

Уровень согласованности	Операции чтения кворума	Операции записи кворума
Строгая	Местное меньшинство	Глобальное большинство
Ограниченное устаревание	Местное меньшинство	Местное большинство
Сеанс	Одна реплика (с использованием токена сеанса)	Местное большинство
Постоянный префикс	Одна реплика	Местное большинство
Итоговая	Одна реплика	Местное большинство

В среде глобально распределенной базы данных существует прямая взаимосвязь между уровнем согласованности и долговечностью данных в случае сбоя в масштабах всего региона. При разработке плана обеспечения непрерывности бизнеса разработчик должен знать максимально допустимое время, в течение которого приложение полностью восстановится после сбоя. Время, необходимое для полного восстановления приложения, называется целевым временем восстановления (*RTO*). Необходимо знать максимальный период по-

следних обновлений данных, при которых приложение может допустить потерю при восстановлении после аварийного события. Период времени обновлений, который вы можете позволить себе потерять, называется целевой точкой восстановления (*RPO*).

В табл. 7.9 определяется взаимосвязь между моделью согласованности и надежностью данных в случае сбоя в масштабах всего региона. Важно отметить, что в распределенной системе даже с высокой согласованностью невозможно иметь распределенную базу данных с нулевым *RPO* и *RTO* из-за теоремы *CAP*.

Таблица 7.9

Взаимосвязи между моделью согласованности и надежности данных

Регионы	**Режим репликации**	**Уровень согласованности**	***RPO***	***RTO***
1	Один или несколько регионов записи	Любой уровень согласованности	< 240 минут	< 1 неделя
>1	Один регион записи	Сеанс, постоянный префикс, случайный	< 15 минут	< 15 минут
>1	Один регион записи	Ограниченное устаревание	*K&T*	< 15 минут
>1	Один регион записи	Уровень согласованности «Строгая»	0	< 15 минут
>1	Несколько регионов записи	Сеанс, постоянный префикс, случайный	< 15 минут	0
>1	Несколько регионов записи	Ограниченное устаревание	*K&T*	0

Учетные записи *Cosmos* с несколькими регионами записи не могут быть настроены для обеспечения строгой согласованности, поскольку распределенная система не может обеспечить нулевое *RPO* и нулевое *RTO*. Кроме того, нет преимуществ задержки записи при использовании строгой согласованности с несколькими регионами, потому что запись в любой регион должна реплицироваться и фиксироваться во всех настроенных регионах в учетной записи. Это приводит к той же задержке, что и для одной учетной записи региона записи.

***Azure Cosmos DB* обеспечивает высокую доступность**

Azure Cosmos DB обеспечивает высокую доступность двумя основными способами:

1) *Azure Cosmos DB* реплицирует данные по регионам, настроенным в учетной записи Cosmos;

2) *Azure Cosmos DB* поддерживает четыре реплики данных в регионе.

В пределах региона *Azure Cosmos DB* поддерживает четыре копии ваших данных в виде реплик в физических разделах, как показано на рис. 7.10:

- данные в контейнерах Azure Cosmos разделены по горизонтали;
- набор разделов – это набор нескольких наборов реплик. В каждом регионе каждый раздел защищен набором реплик, при этом все записи реплицируются и надежно фиксируются большинством реплик. Реплики распределены по 10–20 доменам сбоя;
- каждый раздел во всех регионах реплицируется. Каждый регион содержит все разделы данных контейнера *Azure Cosmos* и может выполнять операции чтения и записи, если включена запись в нескольких регионах.

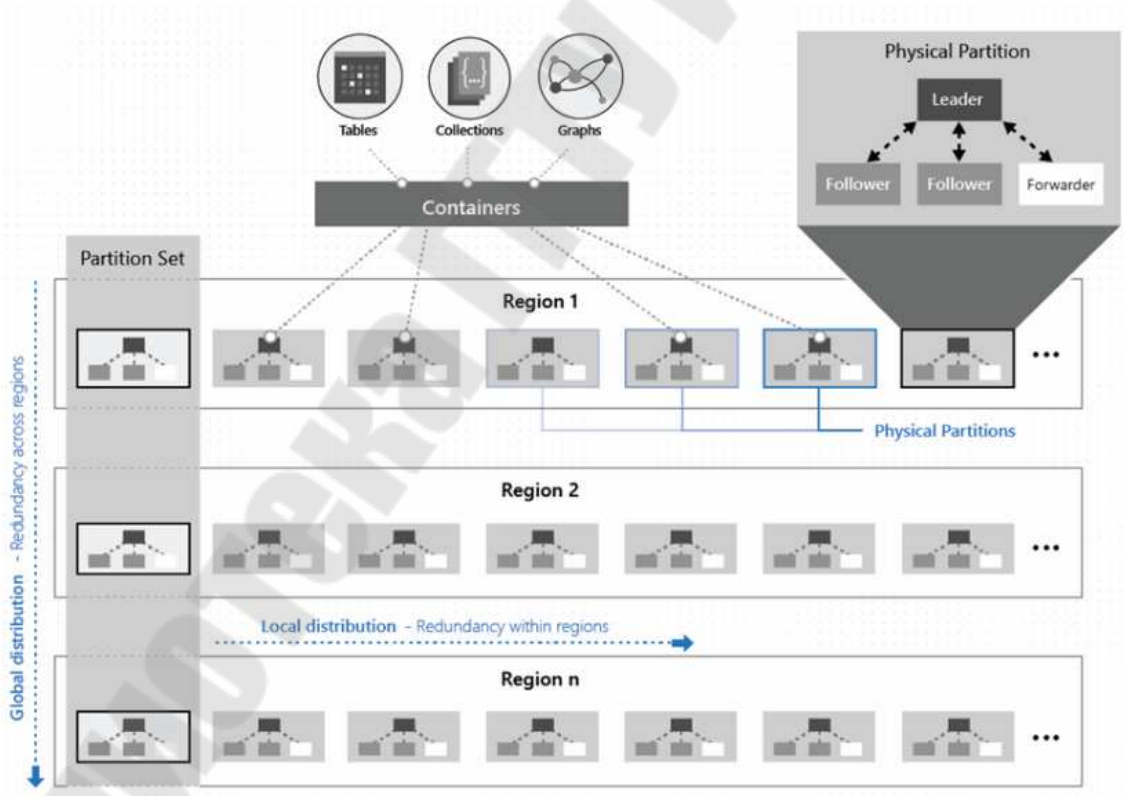

Рис. 7.10. Высокая доступность *Azure Cosmos DB*

Azure Cosmos DB предоставляет комплексные соглашения об уровне обслуживания, которые охватывают пропускную способность,

согласованность и высокую доступность. Для повышения доступности записи настройте свою учетную запись *Azure Cosmos* так, чтобы она имела несколько регионов записи.

В редких случаях региональных сбоев *Azure Cosmos DB* обеспечивает постоянную высокую доступность вашей базы данных. Следующие сведения фиксируют поведение *Azure Cosmos DB* во время простоя в зависимости от конфигурации учетной записи *Azure Cosmos*:

- в *Azure Cosmos DB* перед подтверждением операции записи клиенту данные надежно фиксируются кворумом реплик в регионе, который принимает операции записи;
- учетные записи с несколькими регионами, настроенные с несколькими регионами записи, будут высокодоступными как для записи, так и для чтения. Отработка отказа в регионах обнаруживается и обрабатывается в клиенте *Azure Cosmos DB*. Они также мгновенные и не требуют изменений из приложения;
- учетные записи для одного региона могут потерять доступность из-за сбоя в работе в регионе. Всегда рекомендуется настраивать как минимум два региона (желательно как минимум два региона записи) с вашей учетной записью *Azure Cosmos*, чтобы обеспечить постоянную высокую доступность.

Помимо межрегиональной устойчивости, *Azure Cosmos DB* также поддерживает избыточность зон в поддерживаемых регионах при выборе региона для связи с вашей учетной записью *Azure Cosmos*.

Благодаря поддержке зоны доступности (*AZ*) *Azure Cosmos DB* обеспечит размещение реплик в нескольких зонах в пределах заданного региона, чтобы обеспечить высокую доступность и отказоустойчивость к зональным сбоям. Зоны доступности обеспечивают соглашение об уровне обслуживания 99,995 % без изменения задержки. Нельзя полагаться только на резервирование зоны для достижения региональной отказоустойчивости.

Избыточность зоны можно настроить только при добавлении нового региона в учетную запись *Azure Cosmos*. Для существующих регионов избыточность зоны можно включить, удалив регион, а затем добавив его обратно с включенной избыточностью зоны. Для учетной записи с одним регионом это требует добавления одного дополнительного региона для временного переключения при отказе, а затем удаления и добавления желаемого региона с включенной избыточностью зоны.

При настройке записи в нескольких регионах для своей учетной записи *Azure Cosmos* вы можете выбрать избыточность зоны без дополнительных затрат.

Зоны доступности можно включить через:

1) портал *Azure*;

2) *Azure PowerShell*;

3) *Azure CLI*;

4) шаблоны *Azure Resource Manager*.

При создании высокодоступных приложений нужно учитывать и выполнять следующее:

- ожидаемое поведение пакетов *SDK* для *Azure Cosmos* во время этих событий и возможные конфигурации, которые на него влияют;
- для обеспечения высокой доступности для записи и чтения, настройте свою учетную запись *Azure Cosmos* для охвата как минимум двух регионов с несколькими регионами записи;
- для многорегиональных учетных записей *Azure Cosmos*, для которых настроен регион с одной записью, включите автоматическое переключение при отказе с помощью *Azure CLI* или портала *Azure*. После включения автоматического переключения при возникновении региональной аварии *Cosmos DB* автоматически переключит вашу учетную запись на сбой;
- если учетная запись *Azure Cosmos* является высокодоступной, ваше приложение может быть неправильно спроектировано, чтобы оставаться высокодоступным. Для проверки сквозной высокой доступности приложения в рамках отработки тестирования приложения или аварийного восстановления (*DR*) временно отключите автоматическое переключение при сбое для учетной записи, вызовите переключение вручную с помощью *PowerShell*, *Azure CLI* или *Azure*-портал, а затем отслеживайте аварийное переключение вашего приложения. После завершения вы можете вернуться к основному региону и восстановить автоматический переход на другой ресурс для учетной записи;
- в среде глобально распределенной базы данных существует прямая взаимосвязь между уровнем согласованности и надежностью данных при сбое в масштабах всего региона. При разработке плана обеспечения непрерывности бизнеса необходимо проверить максимально допустимое время, в течение которого приложение полностью восстановится после сбоя.

Для учетных записей *Azure Cosmos*, настроенных с несколькими регионами записи, конфликты обновлений могут возникать, тогда ко-

гда пользователи одновременно обновляют один и тот же элемент в нескольких регионах. Конфликты обновлений могут быть трех типов:

1) конфликты вставки: эти конфликты могут возникать, когда приложение одновременно вставляет два или более элементов с одним и тем же уникальным индексом в двух или более регионах;

2) конфликты замены: эти конфликты могут возникать, когда приложение обновляет один и тот же элемент одновременно в двух или более регионах;

3) конфликты удаления: эти конфликты могут возникать, когда приложение одновременно удаляет элемент в одном регионе и обновляет его в другом.

Azure Cosmos DB предлагает гибкий механизм на основе политик для разрешения конфликтов записи. В контейнере *Azure Cosmos* можно выбрать одну из двух политик разрешения конфликтов:

1) приоритет последней записи (*LWW*): эта политика разрешения по умолчанию использует свойство отметки времени, определенное системой. Она основана на протоколе синхронизации времени. Если вы используете *SQL API*, вы можете указать любое другое настраиваемое числовое свойство, которое будет использоваться для разрешения конфликтов. Настраиваемое числовое свойство также называется путем разрешения конфликта. Если два или более элемента конфликтуют при операциях вставки или замены, элемент с наибольшим значением для пути разрешения конфликта становится приоритетным. Система определяет победителя, если несколько элементов имеют одинаковое числовое значение для пути разрешения конфликтов. Этот результат происходит независимо от значения пути разрешения конфликта;

2) настраиваемая политика: эта политика разрешения предназначена для определяемой приложением семантики для разрешения конфликтов. Когда устанавливается данная политика в своем контейнере *Azure Cosmos*, необходимо зарегистрировать хранимую процедуру слияния. Эта процедура вызывается автоматически при обнаружении конфликтов в транзакции базы данных на сервере. Система предоставляет гарантию выполнения процедуры слияния как части протокола фиксации ровно один раз.

API*-интерфейсы *Azure Cosmos DB

Azure Cosmos DB предлагает несколько *API*-интерфейсов баз данных, включая *Core* (*SQL*) *API*, *API* для *MongoDB*, *Cassandra API*, *Gremlin API* и *Table API*. С помощью данных *API* можно моделировать данные реального мира с помощью документов, моделей данных

типа «ключ–значение», графиков и семейств столбцов. *API*-интерфейсы позволяют вашим приложениям обрабатывать *Azure Cosmos DB* так, как если бы это были различные другие технологии баз данных, без накладных расходов на управление и подходы к масштабированию. Применяя эти *API*, *Azure Cosmos DB* может использовать экосистемы, инструменты и навыки, которые у вас уже есть для моделирования данных и запросов.

Различные *API*-интерфейсы предлагают автоматическое масштабирование хранилища и гарантии, гибкости и производительности. Имеется возможность выбора *API* для создания своего приложения в зависимости от вашей рабочей нагрузки и требований проекта.

Core (*SQL*) *API* встроен в *Azure Cosmos DB*. *API* для *MongoDB*, *Cassandra, Gremlin* и *Table* реализуют сетевой протокол движков баз данных с открытым исходным кодом. Эти *API* лучше всего подходят, если выполняются следующие условия:

- наличие в проекте приложения *MongoDB*, *Cassandra* или *Gremlin*;
- нежелание перезаписывать весь уровень доступа к данным;
- использование экосистемы разработчиков с открытым исходным кодом, клиентские драйверы, опыт и ресурсы для своей базы данных;
- использование ключевых функций *Azure Cosmos DB*, такие как глобальное распространение, эластичное масштабирование хранилища и пропускной способности, производительность, низкую задержку, возможность выполнять транзакционные и аналитические рабочие нагрузки, а также применение полностью управляемой платформы;
- разработка модернизированного приложения в мультиоблачной среде.

Возможно создавать новые приложения с указанными *API* или переносить существующие данные. Чтобы запустить перенесенные приложения, измените строку подключения вашего приложения и продолжайте работать, как прежде. При переносе существующих приложений обязательно оцените поддержку функций применяемых *API*.

В зависимости от вашей рабочей нагрузки вы должны выбрать *API*, который соответствует вашим требованиям. На рис. 7.11 [10] показана блок-схема выбора подходящего *API* при создании новых приложений или миграции существующих приложений в *Azure Cosmos DB*.

Core (*SQL*) *API* хранит данные в формате документа. Он предла-

гает лучший сквозной опыт, поскольку есть возможность полностью контролировать интерфейс, сервис и клиентские библиотеки *SDK*. Любая новая функция, развернутая в *Azure Cosmos DB*, сначала доступна в учетных записях *SQL API*. Учетные записи *API SQL Azure Cosmos DB* обеспечивают поддержку запросов к элементам с использованием синтаксиса языка структурированных запросов (*SQL*) – одного из самых знакомых и популярных языков запросов для запросов к объектам *JSON*.

Если вы выполняете миграцию из других баз данных, таких как *Oracle, DynamoDB, HBase* и хотите использовать модернизированные технологии для создания своих приложений, рекомендуется использовать *SQL API. SQL API* поддерживает аналитику и обеспечивает изоляцию производительности между операционными и аналитическими рабочими нагрузками.

API for MongoDB хранит данные в структуре документа в формате *BSON*. Он совместим с проводным протоколом *MongoDB*, однако он не использует никакого собственного кода, связанного с *MongoDB. API for MongoDB* – отличный выбор, если вы хотите использовать более широкую экосистему и навыки *MongoDB*, не жертвуя такими функциями *Azure Cosmos DB*, как масштабирование, высокая доступность, георепликация, несколько мест записи, автоматическое и прозрачное управление сегментом, прозрачная репликация между операционными и аналитическими магазинами и другое.

Можно использовать существующие приложения *MongoDB* с *API* для *MongoDB*, просто изменив строку подключения. Можно также переместить любые существующие данные с помощью собственных инструментов *MongoDB*, таких как *mongodump* и *mongorestore*, или с помощью инструмента миграции базы данных *Azure*. Такие инструменты, как оболочка *MongoDB, MongoDB Compass* и *Robo3T*, могут выполнять запросы и работать с данными как с родным *MongoDB*.

Для выполнения планирования емкости для миграции на *API Azure Cosmos DB* для *MongoDB* из существующего кластера базы данных можно использовать информацию о нем.

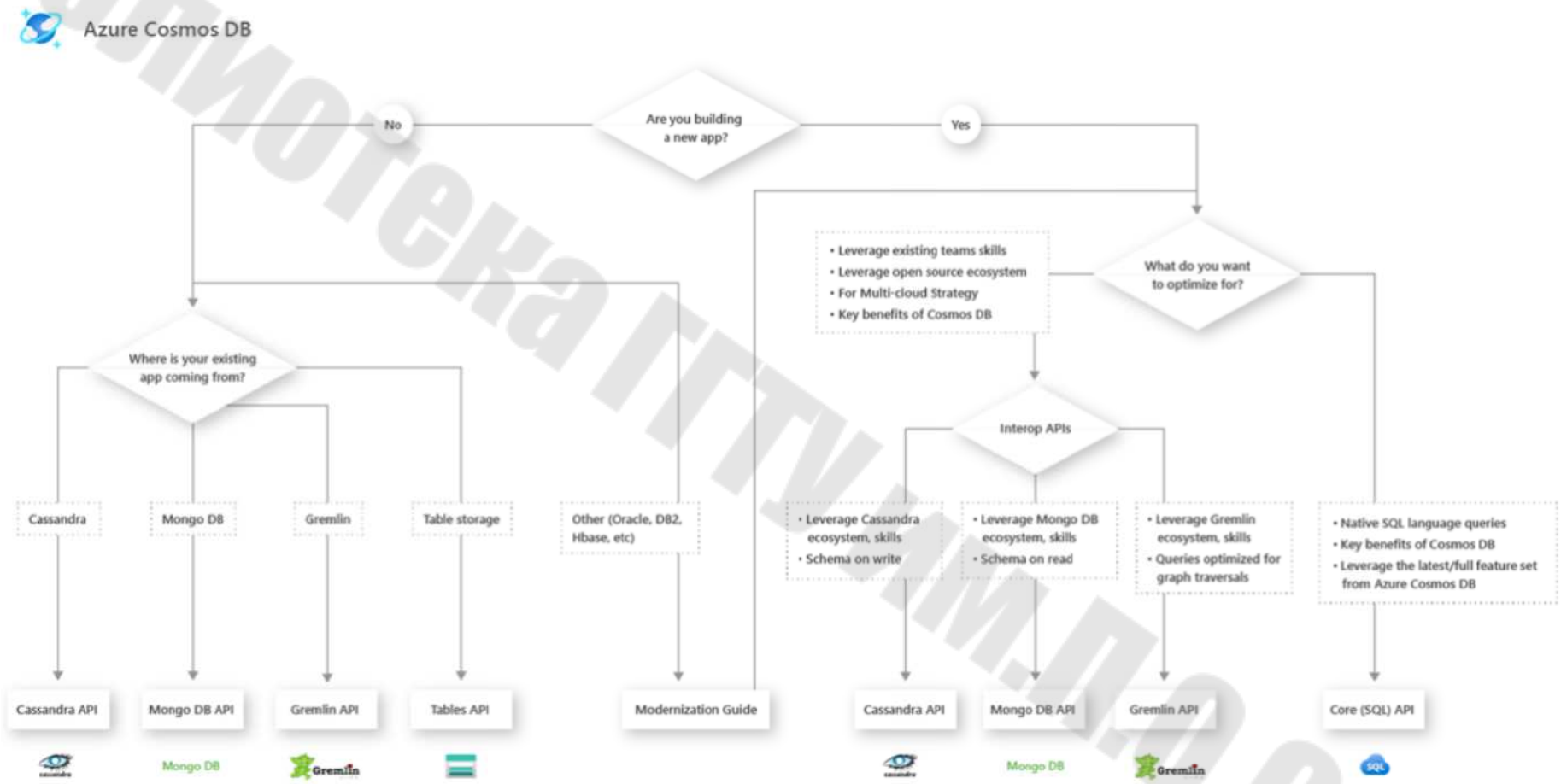

Рис. 7.11. Блок-схема выбора, подходящего *ARI* при создании новых приложений

Зная количество виртуальных ядер и серверов в существующем кластере базы данных, вы можете оценить количество запросов с использованием виртуальных ядер или виртуальных ЦП.

Знание частот запросов для текущей рабочей нагрузки базы данных позволяет оценить объем запросов с помощью планировщика емкости *Azure Cosmos DB*.

Cassandra API хранит данные в схеме, ориентированной на столбцы. *Apache Cassandra* предлагает высоко распределенный подход с горизонтальным масштабированием для хранения больших объемов данных, обеспечивая гибкий подход к схеме, ориентированной на столбцы. *Cassandra API* в *Azure Cosmos DB* соответствует этой философии подхода к распределенным базам данных *NoSQL*. *Cassandra API* – это сетевой протокол, совместимый с *Apache Cassandra*. Следует выбрать *Cassandra API*, если есть возможность воспользоваться преимуществами эластичности и управляемой природы *Azure Cosmos DB* и возможность по-прежнему использовать большинство встроенных функций, инструментов и экосистемы *Apache Cassandra*. Это означает, что в *Cassandra API* вам не нужно управлять ОС, виртуальной машиной *Java*, сборщиком мусора, производительностью чтения-записи, узлами, кластерами и т. д.

Возможно применение клиентских драйверов *Apache Cassandra* для подключения к *Cassandra API. Cassandra API* позволяет взаимодействовать с данными с помощью языка запросов *Cassandra* (*CQL*) и таких инструментов, как оболочка *CQL*, клиентские драйверы Cassandra, с которыми вы уже знакомы. *Cassandra API* в настоящее время поддерживает только сценарии *OLTP*. Применяя *Cassandra API*, вы можете использовать уникальные функции *Azure Cosmos DB*, например, канал изменений.

Gremlin API позволяет пользователям выполнять запросы к графам и хранить данные в виде ребер и вершин. Используйте этот *API* для сценариев, включающих динамические данные, данные со сложными отношениями, данные, которые слишком сложны для моделирования с помощью реляционных баз данных, а также если вы хотите использовать существующую экосистему и навыки *Gremlin*. *API*-интерфейс *Gremlin* в *Azure Cosmos DB* сочетает мощь алгоритмов графовых баз данных с хорошо масштабируемой управляемой инфраструктурой. Он обеспечивает уникальное гибкое решение наиболее распространенных проблем с данными, связанных с отсутствием гибкости и реляционных подходов. *Gremlin API* в настоящее время поддерживает только сценарии *OLTP*.

API-интерфейс *Gremlin* в *Azure Cosmos DB* основан на графической вычислительной платформе *Apache TinkerPop*. *Gremlin API* использует один и тот же язык запросов *Graph* для приема и запроса данных. Он использует стратегию разделения *Azure Cosmos DB* для выполнения операций чтения и записи из ядра базы данных *Graph. Gremlin API* поддерживает сетевой протокол с *Gremlin* с открытым исходным кодом, поэтому вы можете использовать пакеты *Gremlin SDK* с открытым исходным кодом для создания своего приложения. *Azure Cosmos DB Gremlin API* также работает с *Apache Spark* и *GraphFrames* для сложных аналитических сценариев графов.

Таблица *API* хранит данные в формате «ключ–значение». При использовании хранилища таблиц *Azure* возможны некоторые ограничения в задержке, масштабировании, пропускной способности, глобальном распределении, управлении индексами, низкой производительности запросов. *Table API* преодолевает эти ограничения; рекомендуется перенести приложение, если вы хотите использовать преимущества *Azure Cosmos DB*. *API* таблиц поддерживает только сценарии *OLTP*. Приложения, написанные для хранилища таблиц *Azure*, могут перейти на *API* таблиц с небольшими изменениями кода и воспользоваться преимуществами дополнительных возможностей.

Для переноса существующего веб-приложения *MongoDB Node.js* необходимо создать учетную запись *API Azure Cosmos DB* для *Mongo DB* с помощью *Azure Cloud Shell* и приложения *MEAN* (*MongoDB*, *Express, Angular и Node.js*), клонированного из *GitHub*. *Azure Cosmos DB* – это многомодельная служба базы данных, которая позволяет быстро создавать и запрашивать базы данных документов, таблиц, ключей и значений, графов с возможностями глобального распределения и горизонтального масштабирования.

Node.js – программная платформа, основанная на движке *V*8 (транслирующем *JavaScript* в машинный код), превращающая *JavaScript* из узкоспециализированного языка в язык общего назначения. *Node.js* добавляет возможность *JavaScript* взаимодействовать с устройствами ввода-вывода через свой *API*, написанный на *C++*, подключать другие внешние библиотеки, написанные на разных языках, обеспечивая вызовы к ним из *JavaScript*-кода. *Node.js* применяется преимущественно на сервере, выполняя роль веб-сервера, также есть возможность разрабатывать на *Node.js* и десктопные оконные приложения (при помощи *NW.js*, *AppJS* или *Electron* для *Linux*, *Windows* и *macOS*) и даже программировать микроконтроллеры (например, *tessel*, *low.js* и *espruino*). В основе *Node.js* лежит событийно-

ориентированное и асинхронное (или реактивное) программирование с неблокирующим вводом-выводом.

Git – система управления версиями с распределенной архитектурой. В отличие от некогда популярных систем вроде *CVS* и *Subversion* (*SVN*), где полная история версий проекта доступна лишь в одном месте, в *Git* каждая рабочая копия кода сама по себе является репозиторием. Это позволяет всем разработчикам хранить историю изменений в полном объеме.

Используйте среду *Bash* в *Azure Cloud Shell*. Запустите *Cloud Shell* в новом окне. Установите *Azure CLI* для выполнения справочных команд *CLI*. При использовании локальной установки войдите в *Azure CLI* с помощью команды *az login*. Для завершения процесса аутентификации следует использовать инструкции, отображаемые на терминале. Запустите команду *az version*, чтобы найти установленную версию и зависимые библиотеки. Чтобы выполнить обновление до последней версии, запустите команду *az upgrade*.

Для клонирования образца приложения из репозитория следует выполнить некоторые команды. Пример репозитория содержит приложение *MEAN.js* по умолчанию. Откройте командную строку, создайте новую папку с именем *git-samples*, затем закройте командную строку:

mkdir "C: \ git-samples"

Откройте окно терминала *git*, например, *git bash*, используйте команду *cd*, чтобы перейти в новую папку для установки примера приложения:

cd "C: \ git-samples"

Выполните следующую команду, чтобы клонировать образец репозитория:

git clone https://github.com/prashanthmadi/mean

Указанное приложение *MongoDB*, написанное на *Node.js,* подключается к вашей базе данных *Azure Cosmos DB*, которая поддерживает клиент *MongoDB*.

Установите необходимые пакеты и запустите приложение:

cd mean

npm install
npm start

Приложение попытается подключиться к источнику *MongoDB* и завершится ошибкой; продолжайте и выходите из приложения, когда на выходе будет возвращено *MongoError*: *connect ECONNREFUSED* 127.0.0.1:27017.

Если вы используете установленный интерфейс командной строки *Azure*, войдите в *Azure* с помощью команды *az login* и следуйте инструкциям на экране. Можно пропустить этот шаг, если используется *Azure Cloud Shell*:

az login

При использовании установленного интерфейса командной строки *Azure* проверьте, установлен ли компонент *cosmosdb*, выполнив команду *az*. Если *cosmosdb* находится в списке базовых команд, перейдите к следующей команде. Можно пропустить этот шаг, если вы используете *Azure Cloud Shell*. Если *cosmosdb* отсутствует в списке базовых команд, переустановите *Azure CLI*.

Для создания группы ресурсов используется команда *az group create*. Группа ресурсов *Azure* – это логический контейнер, в котором развертываются и управляются такие ресурсы *Azure*, как веб-приложения, базы данных и учетные записи хранения.

В следующем примере создается группа ресурсов в регионе Западная Европа. Выберите уникальное имя для группы ресурсов.

При использовании *Azure Cloud Shell*, выберите «Попробовать», следуйте инструкциям на экране для входа в систему, затем скопируйте команду в командную строку:

az group create --name myResourceGroup --location "West Europe"

Создайте учетную запись *Cosmos* с помощью команды *az cosmosdb create*. В следующей команде замените на свое собственное уникальное имя учетной записи *Cosmos* в заполнителе *<cosmosdb-name>*. Это уникальное имя будет использоваться как часть вашей конечной точки *Cosmos DB* (*https: // <cosmosdb-name> .documents. azure.com* /), поэтому имя должно быть уникальным для всех учетных записей *Cosmos* в *Azure*:

az cosmosdb create --name <cosmosdb-name> --resource-group myResourceGroup --kind MongoDB

При создании учетной записи *Azure Cosmos DB* в интерфейсе командной строки *Azure* отображается информация, аналогичная приведенной в следующем примере:

```
{
  "databaseAccountOfferType": "Standard",
  "documentEndpoint": "https://<cosmosdb-name>.documents.azure.com:443/",
  "id": "/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/myResourceGroup/providers/Microsoft.DocumentDB/databaseAccounts/<cosmosdb-name>",
  "kind": "MongoDB",
  "location": "West Europe",
  "name": "<cosmosdb-name>",
  "readLocations": [
    {
      "documentEndpoint":"https://<cosmosdb-name>-westeurope.documents.azure.com:443/",
      "failoverPriority": 0,
      "id": "<cosmosdb-name>-westeurope",
      "locationName": "West Europe",
      "provisioningState": "Succeeded"
    }
  ],
  "resourceGroup": "myResourceGroup",
  "type": "Microsoft.DocumentDB/databaseAccounts",
  "writeLocations": [
    {
      "documentEndpoint":"https://<cosmosdb-name>-westeurope.documents.azure.com:443/",
      "failoverPriority": 0,
      "id": "<cosmosdb-name>-westeurope",
      "locationName": "West Europe",
      "provisioningState": "Succeeded"
    }
  ]
}
```

Для настройки строки подключения в приложении *Node.js* в репозитории *MEAN.js* откройте *config / env / local-development.js*. Замените содержимое этого файла следующим кодом. Не забудьте также заменить два заполнителя *<cosmosdb-name>* на имя вашей учетной записи *Cosmos*:

```
'use strict';

module.exports = {
  db: {
    uri:'mongodb://<cosmosdb-
name>:<primary_master_key>@<cosmosdb-
name>.documents.azure.com:10255/mean-
dev?ssl=true&sslverifycertificate=false'
  }
};
```

Чтобы подключиться к базе данных *Cosmos*, вы должны получить ключ базы данных. Используйте команду *az cosmosdb keys list*, чтобы получить первичный ключ:

az cosmosdb keys list --name <cosmosdb-name> --resource-group myResourceGroup --query "primaryMasterKey"

Скопируйте значение *primaryMasterKey*. Вставьте это поверх *<primary_ma-ster_key> в local-development.js*. Сохраните изменения.

Снова запустите приложение:

npm start

Теперь консольное сообщение должно объявить вам, что среда разработки запущена и работает. Перейдите по адресу *http: // localhost:* 3000 в браузере. Выберите «Зарегистрироваться» в верхнем меню и создайте двух фиктивных пользователей. Пример приложения *MEAN.js* хранит пользовательские данные в базе данных. Если все прошло успешно и *MEAN.js* автоматически входит в систему для созданного пользователя, ваше соединение с *Azure Cosmos DB* работает (рис. 7.12).

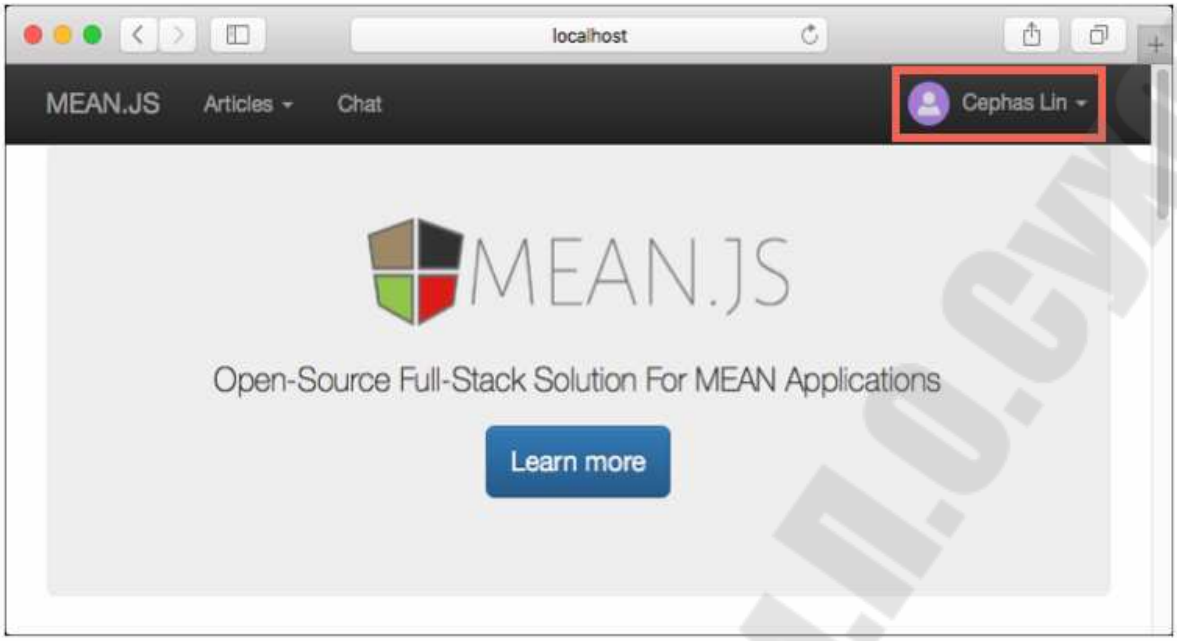

Рис. 7.12. Окно программы

Данные, хранящиеся в базе данных *Cosmos*, доступны для просмотра и запросов на портале *Azure*. Чтобы просматривать, запрашивать и работать с пользовательскими данными, созданными на предыдущем шаге, войдите на портал *Azure* в своем веб-браузере.

В верхнем поле поиска введите *Azure Cosmos DB*. Когда откроется колонка вашей учетной записи *Cosmos*, выберите свою учетную запись *Cosmos*. На левой панели навигации выберите *Data Explorer*. Разверните свою коллекцию на панели «Коллекции», и тогда вы сможете просматривать документы в коллекции, запрашивать данные и даже создавать и запускать хранимые процедуры, триггеры и пользовательские функции (рис. 7.13).

На этом этапе вы развертываете приложение *Node.js* в *Cosmos DB*.

Вы могли заметить, что ранее измененный файл конфигурации предназначен для среды разработки (/*config/env/local-development.js*). Когда вы развертываете свое приложение в службе приложений, оно по умолчанию запускается в производственной среде. Теперь вам нужно внести те же изменения в соответствующий файл конфигурации.

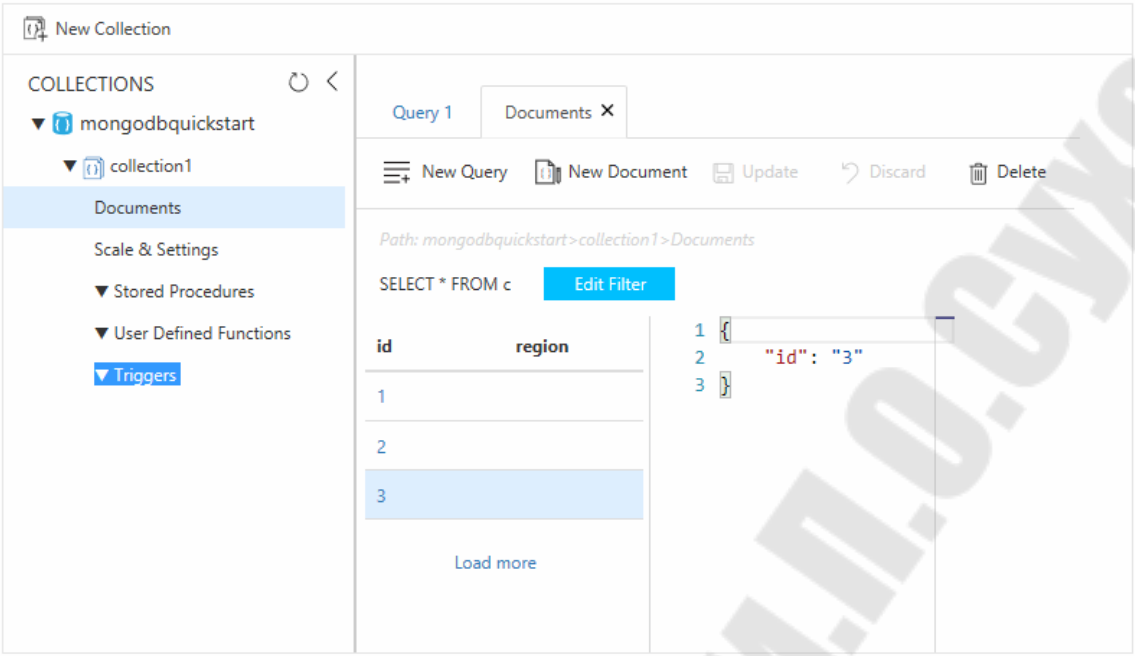

Рис. 7.13. Окно проводника

В вашем репозитории *MEAN.js* откройте *config/env/production.js*. В объекте *db* замените значение *uri*, как показано в следующем примере. Обязательно замените заполнители, как и раньше:

'mongodb://<cosmosdb-name>:<primary_master_key>@<cosmosdb-name>.documents.azure.com:10255/mean?ssl=true&sslverifycertificate=false'

В терминале зафиксируйте все свои изменения в *Git*. Можно скопировать обе команды, чтобы запускать их вместе:

```
git add .
git commit -m "configured MongoDB connection string"
```

Когда закончите работу с приложением и учетной записью *Azure Cosmos DB*, вы можете удалить созданные ресурсы *Azure*, чтобы не взималась дополнительная плата. Чтобы удалить ресурсы, выполните следующее (рис. 7.14):

1) на панели поиска портала *Azure* найдите и выберите группы ресурсов;

2) в списке выберите группу ресурсов, которую вы создали для этого краткого руководства;

3) на странице «Обзор группы ресурсов» выберите «Удалить группу ресурсов» (рис. 7.15);

4) в следующем окне введите имя удаляемой группы ресурсов, а затем выберите «Удалить».

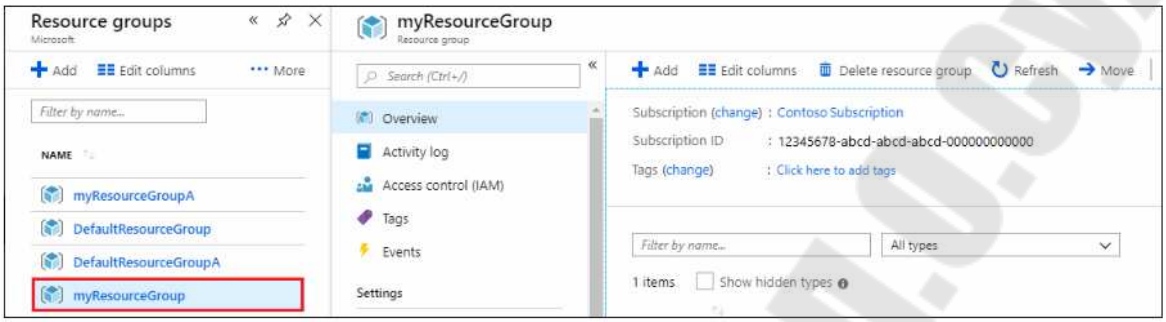

Рис. 7.14. Окно проводника

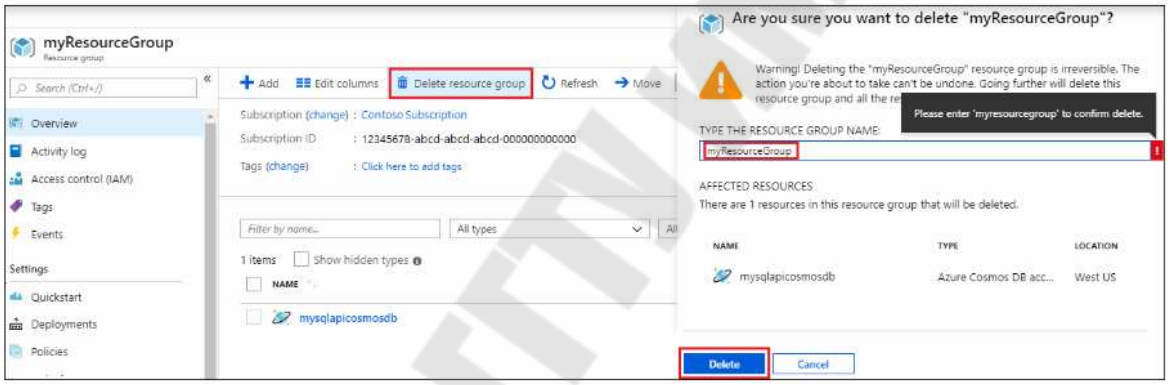

Рис. 7.15. Окно удаления группы ресурсов

ЛИТЕРАТУРА

1. Базы данных NoSQL/ Microsoft Azure. – 2021. – Режим доступа: https://azure.microsoft.com/ ru-ru/overview/-nosql-database/.

2. Введение в Node JS. – 2021. – Режим доступа: https://metanit.-com/web/nodejs/1.1.php.

3. Документация по Azure Cosmos DB. – 2021. – Режим доступа: https://docs.micro-soft.com/ru-ru/azure/cosmos-db/.

4. Драйвер MongoDB .NET. – 2021. – Режим доступа: https://mongodb.git-hub.io/mongo-csharp-driver/.

5. Онлайн-руководство по MongoDB. – 2021. –Режим доступа: https://metanit.com/ nosql/mongodb/.

6. Работа с данными NoSQL в Azure Cosmos DB. – 2021. – Режим доступа: https://docs.micro-soft.com/ru-ru/learn/paths/work-with-nosql-data-in-azure-cosmos-db/.

7. Руководство по NoSQL с Azure Cosmos DB. – 2021. – Режим доступа: https://github.com/ PacktPublishing/Guide-to-NoSQL-with-Azure-Cosmos-DB.

8. Брэдшоу, Ш., MongoDB: полное руководство. Мощная и масштабируемая система управления базами данных / Ш. Брэдшоу, Й. Брэзил, К. Ходоров; пер. с англ. Д. А. Беликова. – М. : ДМК Пресс, 2020. – 540 с.

9. MongoDB 5.0 Руководство. – 2021. – Режим доступа: https:// docs.mongodb.com/ manual/.

10. MongoDB Azure. – 2021. – Режим доступа: https://ru. wikipedia.org/wiki/ MongoDB.

11. MongoDB C#/.NET Driver. – 2021. – Режим доступа: https:// docs.mongo-db.com/drivers/csharp/.

12. MongoDB Compass. – 2021. – Режим доступа: https:// docs.mongodb.com/compass/ current/.

13. Tutorial MongoDB/ MongoDB. – 2021. – Режим доступа: https:// www.tutorialspoint.com/mongodb/index. htm.

Приложения

Приложение 1

Таблица П.1.1

Сравнение хранилищ данных: документов, столбцов, пар «ключ–значение», графов

Требование	Хранилище данных документов	Столбчатое хранилище данных	Хранилище данных пар «ключ–значение»	Хранилище данных графов
Нормализация	Денормализированные данные	Денормализированные данные	Денормализированные данные	Нормализированные данные
Схема	Схема при чтении	Семейства столбцов, определенные при записи, схема столбца при чтении	Схема при чтении	Схема при чтении
Согласованность (между параллельными транзакциями)	Настраиваемый уровень согласованности, гарантии на уровне документа	Гарантии на уровне семейства столбцов	Гарантии на уровне ключей	Гарантии на уровне графа
Атомарность (область транзакции)	Коллекция	Таблица	Таблица	Граф
Стратегия блокировки	Оптимистичная (без блокировки)	Пессимистичная (блокировка строк)	Оптимистичная	–
Шаблон доступа	Прямой доступ	Статистические выражения на основе данных большого формата	Прямой доступ	Прямой доступ
Индексация	Первичный и вторичные индексы	Первичный и вторичные индексы	Только первичный индекс	Первичный и вторичные индексы
Форма представления данных	Документ	Таблица с семействами столбцов	Ключ и значение	Граф с ребрами и вершинами
Разреженные	Да	Да	Да	Нет
Масштабность (большое количество столбцов и атрибутов)	Да	Да	Нет	Нет

Окончание табл. П.1.1

Требование	**Хранилище данных документов**	**Столбчатое хранилище данных**	**Хранилище данных пар «ключ–значение»**	**Хранилище данных графов**
Размер данных	От малого (КБ) до среднего (несколько МБ)	От среднего (МБ) до большого (несколько ГБ)	Небольшой (КБ)	Небольшой (КБ)
Общий максимальный масштаб	Очень большой (ПБ)	Очень большой (ПБ)	Очень большой (ПБ)	Большой (ТБ)

Таблица П.1.2

Сравнение хранилищ данных: временных рядов, объектов, внешних индексов

Требование	**Данные временных рядов**	**Хранилище данных объектов**	**Хранилище данных внешних индексов**
Нормализация	Нормализированные данные	Денормализированные данные	Денормализированные данные
Схема	Схема при чтении	Схема при чтении	Схема при записи
Согласованность (между параллельными транзакциями)	Недоступно	Недоступно	Недоступно
Атомарность (область транзакции)	Недоступно	Объект	Недоступно
Стратегия блокировки	Недоступно	Пессимистичная (блокировка больших двоичных объектов)	Недоступно
Шаблон доступа	Прямой доступ и агрегирование	Последовательный доступ	Прямой доступ
Индексация	Первичный и вторичные индексы	Только первичный индекс	Недоступно
Форма представления данных	Таблица	Большой двоичный объект и метаданные	Документ
Разреженные;	Нет	Недоступно	Нет

Окончание табл. П.1.2

Требование	**Данные временных рядов**	**Хранилище данных объектов**	**Хранилище данных внешних индексов**
Масштабность (большое количество столбцов и атрибутов)	Нет	Да	Да
Размер данных	Небольшой (КБ)	От большого (ГБ) до очень большого (ТБ)	Небольшой (КБ)
Общий максимальный масштаб	Большой (несколько ТБ)	Очень большой (ПБ)	Большой (несколько ТБ)

Приложение 2

Программный код файла *app.js*

const express = require ("экспресс");

const MongoClient = требуется ("mongodb"). MongoClient;

const objectId = require ("mongodb"). ObjectId;

**

константное приложение = экспресс ();

const jsonParser = express.json ();

**

const mongoClient = новый MongoClient ("mongodb: // localhost: 27017 /");

**

let dbClient;

**

app.use (express.static (__ dirname + "/ public"));

**

mongoClient.connect (функция (ошибка, клиент) {

если (ошибка) вернуть console.log (ошибка);

dbClient = клиент;

app.locals.collection = client.db ("usersdb"). collection ("users");

app.listen (3000, function () {

console.log ("Сервер ожидает подключения ...");

});

*});

**

app.get ("/ api / users", function (req, res) {

**

const collection = req.app.locals.collection;

collection.find ({}). toArray (function (err, users) {

**

если (ошибка) вернуть console.log (ошибка);

res.send (пользователи)

*});

**

});

app.get ("/ api / users /: id", function (req, res) {

<font

style="vertical-align: inherit;">

const id = новый идентификатор объекта (req.params.id);

const collection = req.app.locals.collection;

collection.findOne ({_ id: id}, function (err, user) {

**

если (ошибка) вернуть console.log (ошибка);

res.send (пользователь);

});

});

**

app.post ("/ api / users", jsonParser, function (req, res) {

**

если (! req.body) return res.sendStatus (400);

**

const userName = req.body.name;

const userAge = req.body.age;

const user = {имя: имя пользователя, возраст: возраст пользователя};

**

const collection = req.app.locals.collection;

collection.insertOne (пользователь, функция (ошибка, результат) {

**

если (ошибка) вернуть console.log (ошибка);

res.send (пользователь);

});

});

**

app.delete ("/ api / users /: id", function (req, res) {

**

const id = новый идентификатор объекта (req.params.id);

const collection = req.app.locals.collection;

collection.findOneAndDelete ({_ id: id}, function (err, result) {

**

*если (ошибка) вернуть console.log (ошибка); *

пусть user = result.value;

res.send (пользователь);

});<font style="vertical-align: in-

```
herit;"><font style="vertical-align: inherit;">
      });</font></font><font></font>
        <font></font><font style="vertical-align: inherit;"><font
style="vertical-align: inherit;">
      app.put ("/ api / users", jsonParser, function (req, res)
{</font></font><font></font>
          <font></font><font style="vertical-align: inherit;"><font
style="vertical-align: inherit;">
      если (! req.body) return res.sendStatus
(400);</font></font><font></font><font style="vertical-align: in-
herit;"><font style="vertical-align: inherit;">
      const id = новый идентификатор объекта
(req.body.id);</font></font><font></font><font style="vertical-align:
inherit;"><font style="vertical-align: inherit;">
      const userName =
req.body.name;</font></font><font></font><font style="vertical-align:
inherit;"><font style="vertical-align: inherit;">
      const userAge = req.body.age;</font></font><font></font>
        <font></font><font style="vertical-align: inherit;"><font
style="vertical-align: inherit;">
      const collection =
req.app.locals.collection;</font></font><font></font><font
style="vertical-align: inherit;"><font style="vertical-align: inherit;">
      collection.findOneAndUpdate ({_ id: id}, {$ set: {age: userAge,
name: userName}},</font></font><font></font><font style="vertical-
align: inherit;"><font style="vertical-align: inherit;">
          {returnDocument: "после"}, функция (ошибка, результат)
{</font></font><font></font>
            <font></font><font style="vertical-align: inherit;"><font
style="vertical-align: inherit;">
      если (ошибка) вернуть console.log (ошибка);
</font></font><font></font><font style="vertical-align: inherit;"><font
style="vertical-align: inherit;">
      const user = result.value;</font></font><font></font><font
style="vertical-align: inherit;"><font style="vertical-align: inherit;">
      res.send (пользователь);</font></font><font></font><font
style="vertical-align: inherit;"><font style="vertical-align: inherit;">
      });</font></font><font></font><font style="vertical-align: in-
herit;"><font style="vertical-align: inherit;">
```

```
});</font></font><font></font>
<font></font><font style="vertical-align: inherit;"><font
style="vertical-align: inherit;">
// прослушиваем прерывание работы программы (ctrl-
c)</font></font><font></font><font style="vertical-align: in-
herit;"><font style="vertical-align: inherit;">
process.on ("СИГНАЛ", () => {</font></font><font></font><font
style="vertical-align: inherit;"><font style="vertical-align: inherit;">
dbClient.close ();</font></font><font></font><font
style="vertical-align: inherit;"><font style="vertical-align: inherit;">
process.exit ();</font></font><font></font><font
style="vertical-align: inherit;"><font style="vertical-align: inherit;">
});</font></font><font></font>
```

Приложение 3

Программный код файла *index.html*

```
<!DOCTYPE html><font></font>
<html><font></font>
<head><font></font>
  <meta charset="utf-8" /><font></font>
  <meta name="viewport" content="width=device-width" /><font></font>
  <title>Список пользователей</title><font></font>
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" rel="stylesheet" /><font></font>
</head><font></font>
<body><font></font>
  <h2>Список пользователей</h2><font></font>
  <form name="userForm"><font></font>
    <input type="hidden" name="id" value="0" /><font></font>
    <div class="form-group"><font></font>
      <label for="name">Имя:</label><font></font>
      <input class="form-control" name="name" /><font></font>
    </div><font></font>
    <div class="form-group"><font></font>
      <label for="age">Возраст:</label><font></font>
      <input class="form-control" name="age" /><font></font>
    </div><font></font>
    <div class="panel-body"><font></font>
      <button type="submit" class="btn btn-sm btn-primary">Сохранить</button><font></font>
      <a id="reset" class="btn btn-sm btn-primary">Сбросить</a><font></font>
    </div><font></font>
  </form><font></font>
  <table class="table table-condensed table-striped table-bordered"><font></font>
    <thead><tr><th>Id</th><th>Имя</th><th>возраст</th>
```

```
<th></th></tr></thead><font></font>
            <tbody><font></font>
            </tbody><font></font>
          </table><font></font>
        <font></font>
          <script><font></font>
          // Получение всех пользователей<font></font>
            async function GetUsers() {<font></font>
              // отправляет запрос и получаем ответ<font></font>
              const response = await fetch("/api/users", {<font></font>
                method: "GET",<font></font>
                headers: { "Accept": "application/json" }<font></font>
              });<font></font>
              // если запрос прошел нормально<font></font>
              if (response.ok === true) {<font></font>
                // получаем данные<font></font>
                const users = await response.json();<font></font>
                let rows = document.querySelector("tbody");
<font></font>
                users.forEach(user => {<font></font>
                  // добавляем полученные элементы в табли-
цу<font></font>
                  rows.append(row(user));<font></font>
                });<font></font>
              }<font></font>
            }<font></font>
            // Получение одного пользователя<font></font>
            async function GetUser(id) {<font></font>
              const response = await fetch("/api/users/" + id,
{<font></font>
                method: "GET",<font></font>
                headers: { "Accept": "application/json" }<font></font>
              });<font></font>
              if (response.ok === true) {<font></font>
                const user = await response.json();<font></font>
                const form = document.forms["userForm"];<font></font>
                form.elements["id"].value = user._id;<font></font>
                form.elements["name"].value = user.name;<font></font>
                form.elements["age"].value = user.age;<font></font>
```

```
        }<font></font>
    }<font></font>
    // Добавление пользователя<font></font>
    async function CreateUser(userName, userAge) {<font></font>
<font></font>
        const response = await fetch("api/users", {<font></font>
            method: "POST",<font></font>
            headers: { "Accept": "application/json", "Content-Type": "application/json" },<font></font>
            body: JSON.stringify({<font></font>
                name: userName,<font></font>
                age: parseInt(userAge, 10)<font></font>
            })<font></font>
        });<font></font>
        if (response.ok === true) {<font></font>
            const user = await response.json();<font></font>
            reset();<font></font>
            docu-
ment.querySelector("tbody").append(row(user));<font></font>
        }<font></font>
    }<font></font>
    // Изменение пользователя<font></font>
    async function EditUser(userId, userName, userAge) {<font></font>
        const response = await fetch("api/users", {<font></font>
            method: "PUT",<font></font>
            headers: { "Accept": "application/json", "Content-Type": "application/json" },<font></font>
            body: JSON.stringify({<font></font>
                id: userId,<font></font>
                name: userName,<font></font>
                age: parseInt(userAge, 10)<font></font>
            })<font></font>
        });<font></font>
        if (response.ok === true) {<font></font>
            const user = await response.json();<font></font>
            reset();<font></font>
            document.querySelector("tr[data-rowid='" + user._id +
```

```
"]").replaceWith(row(user));<font></font>
            }<font></font>
        }<font></font>
        // Удаление пользователя<font></font>
        async function DeleteUser(id) {<font></font>
            const response = await fetch("/api/users/" + id, {<font></font>
                method: "DELETE",<font></font>
                headers: { "Accept": "application/json" }<font></font>
            });<font></font>
            if (response.ok === true) {<font></font>
                const user = await response.json();<font></font>
                document.querySelector("tr[data-rowid='" + user._id + "']").remove();<font></font>
            }<font></font>
        }<font></font>
    <font></font>
        // сброс формы<font></font>
        function reset() {<font></font>
            const form = document.forms["userForm"];<font></font>
            form.reset();<font></font>
            form.elements["id"].value = 0;<font></font>
        }<font></font>
        // создание строки для таблицы<font></font>
        function row(user) {<font></font>
    <font></font>
            const tr = document.createElement("tr");<font></font>
            tr.setAttribute("data-rowid", user._id);<font></font>
    <font></font>
            const idTd = document.createElement("td");<font></font>
            idTd.append(user._id);<font></font>
            tr.append(idTd);<font></font>
    <font></font>
            const nameTd = document.createElement("td");<font></font>
            nameTd.append(user.name);<font></font>
            tr.append(nameTd);<font></font>
    <font></font>
            const ageTd = document.createElement("td");<font></font>
```

```
            ageTd.append(user.age);<font></font>
            tr.append(ageTd);<font></font>
             <font></font>
            const linksTd = document.createElement("td");<font>
</font>
        <font></font>
            const editLink = document.createElement("a");<font>
</font>
            editLink.setAttribute("data-id", user._id);<font></font>
            editLink.setAttribute("style","cursor:pointer;padding:
15px;");<font></font>
            editLink.append("Изменить");<font></font>
            editLink.addEventListener("click", e => {<font></font>
        <font></font>
                e.preventDefault();<font></font>
                GetUser(user._id);<font></font>
            });<font></font>
            linksTd.append(editLink);<font></font>
        <font></font>
            const removeLink = document.createElement("a");<font>
</font>
            removeLink.setAttribute("data-id", user._id);<font></font>
            removeLink.setAttribute("style", "cur-
sor:pointer;padding:15px;");<font></font>
            removeLink.append("Удалить");<font></font>
            removeLink.addEventListener("click", e => {<font></font>
        <font></font>
                e.preventDefault();<font></font>
                DeleteUser(user._id);<font></font>
            });<font></font>
        <font></font>
            linksTd.append(removeLink);<font></font>
            tr.appendChild(linksTd);<font></font>
        <font></font>
            return tr;<font></font>
        }<font></font>
        // сброс значений формы<font></font>
        docu-
ment.getElementById("reset").click(function(e){<font></font>
```

```
        <font></font>
            e.preventDefault();<font></font>
            reset();<font></font>
        })<font></font>
        <font></font>
        // отправка формы<font></font>
        document.forms["userForm"].addEventListener("submit", e =>
{<font></font>
            e.preventDefault();<font></font>
            const form = document.forms["userForm"];<font></font>
            const id = form.elements["id"].value;<font></font>
            const name = form.elements["name"].value;<font></font>
            const age = form.elements["age"].value;<font></font>
            if (id == 0)<font></font>
                CreateUser(name, age);<font></font>
            else<font></font>
                EditUser(id, name, age);<font></font>
        });<font></font>
        <font></font>
        // загрузка пользователей<font></font>
        GetUsers();<font></font>
    </script><font></font>
</body><font></font>
</html><font></font>
```

Учебное электронное издание комбинированного распространения

Учебное издание

НЕРЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ

Учебно-методическое пособие для студентов специальностей 1-40 05 01 «Информационные системы и технологии (по направлениям)» и 1-40 80 04 «Информатика и технологии программирования» дневной и заочной форм обучения

Составители: **Асенчик** Олег Даниилович
Токочаков Владимир Иванович

Электронный аналог печатного издания

Редактор *О. С. Ковалёва*
Компьютерная верстка *И. П. Минина*

Подписано в печать 22.12.22.
Формат 60x84/16. Бумага офсетная. Гарнитура «Таймс».
Ризография. Усл. печ. л. 15,11. Уч.-изд. л. 16,6.
Изд. № 34.
http://www.gstu.by

Издатель и полиграфическое исполнение
Гомельский государственный
технический университет имени П. О. Сухого.
Свидетельство о гос. регистрации в качестве издателя
печатных изданий за № 1/273 от 04.04.2014 г.
пр. Октября, 48, 246746, г. Гомель