**Нереляционные базы данных**

Объем данных в мире непрерывно растёт. Реляционные базы данных **плохо** приспособлены к работе с действительно **большими** объемами данных.

Популярность нереляционных баз данных (термин NoSQL-базы стал применяться с 2009 года) началась с появлением интернета и его главного инструмента — поисковых систем.

Для поиска данных нужно было обеспечивать параллельную обработку больших массивов данных и масштабируемость. Первой в этом направлении стала работать компания Google, создавшая крупнейшую поисковую систему и комплекс других сервисов и приложений, работавших с большими объемами информации.

Создатели программного обеспечения стали разрабатывать программные продукты, основанные на принципах доступности, простоты и масштабируемости.

В **реляционных** базах данных информация хранится в таблицах — наборах связанных друг с другом записей, организованных в столбцах и строках. При этом для них заранее должна быть определена схема, по которой приложения будут записывать информацию в БД. Связь имеется и между таблицами с помощью ключей, которые позволяют, например, извлечь определенную строку для проверки или изменения.

В **нереляционных** БД строго определенной схемы взаимосвязи между данными нет. Это позволяет быстро адаптировать базу данных в зависимости от того, с каким типом информации в конкретный момент времени работает приложение.

**Особенности NoSQL**

Термин объединяет множество СУБД, имеющих различную архитектуру и характеристики. Однако, можно выделить несколько присущих им всем особенностей:

**Неструктурированность**. В NoSQL-базах структура данных не регламентирована вообще или лишь в малой степени. Если нужно внести изменения в поля отдельного документа, для этого не потребуется менять всю структуру таблицы. При необходимости поменять модель данных потребуется просто указать изменения в коде приложения.

**Использование альтернатив языку SQL**. В нереляционных базах данных не применяется именно SQL. В то же время разработчики многих NoSQL-СУБД применяют языки управления данными, в той или иной мере похожие на него по синтаксису.

**Агрегация данных**. В то время как реляционные БД сохраняют данные в виде таблиц, в нереляционных они представляют собой целостные объекты.

Например, в базе данных электронного магазина заказ и позиции заказа объединяются в один логический объект, причем позиции содержат ссылку на товар и его характеристики, которые регулярно демонстрируются пользователю. Если же вместе с заказом регулярно извлекаются платежи, то имеет смысл объединить их в один объект. То есть агрегация данных адаптируется под модель работы конкретного приложения.

**Распределенность**. В нереляционных базах данных реализована горизонтальная масштабируемость. Она достигается за счет соединения быстрым подключением нескольких независимых друг от друга серверов, каждый из которых обрабатывает только часть данных, а не весь массив. Соответственно, нет необходимости наращивать мощность каждого сервера — достаточно просто добавить в систему новый.

**Виды NoSQL-баз данных**

**Документо-ориентированные**. В базах данных этого типа данные записываются в специальный документ в формате JSON или близком к нему. Таким БД свойственны одновременно иерархичность и гибкость. Чаще всего они применяются в системах управления контентом, в каталогах, специализированных поисковых системах (например, в электронных архивах).

**Ключ-значение**. В таких БД для доступа к значению используется ключ. Они применяются в качестве хранилищ изображений, специализированных файловых систем, кэшей, информационных платформ для онлайн-игр и т.д. — везде, где главными требованиями являются высокая масштабируемость и минимальная задержка обработки запроса.

**Матричные**. В таких БД данные хранятся в виде разреженных матриц, где строки и столбцы используются как ключи доступа к значению. Чаще всего такие СУБД применяют в индексировании веб-страниц и других задачах, связанных с обработкой больших данных.

**Графовые**. Такие БД сохраняют информацию в виде сложно связанных друг с другом графов. Связанность данных упрощает их хранение, навигацию и поиск. Типичными примерами использования графовых БД являются социальные сети, системы выявления мошенничества.

NoSQL базы: **MongoDB**, **Cassandra**, **Apache HBase**, **Oracle NoSQL,** **DynamoDB**, **Redis** и др. Различные базы данных NoSQL предназначены для решения различных задач.

**Преимущества NoSQL**

**Горизонтальная масштабируемость** — для увеличения производительности достаточно добавить новый сервер в систему, а не наращивать мощности уже имеющихся.

**Высокая устойчивость** — так как NoSQL-базы размещаются на независимых серверах, выход одного из них из строя не обрушит всю базу данных и, следовательно, не приведет к полному отказу приложения.

**Производительность** — с одной стороны, каждый сервер обрабатывает только свои запросы, не растрачивая свои мощности на все; с другой — информационные модели таких СУБД адаптируются под специфику каждого приложения.

**Гибкость** — нереляционные БД могут работать с неструктурированными данными и различными моделями представления информации.

**Широкая применимость** — горизонтальная масштабируемость и производительность позволяют применять нереляционные БД в обработке больших данных, онлайн-играх, интернете вещей, электронной коммерции, научной деятельности, издательском бизнесе и т.д.

**Недостатки NoSQL**

**Ограниченность языка** — из-за чего в работе с ними приходится использовать сторонние инструменты для трансляции стандартных SQL-команд.

**Недостаточная надежность транзакций** — из-за того, что NoSQL-БД заточены под высокую производительность и масштабируемость, в них страдает согласованность данных, критически важная для таких, например, сфер, как денежные переводы.

**База данных MongoDB**

**MongoDB** − это кроссплатформенная ***документно-ориентированная*** технология организации данных (написана на языке [C++](https://ru.wikipedia.org/wiki/C%2B%2B)).

Компания **10gen** начала разработку MongoDB в **2007** г. В 2009 году компания перешла на модель разработки продуктов с открытым исходным кодом, предлагая коммерческую поддержку и другие услуги.

В 2013 г. 10gen изменила свое название на MongoDB, Inc.

Есть официальные драйверы для основных языков программирования (Си, C++, C#, [Go](https://ru.wikipedia.org/wiki/Golang), Java, Node.js,  Perl,  PHP,  Python,  Ruby и др.)

База данных NoSQL − это база, в которой структура таблиц не является фиксированной, в отличие от структурированных баз данных SQL. Это позволяет разработчикам **быстро вносить изменения** в объекты без необходимости каких-либо изменений на уровне базы данных.

СУБД MongoDB применяется в различных сферах.

1. Социальные сети, чаты, генераторы новостей и схожие сценарии использования.
2. Большие данные — нереляционная структура системы хорошо подходит для работы с big data.
3. Каталоги для магазинов электронной торговли, содержащие большое количество разных наименований товаров.
4. Работа с данными, основанными на местоположении — геопространственными данными.
5. Сбор и обработка информации с различных датчиков и считывающих устройств;
6. Сервисы блогосферы, особенно те, которые подразумевают большое количество изображений, аудио и видеоматериалов.

Система плохо подходит для хранения сильно связанных данных и проектов, основной упор в которых сделан на транзакции на уровне базы данных. К **минусам** MongoDB традиционно относят:

* неполное соответствие требованиям ACID (реляционные БД в этом плане выигрывают);
* невозможность реализовать бизнес-логику на уровне БД, так как в базе нет положений о хранимых функциях и процедурах;
* сложная реализация транзакций (в 2018 г. в версию 4.0 добавлена поддержка транзакций).

**Общие сведения**

Работа с базой данных MongoDB основана на таких понятиях как коллекция и документ.

**Документом** является набор пар ключ-значение. Формат хранения данных − **BSON** (binary JSON − JavaScript Object Notation).

**JSON** – это текстовой формат обмена данными. Он основан на подмножестве языка программирования JavaScript.

JSON строится на двух структурах:

– набор пар ключ/значение. Ключом может быть только строка, значением – любая форма;

– пронумерованный набор значений. Во многих языках это реализовано как массив, вектор, список или последовательность.

Пример: { “hello”: “world” } { “BSON”: [“awesome”, 5.05, 1986] }

Формат обмена данными **BSON** (от англ. «Binary JavaScript Object Notation») – бинарная версия JSON.

Данные в JSON-формате занимают меньше места, но данные в формате BSON обрабатываются быстрее.

**Коллекция** − это группа документов, которые вместе формируют набор данных. В рамках одной коллекции допускается наличие *различных по структуре* документов.

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

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

{"signoff":"Good night, andgood luck"}

Документы имеют разные ключи, разное количество ключей и значения разных типов.

**Нормализация и денормализация**

**Денормализация** - это хранение нескольких данных в **одном документе JSON**.

Например, есть документ, в котором хранится информация о пользователях, туда же вставляются адреса каждого пользователя. Денормализация будет работать лучше при чтении, но будет медленнее при записи и займет больше места.

**Нормализация** - это хранение данных в нескольких коллекциях со ссылками между ними.

Например, сохранение информация о пользователях в одном документе и адресов в другом документе. Нормализация определяет данные только один раз, упрощая задачи записи (обновления). Когда дело доходит до чтения, нормализация имеет свои недостатки. Если надо получать данные из нескольких коллекций, то придется выполнять несколько запросов, что замедлит чтение.

Выбор способа хранения данных зависит от того, как будет использоваться база данных:

* Если база данных не нуждается в регулярных обновлениях, имеет небольшие документы, размер которых медленно увеличивается, немедленная согласованность при обновлении не очень важна, но нужна хорошая производительность при чтении, тогда *денормализация* может быть разумным выбором.
* Если в базе данных есть большие документы с постоянными обновлениями, и необходима хорошая производительность при записи, тогда можно рассмотреть возможность *нормализации*.

Отдельные коллекции могут быть полезны по ряду причин:

* хранение разных видов документов в одной коллекции может быть неудобным для разработчиков и администраторов. Сложно отсеять нужную информацию из документов, содержащих разные данные;
* получить список коллекций можно быстрее;
* помещая в одну коллекцию только документы одного типа, можно более эффективно индексировать свои коллекции.

Главным отличие БД mongo от РБД является отсутствие аналога операции ***соединения*** (JOIN). Если существует необходимость использовать соединения в базе данных, то они реализуются в программном коде приложения.

Одним из способов избавления от связей между документами является использование ***вложенных*** документов. Структуры в документах могут быть неограниченными по вложенности. Позволяя использовать вложенные документы и массивы, документо-ориентированный подход дает возможность представлять сложные иерархические отношения с помощью одной записи.

**Вложенные коллекции**

Одно из соглашений для организации коллекций состоит в том, чтобы использовать вложенные коллекции пространства имен, разделенные символом «.».

Например, приложение, содержащее блог, может иметь коллекцию с именем blog.posts и отдельную коллекцию с именем blog.authors. Это служит только организационным целям – нет никакой связи между коллекцией blog (она даже не должна существовать) и ее «потомками».

Хотя вложенные коллекции не имеют каких-либо специальных свойств, они полезны и включены во многие инструменты MongoDB. Например, GridFS, протокол для хранения больших файлов, использует вложенные коллекции для хранения метаданных файлов отдельно от блоков содержимого;

Большинство драйверов предоставляют возможности для доступа к вложенной коллекции.

Например, db.blog предоставит коллекцию blog, а db.blog.posts – коллекцию blog.posts.

Объединяя имя базы данных с коллекцией в этой базе данных, можно получить полное имя коллекции, которое называется *пространством имен*.

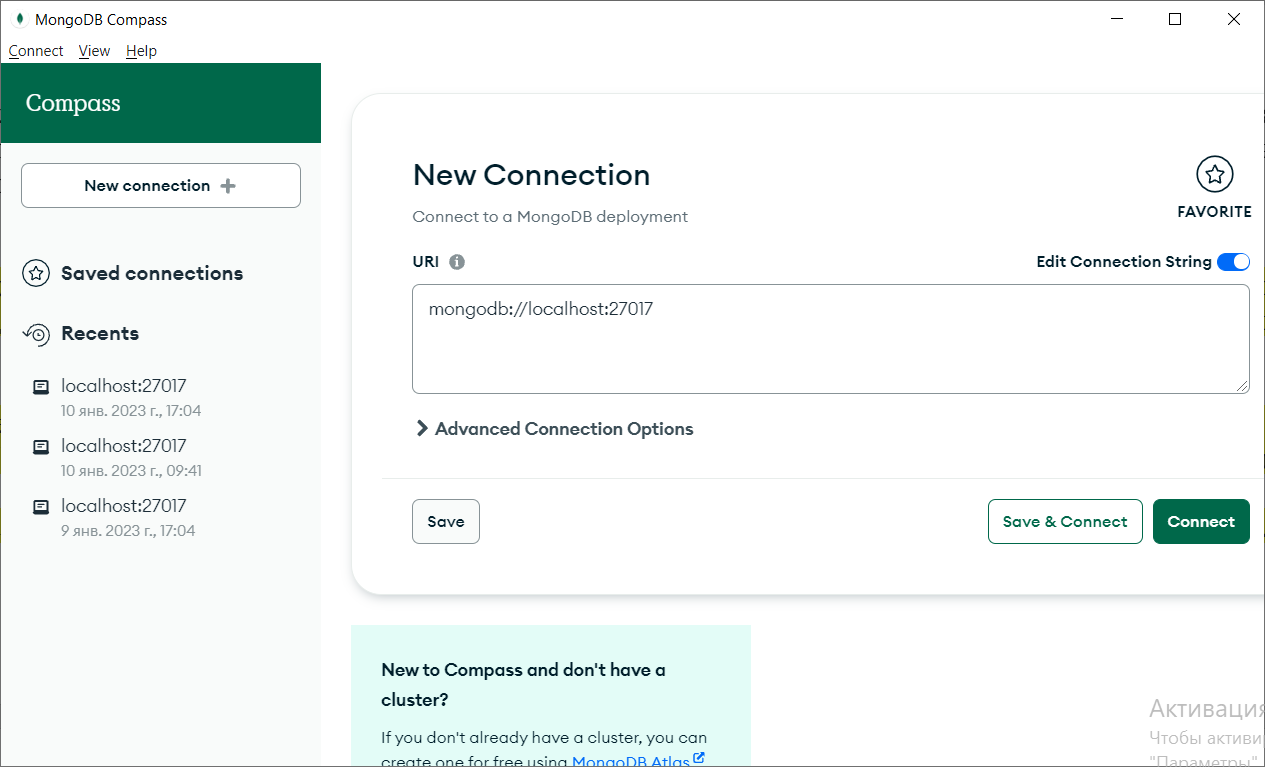
Например, если используется коллекция blog.posts в базе данных cms, пространство имен этой коллекции будет: cms.blog.posts.

Длина пространств имен ограничена 120 байтами, а на практике должна быть менее 100 байт.

## Подключение к серверу и создание базы данных

## с помощью MongoDB Compass

Для установки базы данных MongoDB нужно перейти на официальный сайт разработчиков: **https://www.mongodb.com/**.

На открывшейся странице следует выбрать сборку **MongoDB Community Server**, затем определить необходимые параметры, такие как версия, платформа, тип загружаемого архива или оставить все по умолчанию и нажать кнопку загрузки.

При установке сервера MongoDB также загружается консольный клиент **MongoDB Compass** для работы с базами данных.

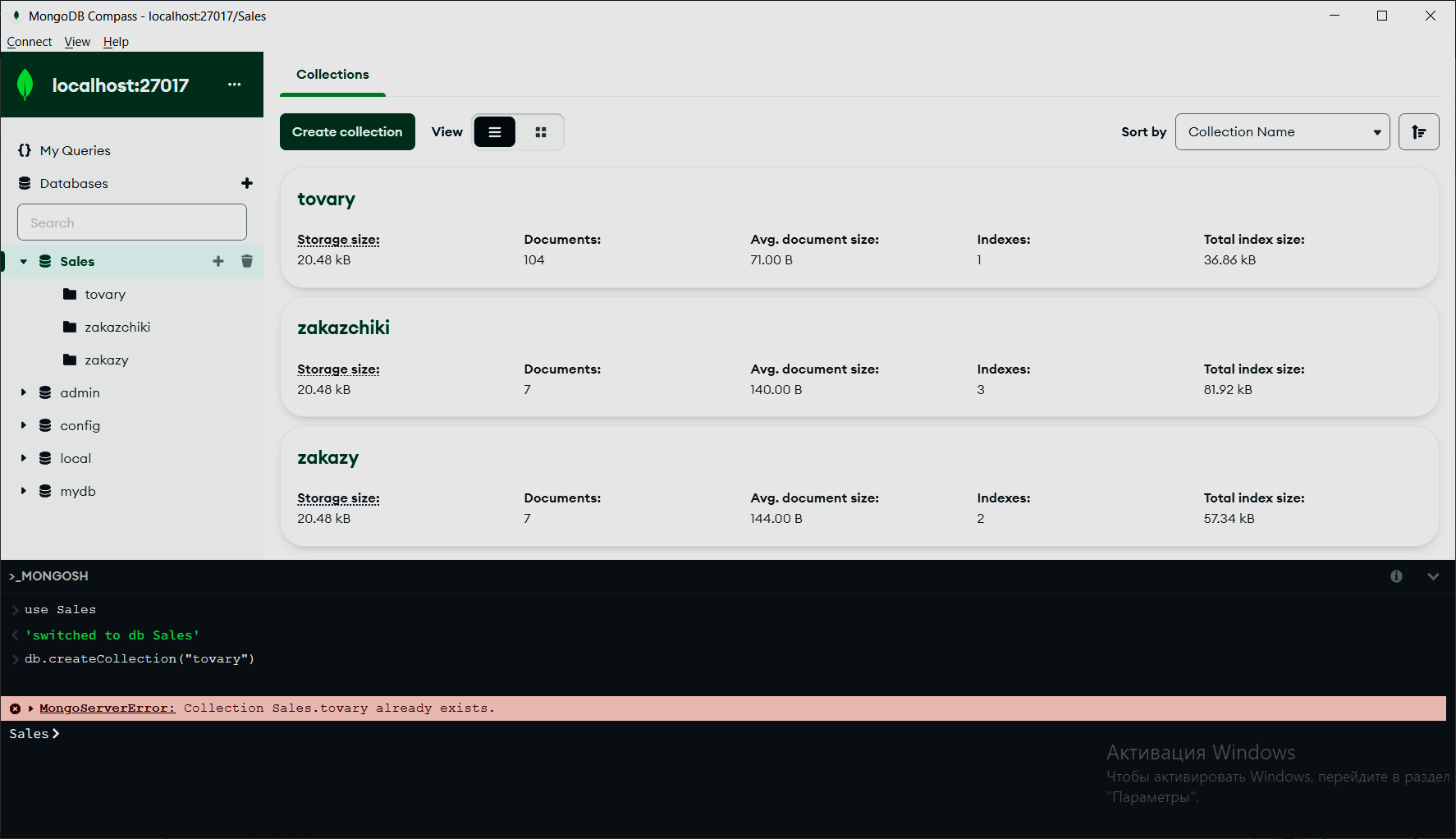
MongoDB - концептуально то же самое, что обычная база данных (или в терминологии Oracle - **схема**).

Используя графический интерфейс программы Compass, можно управлять данными, добавлять, изменять, удалять.

На первой странице следует нажать на кнопку **Connect**. После успешного подключения откроется содержимое сервера.

На вкладке **DataBases** отображается список всех базы данных по текущему соединению, с размером хранилища каждого из них, количество коллекций и количество индексов.

Имеется вкладка **Performance**, которая предоставляет информацию о производительности сервера, о том, выполнение каких операций занимает больше всего времени, как используется память.



Есть зарезервированные имена баз данных, к которым можно получить доступ, но которые имеют особую семантику:

**admin**

База данных admin играет роль в аутентификации и авторизации. Кроме того, доступ к этой базе данных необходим для ряда административных операций.

**local**

В этой базе данных хранятся данные, относящиеся к одному серверу. В наборах реплик в базе local хранятся данные, используемые в процессе репликации. Сама база данных local никогда не реплицируется.

**config**

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

Можно создавать базы данных с помощью как ***графического интерфейса***, так и с помощью ***команд*** JavaScript.

Для создания базы данных на вкладке **DataBases** служит кнопка **Create Databases**.

Команды вводятся во встроенной оболочке, которая открывается щелчком по ссылке **\_MONGOSH** в нижнем левом углу окна.

Чтобы посмотреть, какая база данных используется в текущий момент времени, надо выполнить команду **db**. Для просмотра списка всех созданных баз служит команда **show dbs**.

Начиная работать с MongoDB, первым делом надо установить нужную нам базу данных в качестве текущей, чтобы затем ее использовать. Для этого надо использовать команду **use**, после которой идет название базы данных. При этом не важно, существует ли такая база или нет. Если ее нет, то MongoDB автоматически создаст базу при добавлении в нее данных.

В MongoDB имеются следующие **типы** значений:

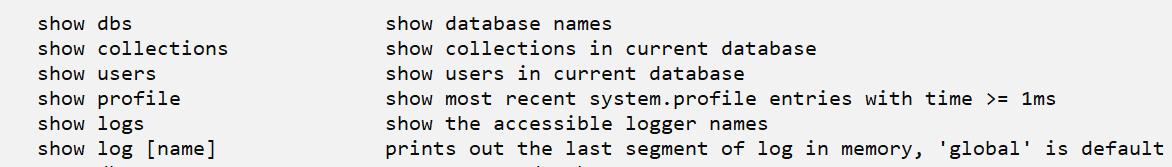
* **String**: строковый тип данных, как в приведенном выше примере (для строк используется кодировка UTF-8)
* **Array (массив)**: тип данных для хранения массивов элементов
* **Binary data (двоичные данные)**: тип для хранения данных в бинарном формате
* **Boolean**: булевый тип данных, хранящий логические значения TRUE или FALSE, например, {"married": FALSE}
* **Date**: хранит дату в формате времени Unix
* **Double**: числовой тип данных для хранения чисел с плавающей точкой
* **Integer**: используется для хранения целочисленных значений, например, {"age": 29}
* **JavaScript**: тип данных для хранения кода javascript
* **Min key/Max key**: используются для сравнения значений с наименьшим/наибольшим элементов BSON
* **Null**: тип данных для хранения значения Null
* **Object**: строковый тип данных, как в приведенном выше примере
* **ObjectID**: тип данных для хранения id документа
* **Regular expression**: применяется для хранения регулярных выражений
* **Symbol**: тип данных, идентичный строковому. Используется преимущественно для тех языков, в которых есть специальные символы.
* **Timestamp**: применяется для хранения времени

**Создание базы данных**

Начиная работать с MongoDB, первым делом надо установить нужную базу данных в качестве текущей с помощью команды **use**, после которой идет название базы данных. При этом не важно, существует такая база или нет. Если ее нет, то MongoDB автоматически создаст базу при добавлении данных.

MongoDB предоставляет полезный ***список команд базы данных*** через **db.help ().**

Существует также список команд **show** для отображения различных элементов, таких как базы данных, коллекции, пользователей и другие.



Создать базу данных **Sales** и наполнить коллекцию **tovary** можно с помощью команд:

**use Sales**

**db.createCollection("tovary")**

**db.tovary.insert([ {"name": "Стол", price: 19, quantity: 50},**

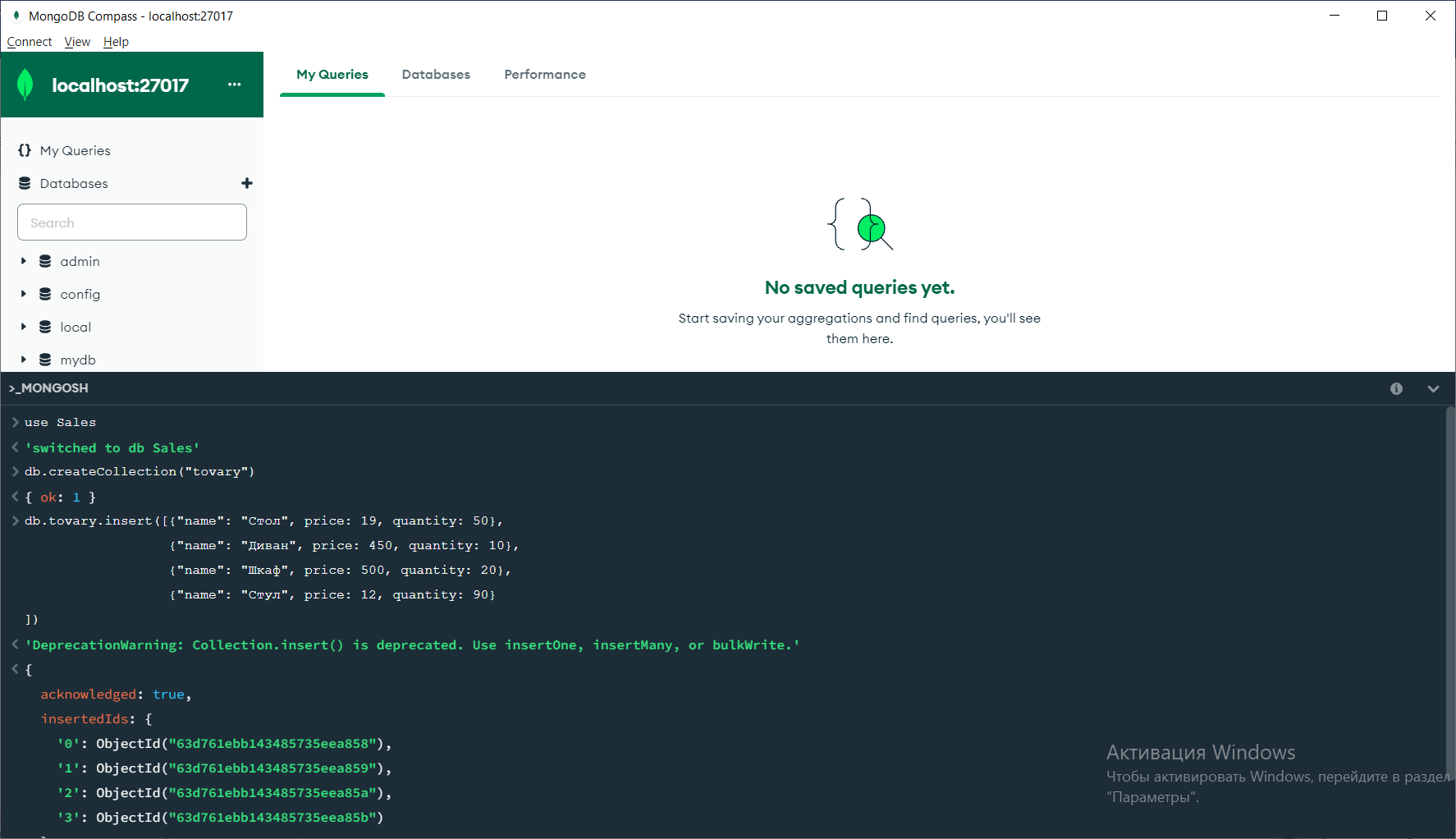
**{"name": "Диван", price: 450, quantity: 10},**

**{"name": "Шкаф", price: 500, quantity: 20},**

**{"name": "Стул", price: 20, quantity: 90}**

**])**

Получен ответ от сервера об успешном создании коллекции в виде объекта с ключом «ok», установленным в 1.



Для каждого документа автоматически генерируется уникальный идентификатор, который называется **\_id**.

В MongoDB запросы обладают **регистрозависимостью** и строгой типизацией. То есть следующие два документа не будут идентичны:

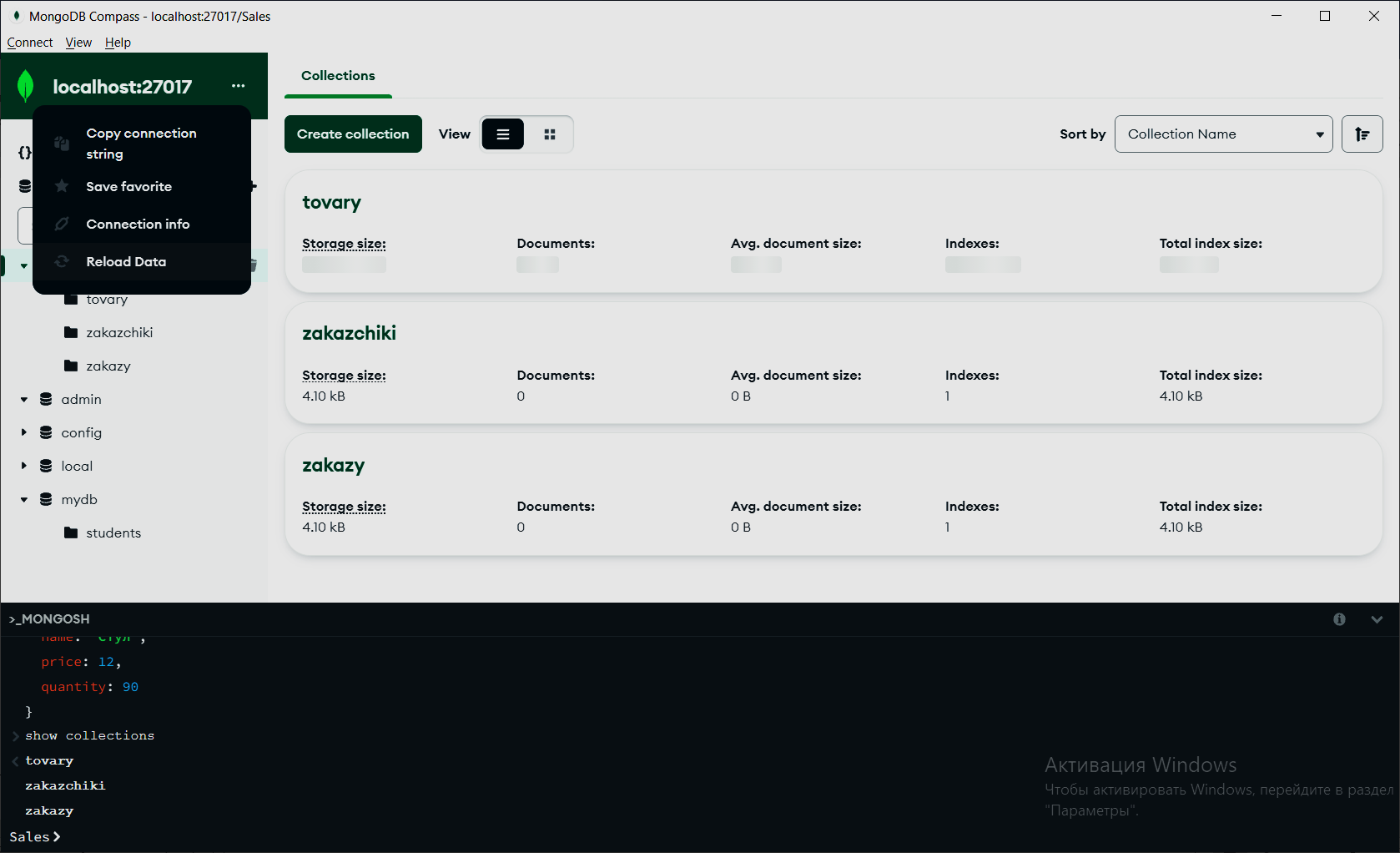
1 {"age": "28"}

2 {"age": 28}

Если в первом случае для ключа **age** определена в качестве значения строка, то во втором случае значением является число.

Чтобы просмотреть информацию о всех коллекциях базы данных используется команда **show collections**.

Для удаления коллекций предусмотрена функция **db.collection\_name.drop()**, где вместо **collection\_name** указывается имя коллекции, которую необходимо удалить.



Используя *графический интерфейс* **MongoDB Compass** можно просмотреть все коллекции и их содержимое. Можно удалить коллекцию и вновь добавить другую.

После внесения изменений надо **обновлять** базу данных с помощью команды **Reload Data**.

Существует несколько команд, с помощью которых ***добавляются*** данные:

− **insertOne()** в качестве аргумента принимает только один документ;

− **insertMany()** добавляет несколько документов в коллекцию;

− **insert()** − универсальная функция.

Добавление в коллекцию **tovary** одного документа:

**use Sales**

**db.tovary.insertOne({"name": "Полка", price: 21, quantity: 20})**

Проверить, добавлен ли документ, можно с помощью оператора:

**db.tovary.find()**

В случае, когда нужно за один запрос вставить несколько документов в коллекцию, целесообразно использовать команду **insertMany()**, проверить работу которой можно также с использованием **find()**.

Функция **insert()** является универсальной, она объединяет возможности **insertOne()** и **insertMany()**, а также предоставляет другие возможности добавления документов. Можно заранее определить документ и только затем выполнить его вставку.

Например, пусть создан массив документов с именем **zakazchikiArray**, содержащий информацию о двух заказчиках:

**zakazchikiArray = [**

**{"company": "IBA", "address": "Минск, ул.Кульман, 1", "setl\_account": "876458"},**

**{"company": "Iisot", "address": "Гродно, ул.Гая, 2", "setl\_account": "541988"}**

**]**

Теперь можно использовать функцию **insert()**, где в качестве параметра следует передать ранее созданный массив:

**db.zakazchiki.insert(zakazchikiArray)**

Документы могут содержать вложенные объекты, например, в коллекцию **zakazy** добавлен документ:

**db.tovary.insert({"name": "Стол", characteristics: { material: "сосна", length: "150см"}, price: 19, quantity: 50})**

Здесь определяется вложенный объект с ключом **characteristics**.

***Изменить*** документ можно с помощью команд:

* **updateOne** обновляет один документ в коллекции, соответствующий фильтру. Если совпадают несколько документов, то будет обновлен только первый совпадающий документ;
* **updateMany** обновляет несколько документов в коллекции, соответствующих фильтру.

**db.tovary.updateOne({"name": "Стул"}, {$set: {price: 25}})**

Для ***удаления*** ***данных*** имеются команды:

* **deleteOne** удаляет один документ из коллекции, соответствующий фильтру. Если совпадают несколько документов, удаляется только первый совпадающий документ;
* **deleteMany** удаляет несколько документов из коллекции, соответствующих фильтру.

**db.tovary.deleteOne({"name": "Стул"})**

Если надо полностью ***заменить*** один документ другим может использоваться функция **replaceOne**:

**db.tovary.replaceOne({"name": "Стол"}, {"name": "Стол", price: 22, quantity: 60})**

Для ***удаления документа*** из коллекции существует функция **remove()**, которая принимает два аргумента. Первый аргумент − это критерий, по которому документы будут удаляться, второй аргумент (не обязательный) позволяет допустить удаление только одного документа.

Чтобы удалить товар с названием **Стол** из коллекции **tovary** базы данных **Sales** можно использовать запрос:

**db.tovary.remove({name: "Стол"})**

После удаления документа будет выведен обновленный список предметов.

Чтобы удалить все документы в коллекции, но при этом коллекция должна сохраниться, нужно выполнить функцию **remove()** и в качестве критерия передать пустой объект:

**db.tovary.remove( {} )**

Если нужно удалить все документы и содержащую их коллекцию используется функция **drop()**.

**db.tovary.drop()**

**show collections**

Чтобы удалить всю базу данных нужно использовать команду **dropDatabase()**.

Например, чтобы удалить базу данных **Sales** и затем проверить ее отсутствие среди баз данных на сервере, надо использовать команды:

**db.dropDatabase({"dropped": "Sales", "ok": 1})**

**show dbs**

После обновления базы с помощью **Reload Data** можно убедиться, что база данных была успешно удалена, как и все её коллекции и документы.

Элементы в виде массива можно добавить с помощью **insertOne**.

Пусть требуется добавить в коллекцию **zakazchiki** для некоторых фирм информацию о телефонах в виде массива:

**Use Sales**

**db.zakazchiki.insertOne({"company": "Iba", "address": "Минск, ул.Короля, 3", "setl\_account": "123458", phone: ["132-43-55","132-48-51"]})**

**db.zakazchiki.insertOne({"company": "Белвест", "address": "Брест, ул.Герцена, 2", "setl\_account": "342988", phone: ["142-47-25","142-28-91"]})**

**Операторы выборки и объединения**

Для ***получения документов*** из коллекции используется метод **find()**.

Вывод записей, в которых содержится информация о фирме **Белвест**:

**db.zakazchiki.find({"company": "Белвест"})**

Помимо **find()** существует еще метод **findOne()**, который возвращает только один документ.

Метод **pretty()** отображает результат в структурированном формате. Он используется совместно с **find():**

**db.zakazchiki.find().pretty()**

В результате выполнения запроса будет выведено содержимое всей коллекции **zakazchiki** в отформатированном виде.

Условные операторы задают условие, которому должно соответствовать значение поля документа:

$**eq** (равно), $**ne** (не равно), $**gt** (больше чем), $**lt** (меньше чем), $**gte** (больше или равно), $**lte** (меньше или равно).

Например, найти документы, у которых значение ключа **price** меньше 30 можно с помощью запроса:

**db.tovary.find({price: {$lt: 30}})**

Сравнение здесь проводится над целочисленными типами. Если ключ представляет строковое значение, то строка заключается в кавычки.

Если надо найти все объекты со значением поля **price** больше 30, но меньше 50, то можно комбинировать два оператора:

**db.tovary.find({price: {$gt: 10, $lt: 150}})**

Оператор $**in** определяет массив значений, одно из которых должно иметь поле документа.

$**nin** определяет массив значений, которые не должно иметь поле документа.

Например, найти товары с ценой 19 и (или) 32:

**db.tovary.find({price: {$in: [19, 32]}})**

В запросах используются также операторы $**or**, $**and**, $**not**, $**nor** (соединяет два условия, и документ должен не соответствовать обоим условиям).

Например, найти все документы, в которых **name = "Луч"**, а **price** равно либо 12, либо среди значений **guantity** есть 50:

**db.tovary.find({name: "Стул", $or: [{price: 12}, {guantity: 50}]})**

Ряд операторов предназначены для работы с **массивами**:

$**all** определяет набор значений, которые должны иметься в массиве;

$**size** определяет количество элементов, которые должны быть в массиве;

$**elemMatch** определяет условие, которому должны соответствовать элементы в массиве.

Оператор **$exists** позволяет извлечь только те документы, в которых определенное поле присутствует или отсутствует.

Оператор **$type** извлекает только те документы, в которых определенное поле имеет значение определенного типа, например, строку или число:

Оператор **$regex** задает регулярное выражение, которому должно соответствовать значение поля (регулярное выражение - это серия символов, образующих шаблон для соответствия).

Например, пусть надо найти документы, в которых поле **name** имеет букву "**С**":

**db.tovary.find({name: {$regex:"С"}})**

**Проекция**

Конструкция, которая добавляет в выборку только нужные поля.

Например, пусть нужно найти из коллекции **zakazy** значения поля **customer** у всех документов, в которых **item\_name ="Стол"**:

**db.zakazy.find({item\_name: "Стол"}, {customer: 1})**

Использование **единицы** в качестве параметра указывает, что запрос должен вернуть только содержание поля **customer**.

Если надо найти все поля документа кроме поля **customer**, то в качестве параметра указывается 0 (вместо 1 и 0 можно использовать **true** и **false**).

При этом поле **\_id** будет включено в результирующую выборку. Если его выводить не нужно, то надо явным образом указать **{\_id:0}**.

**db.zakazy.find({item\_name: "Стол"}, {\_id: false, customer: 0})**

Если надо вывести все документы без учета фильтра, то можно оставить первые фигурные скобки пустыми:

**db.zakazy.find({}, {\_id: 0})**

**Вложенные документы**

Документы могут быть сложными по структуре. Добавим в коллекцию **zakazy** следующий документ:

**db.zakazy.insert({"item\_name": "Стул", sales\_price: 20, quantity: 4,**

**"delivery\_date": new Date('2023-01-11'), "customer": "Toi",**

**company: {"address": "Брест, ул.Кирова, 3", "setl\_account": "831251"}})**

Здесь определяется вложенный объект с ключом **company**.

Чтобы найти все документы, у которых в ключе **company** вложенное свойство **address =** **"Брест, ул.Кирова, 3"** надо использовать операцию “**.**”:

**db.zakazy.find({"company.address": "Брест, ул.Кирова, 3"})**

**Встроенные функции**

Функция **limit()** задает максимально допустимое количество получаемых документов.

Например, следующий запрос ограничивает выборку тремя документами:

**use Sales**

**db.tovary.find().limit(3)**

Функция **skip()** обеспечивает выборку документов не сначала, а пропустив некоторое количество документов. Например, запрос с пропуском первых трех записей:

**db.tovary.find().skip(3)**

Для сортировки данных предназначена функция **sort()**. Передавая в эту функцию значения 1 или −1, можно указать в каком порядке сортировать: по возрастанию или по убыванию.

Например, запрос с сортировкой по возрастанию по полю **name**:

**db.tovary.find().sort({name: 1})**

В запросе функции могут совмещаться:

**db.tovary.find().sort({name: 1}).skip(2).limit(3)**

Для объединения запросов предназначена функция **bulkWrite().**

**db.** **tovary.bulkWrite([**

**{insertOne: {"document": {"name": "Кресло", price: 49, quantity: 30}}},**

**{deleteOne: {"filter": {price: 19}}}**

**])**

Здесь объединяются операции добавления одной записи и удаления другой.

Допустимые операции для **bulkWrite():** [**insertOne**](https://docs.mongodb.com/manual/reference/method/db.collection.bulkWrite/#std-label-bulkwrite-write-operations-insertOne), [**updateOne**](https://docs.mongodb.com/manual/reference/method/db.collection.bulkWrite/#std-label-bulkwrite-write-operations-updateOneMany), [**updateMany**](https://docs.mongodb.com/manual/reference/method/db.collection.bulkWrite/#std-label-bulkwrite-write-operations-updateOneMany), [**deleteOne**](https://docs.mongodb.com/manual/reference/method/db.collection.bulkWrite/#std-label-bulkwrite-write-operations-deleteOneMany), [**deleteMany**](https://docs.mongodb.com/manual/reference/method/db.collection.bulkWrite/#std-label-bulkwrite-write-operations-deleteOneMany), [**replaceOne**](https://docs.mongodb.com/manual/reference/method/db.collection.bulkWrite/#std-label-bulkwrite-write-operations-replaceOne).

С помощью функции **count**() можно получить число элементов в коллекции:

**use Sales**

**db.zakazy.count()**

Подсчитать, сколько документов в таблице **zakazy**, у которых **item\_name =** **"Стол"**:

**db.zakazy.find({item\_name: "Стол"}).count()**

или

**db.zakazy.count({item\_name: "Стол"})**

Можно создавать цепочки функций, чтобы конкретизировать условия подсчета:

**db.zakazy.find({item\_name: "Стол"}).skip(1).count(true)**

По умолчанию функция **count()** не используется с функциями **limit()** и **skip()**. Чтобы использовать функции совместно в функцию **count()** надо передать булево значение **true***.*

Функция **distinct()** удаляет из результата документы с повторяющимися ключами.

**db.zakazy.distinct("item\_name")**

В консоль будет выведен массив найденных уникальных имён.

**Блок try-catch**

Пример использования блока **try**-**catch** при удалении документа:

**try {**

**db.tovary.bulkWrite([{ deleteOne: { "filter" : { "name" : "Кресло"} } } ]);**

**}**

**catch (error) { print(error); }**

## Команды группировки

Основное назначение **группировки** − разбиение множества строк на группы в соответствии с заданным фильтром, а также выполнение вычислений над группами строк с помощью функций:

$**avg** (вычисление среднего значения), $**max** (вычисление максимального значения), $**min** (вычисление минимального значения), $**sum** (вычисление суммы значений) и др.

Метод **aggregate()** удобно использовать, если нужно подсчитать по определённому фильтру сумму, среднее значение и пр.

Например, узнать общие суммы однотипных товаров можно с использованием запроса:

**db.zakazy.aggregate([ {$match: {}},**

**{$group: {\_id:"$item\_name",**

**"Общая сумма": {$sum: "$sales\_price"}}}])**

Здесь секция ***match*** отвечает за фильтр агрегации, причём пустые скобки говорят о том, что фильтра нет, а значит агрегация проводится по всем записям коллекции.

Секция ***group*** указывает, какие поля должны быть в итоговой выборке, ***\_id*** обозначает поле для группировки.

**sales\_price** *−* агрегируемое поле, к нему применяется агрегатная функция *$***sum**.

Тот же результат можно получить, выполнив следующий запрос:

**db.zakazy.aggregate([{$group:{\_id: "$item\_name",**

**"Общая сумма": {$sum: "$sales\_price"}}}])**

**Курсоры**

**Курсор** − это результат выборки, полученный с помощью функции **find()**.

Чтобы определить курсор и не выводить сразу все содержащиеся в нем данные, после метода **find()**добавляется через точку с запятой выражение **null**:

**var cursor = db.tovary.find(); null;**

Используя синтаксис языка JavaScript и различные методы, можно обработать полученные документы и вывести их на экран:

**var cursor = db.tovary.find();null;**

**while(cursor.hasNext())**

**{**

**obj = cursor.next();**

**print(obj["name"]);**

**}**

Здесь метод **hasNext**() определяет при переборе данных, имеется ли еще в наборе документ, метод **next**() извлекает текущий документ и перемещает курсор к следующему документу в наборе. В итоге в переменной **obj** оказывается документ, к полям которого можно получить доступ.

Для перебора документов в курсоре можно также использовать конструкцию **forEach**:

**var cursor = db.tovary.find()**

**cursor.forEach(function(obj)**

**{ print(obj.name); })**

Чтобы ограничить размер выборки, используется метод **limit**:

**var cursor = db.tovary.find();null;**

**cursor.limit(3);null;**

**cursor.forEach(function(obj)**

**{ print(obj.name); })**

Используя метод **sort()**, можно отсортировать документы в курсоре:

**var cursor = db.tovary.find();null;**

**cursor.sort({name:1});null;**

**cursor.forEach(function(obj)**

**{ print(obj.name); })**

Выражение **cursor.sort({name:1})** сортирует документы по полю **name** в порядке возрастания. Для сортировки по убыванию, вместо 1 надо использовать −1.

Метод **skip()** позволяет пропустить при выборке определенное количество документов:

**var cursor = db.tovary.find();null;**

**cursor.skip(2);null;**

**cursor.forEach(function(obj)**

**{ print(obj.name); })**

**Цепочка** — это такое **объединение методов**, при котором каждый метод выполняется в той последовательности, в которой записан в цепочке.

При этом каждый отдельный метод может выполняться самостоятельно. Если нарушено ограничение целостности, то цепочка не будет выполняться.

**var cursor = db.tovary.find();null;**

**cursor.sort({name:1}).limit(3).skip(2);null;**

**cursor.forEach(function(obj) { print(obj.name); })**

**Использование индексов**

**Индексы** позволяют упорядочить данные определенным способом по некоторому полю, что впоследствии значительно ускорит поиск по этому полю.

Однако слишком много индексов в базе данных могут снизить её производительность.

MongoDB использует **B-tree** структуру данных для хранения деревьев. Позволяет установить до 64 индексов на одну коллекцию. Значение поля, по которому идет индексация, не должно быть больше 1024 байт.

Самый простой пример индекса — **\_id**.

MongoDB создает уникальный индекс для поля **\_id** во время создания документа. Поскольку это поле проиндексировано, то идентификатор каждого документа хранится также в индексе, что позволяет быстро искать документы по **\_id**.

Для тестирования работы базы данных с индексами желательно, чтобы коллекция была заполнена большим количеством документов:

**use Sales**

**db.createCollection("probaIndex")**

**db.probaIndex.insert([**

**{"name": "Стол", price: 19, quantity: 50},**

**{"name": "Стол письменный", price: 49, quantity: 50},**

**{"name": "Стул", price: 20, quantity: 90}**

**])**

**for(let i=0;i < 1000;i++)**

**{db.probaIndex.insert([{"name": "Гвозди"+i, price: 9+i, quantity: 50}])}**

В результате выполнения данных операторов в коллекцию будет вставлено 100 тестовых документов.

С помощью метода **getIndexes()** можно вывести информацию об индексах для конкретной коллекции:

**use Sales**

**db.probaIndex.getIndexes()**

В выводе данного запроса будет несколько полей.

Поле **key** используется для поиска максимального и минимального значений, для различных операций, где надо применять данный индекс.

Поле **name** применяется в качестве идентификатора для операций администрирования.

Например, для удаления индекса с именем **name\_1** надо использовать команду:

**db.probaIndex.dropIndex("name\_1")**

**Простой индекс**. Чтобы найти несколько документов надо выполнить запрос:

**db.probaIndex.find({price: {$gte: 27, $lte: 30}})**

Этот запрос можно повторить с использованием метода **explain()**, который предоставляет сведения о плане запроса и позволяет оценить производительность.

**db.probaIndex.find({price:**

**{$gte: 27, $lte: 30}}).explain("executionStats")**

MongoDB запускает **Оптимизатор запросов**, выполняет план запроса и возвращает статистику.

Среди полученных результатов статистики

параметр **winningPlan stage: 'COLLSCAN'**, означает, что сканируется вся коллекция;

параметр **totalDocsExamined: 104** показывает количество просмотренных документов;

параметр **totalKeysExamined: 0** указывает, что запрос не использует индекс.

Пусть создан индекс по полю **price**:

**db.probaIndex.createIndex({price: 1})**

Метод **createIndex()** создает простой индекс по одному полю **price**. Тип индекса 1 (1 означает, что поле указано в порядке возрастания, а -1 – в порядке убывания).

Если повторить тот же поисковый запрос с методом **explain()**, то в статистике

параметр **stage: 'IXSCAN'** означает, что индекс используется;

параметр **totalDocsExamined: 4** показывает, что в поиске участвуют только 4 документа.

Также для оценки времени выполнения запросов можно использовать профилирование:

**db.setProfilingLevel(1, 50)**

В коллекцию **system.profile** будут попадать все запросы которые выполняются дольше 50 мc, но стоит учесть, что у системных коллекций ограничена память и на профайлер выделяется всего 1мб.

Отключить лог можно следующим образом:

**db.setProfilingLevel(0)**

Можно анализировать коллекцию профайлера, так же, как и остальные:

**db.system.profile.find({ millis: { $gt: 50 } })**

Будут найдены все запросы, которые длятся более 50 мс.

**Составной индекс**. Для того чтобы быстро найти информацию в нескольких полях, необходимо создать составной индекс, так как он поддерживает индексирование на основе нескольких полей (но не более 31).

**db.probaIndex.createIndex({"name": "text", price: 1})**

В данном запросе использован атрибут **"text"**, который применяется к текстовым полям для поиска в них слов.

Важно отметить, что порядок полей имеет значение, то есть по команде

**db.probaIndex.createIndex({price: 1, "name": "text"})**

будет создан другой индекс. Так происходит потому, что порядок полей метода **createIndex()** определяет порядок сортировки записей, а одни и те же записи, но по разному отсортированные − это различные данные.

Составной индекс можно сделать *уникальным*:

**db.probaIndex.createIndex({"name": "text", price: 1}, {"unique": true})**

Однако в этом случае все добавляемые документы должны иметь уникальные значения для обоих полей.

**OR-поиск, AND-поиск.** Чтобы найти записи, в индексируемых полях которых содержатся слова «Стол» или «Гвозди10», надо выполнить запрос:

**db.probaIndex.find({$text:{$search: "Стол Гвозди10"}})**

Такой поиск называется ***OR-поиском***, то есть поиск производится до тех пор, пока не обнаружится запись с хотя бы одним словом из перечисленных.

Если необходимо выполнить запрос по всей фразе целиком (***AND-поиск***), то это можно сделать, указав *двойные кавычки* в тексте поискового запроса.

Добавив к одному из слов префикс - (знак минус), можно убрать все записи, которые содержат это слово. Например, чтобы найти документ, который содержит ключевое слово «Стол», но не содержит «письменный», используется следующий запрос:

**db.probaIndex.find({$text:**

**{$search: "Стол -письменный"}})**

**Частичный индекс**. Если необходимо проиндексировать только те поля, которые удовлетворяют некоторому условию, необходимо использовать ***частичный индекс***. Частичные индексы предъявляют более низкие требования к хранилищу и поиск происходит быстрее, поскольку объем данных меньше.

Например, можно создать индекс с именем **name\_1** (если такой имеется, то его предварительно надо удалить):

**db.probaIndex.createIndex({name: 1},**

**{partialFilterExpression:{price: 30}})**

В этом примере создаётся индекс только к тем полям **name**, которым соответствует значение ключа **price** – 10.

Оценить например, запрос:

**db.probaIndex.find({price: {$gte: 27, $lte: 30}})**

можно с помощью метода **explain**.

**TTL-индекс.** Если надо удалять документы из коллекции по истечении определенного периода времени, можно создать ***TTL-индекс*** для поля. Такие индексы полезно применять для данных, которые регулярно обновляются. Данные в MongoDB удаляются фоновым заданием.

**db.probaIndex.createIndex({name: 1},**

**{expireAfterSeconds: 10})**

Такой индекс подходит для определенных типов информации, таких как машинные события, данные журнала и информация о сеансе, которые должны сохраняться в базе данных в течение конечного времени.

**// Удалить документ через 2 минуты  
 db.users.createIndex(  
 { email: 1 },  
 { expireAfterSeconds: 120 }  
 )**

**Многоключевые (multikey) индексы**. MongoDB использует **multikey** для индексации содержимого, хранящегося в **массивах**.

В таких индексах допускается несколько записей в массиве, указывающих на один и тот же документ.

Например, есть документ:

**{  
 name: 'Mихаил'  
 tags: [ 'tools', 'Mongodb', 'express' ]  
 }**

Если создать индекс по полю **tags**, то в нем будут перечислены все значения из **tags**, следовательно, для поиска требуется указать любой из его тегов. Именно в этом и состоит идея многоключевого индекса. Несколько ключей указывают на одну запись.

**Удаление индекса**. Хотя индексы и позволяют повысить поиск, но с каждым индексом связаны дополнительные накладные расходы.

Каждый раз, когда в коллекцию добавляется документ, его также нужно добавить во все индексы, связанные с этой коллекцией. Получается если над коллекцией построено 10 индексов, то при добавлении документа, нужно изменить 10 разных структур данных. И это относится к любой операции записи, добавления, удаления, обновления полей документа, которые проиндексированы.

Для приложений, ориентированных на чтение, затраты на индексы почти всегда оправданы. Нужно следить за тем, чтобы индексы, которые есть использовались, а если индексы не используются, то следует их удалять.

Индексу можно задать имя для его дальнейшего удаления или поиска.

**db.users.createIndex(  
 { email: 1 },  
 { name: 'catIdx' }  
)**

Чтобы удалить индекс используется команда:

**db.collection.dropIndex("catIdx");**

где **catIdx**  название индекса.

Можно удалить все индексы сразу:

[**db.collection.dropIndexes()**](https://docs.mongodb.com/manual/reference/method/db.collection.dropIndexes/#db.collection.dropIndexes)

## Функции пользователей

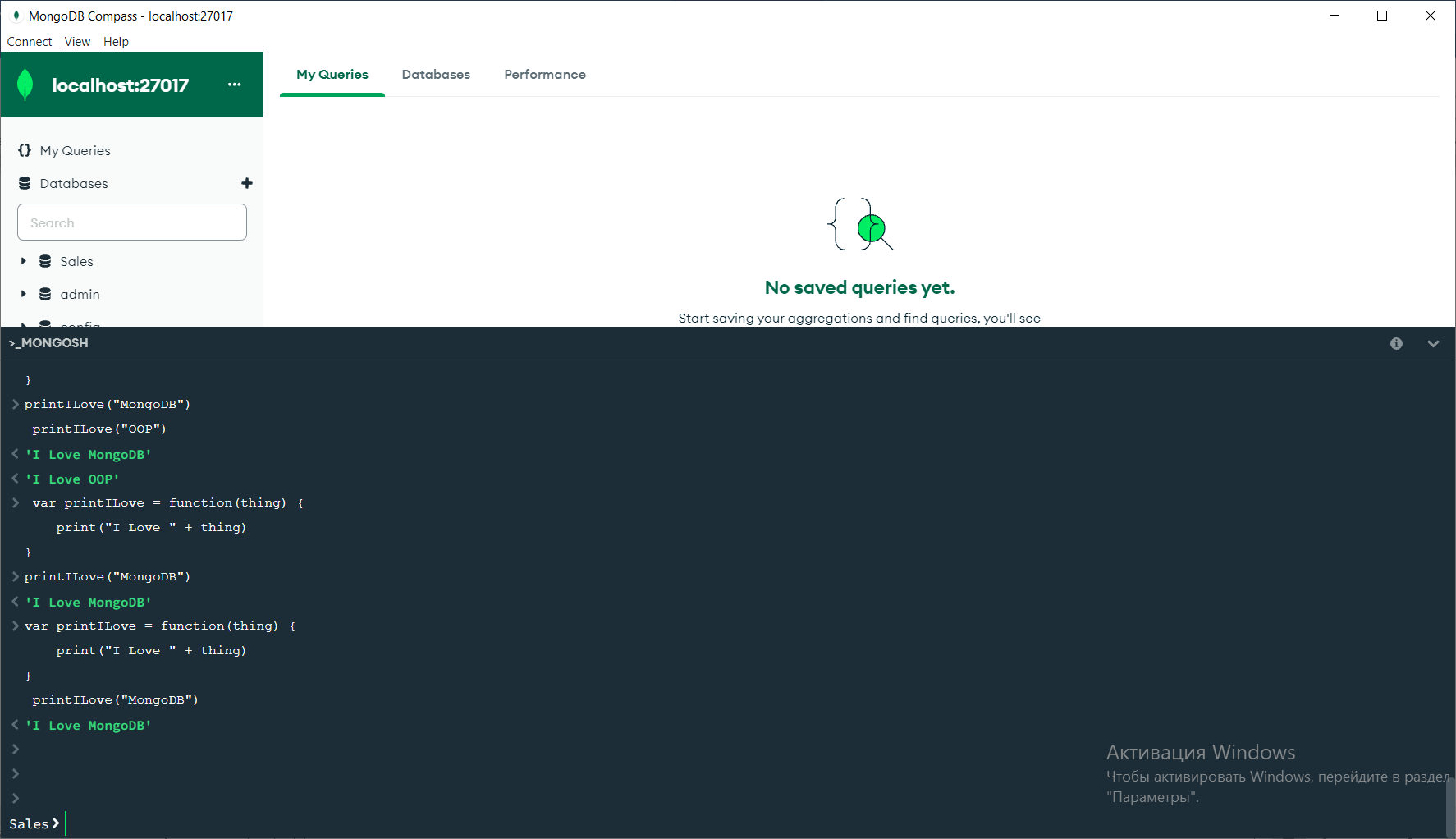
База данных MongoDB связана с форматом данных JSON, а также с языком JavaScript. Это делает возможным написание пользовательских функций, выполняющих различные задачи по добавлению и обработке данных.

Пусть написана простая функция пользователя, выводящая сообщение.

**var printILove = function(thing)**

**{ print("I Love " + thing) }**

Вызов функции: **printILove("MongoDB")**



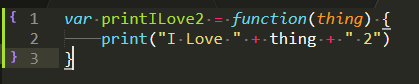
Функция, вычисляющая квадрат числа:

**function squareroot(n) { return n\*n; }**

Вызов функции: **squareroot(5)**

Писать код можно во внешнем файле с расширением **.js**.

Для этого на жестком диске, например, D, надо создать файл с расширением **.js**, содержащий скрипт, например, из задания 1:



Затем файл со скриптом необходимо импортировать при помощи команды **load()**, в параметрах которого нужно записать путь к файлу.

**load("D:/printILove2.js")**

Определенную в скрипте функцию можно вызывать так же, как если бы она была определена внутри оболочки MongoDB.

**printILove2("Music")**

**Примечание.**

В настоящее время команда **load()** не поддерживается клиентом MongoDB Compass, поэтому следует воспользоваться командной строкой (**cmd** или **Windows PowerShell**). В командной строке запускается MongoDB при помощи команды **mongosh**, указывается используемая база данных и прописывается команда **load()**, в параметрах которой указывается путь к файлу.

Во внешних файлах можно писать не только функции, но и любые скрипты, например, скрипт создания и заполнения коллекции.

Пусть записан такой скрипт в файле **myCollection**.

**[ 1 db.tovary.insert([{"name": "Полка", price: 10, quantity: 30},**

**2 {"name": "Консоль", price: 43, quantity: 15}]**

**3 ])**

Его можно загрузить в базу данных:

**load("D:/mongodb/myCollection.js")**

Проверить результат работы можно с помощью оператора:

**db.tovary.find()**

Например:

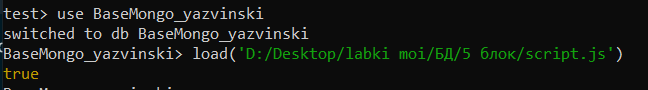
var addCheck = function(name, price, buyer, shop){

    db.tovarniy\_check.insert({"товар": {"название": name, "цена": price},

    "покупатель": buyer, "магазин": shop})

}

Для того, чтобы применить эту функцию в MongoDB при помощи функции load(), необходимо воспользоваться командной строкой (cmd или Windows PowerShell). В командной строке мы запускаем MongoDB при помощи команды mongosh, указываем используемую базу данных и прописываем команду load(), в параметрах которой нужно указать путь к файлу.



Далее запускается внешняя команда с необходимыми параметрами



**Администрирование**

Некоторые **встроенные роли** MongoDB представлены в таблице.

|  |  |
| --- | --- |
| **read** | доступ в режиме чтения |
| **readWrite** | чтение и редактирование данных |
| **dbAdmin** | выполнение административных задач, связанных со схемой, индексирование и сбор статистики |
| **userAdmin** | создание и изменение ролей и пользователей в текущей базе данных |
| **dbOwner** | чтение и редактирование данных, добавление и удаление пользователей, выдача прав пользователям и создание ролей |
| **dbAdminAnyDatabase** | можно создавать индексы, вызывать процедуры сжатия данных, выполнять задачи, связанные со схемой, нельзя создавать пользователей и удалять |
| **root** | суперпользователь, но с ограничениями; он может выполнять большинство действий, но не все: не может изменить системную коллекцию |

Пример создания пользователяс именем “new\_one\_role”, паролем “with\_roles” и ролями: **readWrite** и **dbAdmin**.

**use admin**

**db.createUser(**

**{ user: "new\_one\_role",**

**pwd: "with\_roles",**

**roles: ["readWrite", "dbAdmin"]**

**});**

**Пример** cоздания администратора с именем "user\_admin" и паролем “hellojose” для указанной базы данных. Роль “userAdmin” позволит иметь права администратора *только для базы данных*, указанной в опции db, т.е. базы данных “example”.

**db.createUser(**

**{ user: "user\_admin",**

**pwd: "hellojose",**

**roles: [{role: "userAdmin", db:"example"}]**

**})**

**Пример удаления пользователя:**

**db.dropUser("new\_one\_role")**

В Mongodb имеется возможность создавать ***определяемые пользователем роли***.

Пример создания в базе данных **Sales** новой роли под названием **superRoot** и назначения этой роли любого ресурса и любого действия. Затем создается новый пользователь **anyDBA** с паролем и назначается ему новая роль **superRoot**.

**db = db.Sales("admin");**

**db.createRole(**

**{ role: "superRoot",**

**privileges: [{resource: {anyResource:true},**

**actions: ["anyAction"]}]**

**roles:[]**

**})**

**db.createUser(**

**{ user: "anyDBA",**

**pwd: "EWqeeFpUt9\*8zq",**

**roles: ["superRoot"]**

**})**

Пример создания роли **manageRole** для управления текущей операцией. Эта роль может уничтожить любую операцию.

**use admin**

**db.createRole(**

**{ role: "manageRole",**

**privileges: [{ resource: {cluster: true},**

**actions: ["killop", "inprog"]},**

**{ resource: {db: "", collection: ""},**

**actions: ["killCursors"] } ],**

**roles: []**

**})**

## Сопоставление в индексах

Вы можете думать об этом как о способе сортировать и сравнивать строки относительно языковых особенностей. Представьте, что мы добавили два документа в коллекцию:

**db.users.insertMany([{ name: 'Леня' }, { name: 'Лёня' }])**

Имя записывается в двух вариантах.

Чтобы выполнить поиск существует ключевое слово **collation**, которое позволяет искать в строках относительно некоторого количества символов:

**db.cities.find(  
 { name: 'San Jose' }  
 ).collation({ locale: 'en\_US', strength: 1 })**

strength: 1 будет игнорировать диакритические знаки. Мы там же можешь отсортировать документ по чувствительности к регистру.

**db.words.find({})  
 .sort({ v: 1 })  
 .collation({ locale: 'en\_US', caseLevel: true })**

Поэкспериментируйте со своими данными сами и посмотрите как они меняются.

**db.files.find()  
 .sort({ name: 1 })  
 .collation({ locale: 'en\_US', numericOrdering: true })**

Также в порядке возрастания строк, которые имею цифры в значении

Например, коллекция myColl имеет индекс поля category строки с локализацией сортировки fr.

**db.myColl.createIndex(  
 { name: 1 },  
 { collation: {  
 locale: "fr"  
 }  
 }  
)**

Следующая операция запроса, которая указывает ту же строку, что и индекс, может использовать индекс:

**db.myColl.find( { category: "cafe" } ).collation( { locale: "fr" } )**

Однако следующая операция запроса, которая по умолчанию использует «простой» двоичное сравнение, не может использовать индекс:

**db.myColl.find( { category: "cafe" } )**

Для составного индекса, где ключи префикса индекса не являются строками, массивами и вложенными документами, операция, которая задает другое сравнение, все равно может использовать индекс для поддержки сопоставлений с ключами префикса индекса.

Например, коллекция myColl имеет составной индекс для числовых полей score price и строковое поле category; индекс создается с локализацией сортировки fr для сравнения строк:

**db.myColl.createIndex(  
 { score: 1, price: 1, category: 1 },  
 { collation: { locale: "fr" } } )**

Следующие операции, которые используют ‘простое’ двоичное сравнение для строк, могут использовать индекс:

**db.myColl.find( { score: 5 } ).sort( { price: 1 } )  
db.myColl.find( { score: 5, price: { $gt: NumberDecimal( "10" ) } } ).sort( { price: 1 } )**

Следующая операция которая использует ‘простое’ бинарное сравнение для строк на индексируемом поле category. Мы можем использовать индекс, чтобы выполнить только score: 5 запрос:

**db.myColl.find( { score: 5, category: "cafe" } )**

## Покрытие запросов

Когда критерии запроса включают только индексированные поля, MongoDB будет возвращать результаты непосредственно из индекса без сканирования каких-либо документов или переноса документов в память. Эти охватываемые запросы могут быть очень эффективными.

## Оптимизация запросов

Как выявить медленный запрос и вообще, что такое медленный запрос? Ну попробуйте выполнить find в коллекции где один документ и в коллекции где их миллион. Но с использованием индексов можно добиться скорости как в однодокументной из миллионной. Для мониторинга того, сколько потребовалось запросу, чтобы выполниться воспользуемся методом **explain**:

**db.users.find({})  
 .sort({ nickname: 1 })  
 .limit(1)  
 .explain('allPlansExecution')**

Такой безобидный запрос может занять у нас в миллионной коллекции около 3 секунд. А вернет он примерно следующее:

**{  
 ...  
 millis: 14538  
 ...  
}**

Это значит, что у нас запрос длится более 14 секунд и его следует оптимизировать.

explain дает большое количество информации о запросах. Это один из самых важных инструментов диагностики запросов.

Для любого запроса можно добавить вызов explain в конце (так же, как и в случае с функциями sort или limit, вызов explain должен идти последним).

"imdb.rating" – какой индекс использовался.

"nReturned" – сколько документов фактически было возвращено в результате.

"totalKeysExamined" сообщает о количестве просканированных записей индекса, а

"totalDocsExamined" указывает, сколько документов было отсканировано.

"nscannedObjects" – количество отсканированных документов.

"executionTimeMillis" сообщает, насколько быстро был выполнен запрос, с момента, когда сервер получил запрос, до момента отправки ответа.

"isMultiKey": использование многоключевого индекса.

"nReturned": количество документов, возвращаемых по запросу.

"executionTimeMillis": количество миллисекунд, которое потребовалось базе данных для выполнения запроса. Чем ниже это число, тем лучше.

!

Старайтесь делать индексы универсальными и в тоже время специально под запросы.

Индексы наиболее эффективны при извлечении небольших подмножеств данных, а некоторые типы запросов быстрее работают без индексов. Индексы становятся все менее и менее эффективными, по мере того как нужно получать бóльший процент коллекции, потому что использование индекса требует выполнять поиск дважды: один раз для просмотра индексной записи, и второй раз – следуя за указателем индекса на документ.

Для сканирования коллекции требуется только одно: просмотр документа. В худшем случае (возврат всех документов в коллекции) использование индекса заняло бы вдвое больше поисков и, как правило, было бы значительно медленнее, чем сканирование коллекции.

Не существует строгого правила относительно того, когда индекс помогает, а когда мешает, поскольку в действительности это зависит от размера данных, индексов, документов и среднего набора результатов.

Как правило, индекс часто ускоряет работу, если запрос возвращает 30 % коллекции или более. Однако это число может варьироваться от 2 % до 60 %.

111111111111111

Все индексы базы данных хранятся в системной коллекции indexes. Для обращения к ней мы можем использовать функцию getIndexes, например, чтобы вывести всю информацию об индексах для конкретной коллекции:1

db.users.getIndexes()

***Многоключевые индексы*** создаются для индексации содержимого, хранящегося в массивах. Когда индексируется столбец, содержащий значение массива, MongoDB создает отдельные записи индекса для каждого элемента массива. Эти многоключевые индексы позволяют запросам выбирать документы, содержащие массивы, путем сопоставления с элементом или элементами массивов.

Например, пусть в таблицу **probaIndex** добавлена следующая запись, содержащая массивы в ключах:

**db.probaIndex.insert({"company": "Луч", "info": {**

**"address": {"cities": ["Минск", "Брест", "Витебск"],**

**"accounts": ["12678", "95378"]}}})**

Многоключевой индекс для данной записи может выглядеть так:

**db.probaIndex.find().sort({"info.address.cities.accounts":1})**

MongoDB автоматически создаёт многоключевой индекс в тех полях, где есть массивы, поэтому создавать индекс явно с помощью **createIndex()** не нужно. Индексируемые поля можно лишь при необходимости переопределить в методе **sort()**. В данном примере сказано, что если до выполнения поиска записи были отсортированы по первому ключу массива (**cities),** то после её выполнения записи отсортировались по ключу **accounts.**

Все индексы базы данных хранятся в системной коллекции **system.indexes**. Обратившись к ней, можно получить ***информацию о всех индексах***:

**db.system.indexes.find()**

Также можно воспользоваться методом **getIndexes()** и вывести информацию об индексах для конкретной коллекции:

**db.probaIndex.getIndexes()**

В выводе данного запроса будет несколько полей. Поле **key** используется для поиска максимального и минимального значений, для различных операций, где надо применять данный индекс.

Поле **name** применяется в качестве идентификатора для операций администрирования, например, для удаления индекса:

**db.probaIndex.dropIndex("name\_1")**

## Установка ссылок в БД

В реляционных базах данных можно устанавливать внешние ключи, когда поля из одной таблицы ссылаются на поля в другой таблице. В MongoDB также можно устанавливать ссылки.

### Ручная установка ссылок

Ручная установка ссылок сводится к присвоению значения поля \_id одного документа полю другого документа. Допустим, у нас могут быть коллекции, представляющие компании и работников, работающих в этих компаниях. Итак, сначала добавим в коллекцию companies документ представляющий компанию:

|  |  |
| --- | --- |
| 1 | > db.companies.insert({"\_id" : "microsoft", year: 1974}) |

Теперь добавим в коллекцию persons документ, представляющий работника. В этом документе будет поле company, представляющее компанию, где работает работник. И очень важно, что в качестве значения для этого поля мы устанавливаем не объект company, а значение ключа \_id добавленного выше документа:

|  |  |
| --- | --- |
| 1 | > db.tovary.insert({name: "Tom", age: 28, company: "microsoft"}) |

Теперь получим документ из коллекции tovary:

|  |  |
| --- | --- |
| 1 | > user = db.tovary.findOne() |

В данном случае имеется в виду, что выше добавленный элемент будет единственным в коллекции.

После этого консоль выводит полученный документ. И теперь найдем ссылку на его компанию в коллекции companies:

|  |  |
| --- | --- |
| 1 | > db.companies.findOne({\_id: user.company}) |

И если документ с таким идентификатором обнаружен, он отображается на консоли:

### Автоматическое связывание

Используя функциональность **DBRef**, мы можем установить автоматическое связывание между документами. Посмотрим на примере применение данной функциональности. Вначале добавим новый документ в коллекцию companies:

|  |  |
| --- | --- |
| 1  2 | > apple=({"name" : "apple", "year": 1976})  > db.companies.save(apple) |

Обратите внимание, что в данном случае сохранение идет с помощью метода save, не insert. Метод save при добавлении нового документа генерирует \_id. И после сохранения мы можем вывести документ на консоль: > apple

Теперь создадим новый документ для коллекции person, у которого ключ company свяжем с только что добавленным документом apple:

|  |  |
| --- | --- |
| 1  2 | > steve = ({"name": "Steve", "age": 25, company: new DBRef('companies', apple.\_id)})  > db.tovary.save(steve) |

И мы можем протестировать:

|  |  |
| --- | --- |
| 1 | > db.companies.findOne({\_id: steve.company.$id}) |

Посмотрев на примере, теперь разберем организацию ссылок между документами. Для связывания с документом apple использовалось следующее выражение company: new DBRef('companies', apple.\_id)}). Формальный синтаксис DBRef следующий:

|  |  |
| --- | --- |
| 1 | { "$ref" : название\_коллекции, "$id": значение [, "$db" : название\_бд ]} |

Первый параметр $ref указывает на коллекцию, где хранится связанный документ. Второй параметр указывает на значение, которое и будет представлять что-то типа внешнего ключа. Третий необязательный параметр указывает на базу данных.

При тестировании в качестве запроса на выборку указывается выражение \_id: steve.company.$id. Так как person.company представляет теперь объект new DBRef('companies', apple.\_id)}), то нам надо конкретизировать параметр steve.company.**$id**

**У MongoDB есть пять основных зон безопасности:**

* **Аутентификация.** Аутентификация LDAP централизует элементы в каталоге вашей компании.
* **Авторизация.** Авторизация определяет, какие права доступа предоставляет база данных в зависимости от роли пользователя.
* **Шифрование.** Шифрование можно разделить на At-Rest и In-Transit. Оно имеет решающее значение для обеспечения безопасности MongoDB.
* **Аудит.** Аудит подразумевает возможность видеть, кто что сделал в базе данных (Аудит доступен только в MongoDB Enterprise. Его нет в версии Community).
* **Управление.** Управление подразумевает проверку документов и конфиденциальных данных (таких как номер счета, пароль, номер социального страхования или дата рождения). Это касается как знания того, где хранятся конфиденциальные данные, так и предотвращения их ввода в систему.

**Администрирование**

При установке MongoDB создает по умолчанию три базы данных: ***admin***, ***config*** и ***local***.

Первым созданным пользователем в базе данных является **admin**. Пользователь-администратор имеет права для обслуживания всех пользователей.

Обычный пользователь MongoDB получает доступ только к тем данным, которые требуются для его роли. Роль в MongoDB предоставляет привилегии для выполнения некоторого набора операций с данным ресурсом.

MongoDB предоставляет ряд **встроенных ролей**.

**read**

Чтение данных несистемных и системных коллекций system.indexes, system.js и system.namespaces.

**readWrite**

Предоставляет те же привилегии, что и read, плюс возможность изменять данные во всех несистемных коллекциях и коллекции system.js.

**dbAdmin**

Выполнение административных задач, например, задач, связанных со схемой, индексирование и сбор статистики (не предоставляет привилегии для управления пользователями и ролями).

**userAdmin**

Создание и изменение ролей и пользователей в текущей базе данных.

**dbOwner**

Сочетает привилегии, предоставленные ролями readWrite, dbAdmin и userAdmin.

**clusterManager**

Выполнение действий по управлению и мониторингу в кластере.

**clusterMonitor**

Предоставляет доступ только для чтения к таким инструментам мониторинга, как MongoDB Cloud Manager и агент мониторинга Ops Manager. **hostManager** Мониторинг и управление серверами.

**clusterAdmin**

Сочетает привилегии, предоставляемые ролями clusterManager, clusterMonitor и host-Manager, а также действием dropDatabase.

**backup**

Предоставляет достаточные привилегии для использования агента резервного копирования MongoDB Cloud Manager или агента резервного копирования Ops Manager либо применения mongodump для резервного копирования всего экземпляра сервера mongod.

**restore**

Предоставляет привилегии, необходимые для восстановления данных из резервных копий, которые не включают данные из коллекции system.profile.

**readAnyDatabase**

Предоставляет те же привилегии, что и read для всех баз данных, кроме local и config, плюс действие listDatabases в кластере как единое целое.

**readWriteAnyDatabase**

Предоставляет те же привилегии, что и readWrite во всех базах данных, кроме local и config (фактически роль суперпользователя).

**root**

Предоставляет доступ к операциям и всем ресурсам ролей readWriteAnyDatabase, dbAdminAnyDatabase, userAdminAnyDatabase, clusterAdmin, restore и backup

## Создание пользователя с заданными встроенными ролями

В MongoDB пользователи создаются с помощью метода CreateUser().

Этот метод создает нового пользователя для базы данных, если указанный пользователь уже присутствует в базе данных, тогда этот метод вернет ошибку.

лучшей ролью суперпользователя будет [root](http://docs.mongodb.org/manual/reference/built-in-roles/#root).Синтаксис:

use admin

db.createUser(

{

user: "root",

pwd: "password",

roles: [ "root" ]

})

можете указать роль с документом, например:

|  |
| --- |
| { role: **"<role>"**, db: **"<database>"** } |

Например, чтобы создать ***пользователя с правами администратора***, предоставить ему доступ для чтения и записи данных, а также для изменения определенных настроек, надо сначала войти в базу данных администратора **admin**, а затем использовать метод **CreateUser()**.

**use admin**

**db.createUser(**

**{ user: "hello\_admin",**

**pwd: "hello123",**

**roles: [ { role:"readWrite",db:"config"}, "clusterAdmin" ]**

**});**

Здесь устанавливается имя пользователя “hello\_admin”, пароль “hello12” и роли администратора - ReadWrite, config, clusterAdmin.

**Пример.**

**db.createUser(**

**{ user: "new\_one\_role",**

**pwd: "with\_roles",**

**roles: ["readWrite", "dbAdmin"]**

**});**

Создается пользователь с именем “new\_one\_role”, паролем “with\_roles” и ролями:

**readWrite**

Предоставляет те же привилегии, что и read, плюс возможность изменять данные во всех несистемных коллекциях и коллекции system.js.

**dbAdmin**

Выполнение административных задач, например, задач, связанных со схемой, индексирование и сбор статистики.

**Пример.**

**db.createUser(**

**{ user: "user\_admin",**

**pwd: "hellojose",**

**roles: [{role: "userAdmin", db:"example"}]**

**})**

Cоздается пользователь с именем "user\_admin" и паролем “hellojose”. Назначается роль “UserAdmin”, так как пользователь должен быть администратором базы данных. Эта роль позволит иметь права администратора ***только для базы данных***, указанной в опции db, т.е. базы данных “example”.

## Создание пользователя с ограничениями аутентификации

Аутентификация - это процесс, который проверяет, известен или неизвестен пользователь, который пытается получить доступ к базе данных.

С помощью метода **CreateUser()** можно создать пользователя с ограничениями аутентификации, установив значение поля **authenticationRestrictions** .

Это поле предоставляет пользователю разрешение на аутентификацию и содержит поля:

**clientSource:** если значение этого поля присутствует, то сервер проверяет IP-адрес клиента, когда он проходит аутентификацию.

**Адрес сервера:** это список IP-адресов, к которым может подключаться клиент.

**Пример.**

**use admin**

**db.createUser(**

**{ user: "restrict",**

**pwd: passwordPrompt(),**

**roles: [ { role: "readWrite", db: "example" } ],**

**authenticationRestrictions: [{**

**clientSource: ["192.168.65.10"],**

**serverAddress: ["198.157.56.0"] }]**

**})**

Создается пользователь с именем “restrict” в базе данных администратора. Он может проходить аутентификацию только при подключении с IP-адреса **192.168.65.10** к адресу сервера с IP-адресом **198.157.56.0**.

В Mongodb можно **удалить пользователя**, используя метод **dropUser()**. Этот метод возвращает true, когда пользователь удаляется, в противном случае возвращает false.

**Пример удаления пользователя:**

**db.dropUser("hello\_admin")**

**Создание пользовательских ролей**

В Mongodb имеется возможность создавать ***определяемые пользователем роли***. Это специальные роли, которые группируют полномочия на выполнение определенных операций и обозначают их именем, чтобы предоставить этот набор прав пользователям.

Пример

**db = db.Sales("admin");**

**db.createRole(**

**{ role: "superRoot",**

**privileges: [{resource: {anyResource:true},**

**actions: ["anyAction"]}]**

**roles:[]**

**})**

**db.createUser(**

**{ user: "anyDBA",**

**pwd: "EWqeeFpUt9\*8zq",**

**roles: ["superRoot"]**

**})**

Эти команды создают в базе данных **Sales** новую роль под названием **superRoot** и назначают этой роли любой ресурс и любое действие. Затем создается новый пользователь **anyDBA** с паролем и назначается ему новая роль **superRoot**.

Пример создания роли **manageRole** для управления текущей операцией. Эта роль может уничтожить любую операцию.

**use admin**

**db.createRole(**

**{ role: "manageRole",**

**privileges: [{ resource: {cluster: true},**

**actions: ["killop", "inprog"]},**

**{ resource: {db: "", collection: ""},**

**actions: ["killCursors"] } ],**

**roles: []**

**})**

**Резервирование и копирование базы данных**

**Резервное копирование** базы данных выполняется с использованием команды **mongodump**, а восстановление — с использованием команды **mongorestore**.

Эти команды запускаются из системной **командной** строки в Windows.

 Вначале следует создать каталог, в котором будет хранится резервная копия, например, **$sudo mkdir /var/backups/mongobackups**

Затем выполнить команду **mongodump**:

**$sudo mongodump --db newdb --out /var/backups/mongoback-ups/`date +"%m-%d-%y"`**

Здесь использован формат **date +"%m-%d-%y"**, который автоматически получает текущую дату.

Как правило, резервное копирование выполняется регулярно, и обычно это делается в периоды наименьшей нагрузки на сервер.

При **восстановлении** базы данных MongoDB используется команда **mongorestore**, которая работает с двоичными резервными копиями, которые производятся **mongodump**.

Например, пусть надо восстановить базу данных **newdb** из резервной копии, созданной 29 октября 2023 года.

**sudo mongorestore --db newdb --drop /var/backups/mongoback-ups/10-29-20/newdb/**

1111

Чтобы создать резервную копию, используется команда:

**mongodump --out=C:\backups\db** //создание копии всех баз данных

**mongodump -h localhost:27017 -d my\_app --out=C:\backups\db** //создание копии конкретной отдельной базы данных

Для восстановления надо открыть командную строку CMD и перейти в папку, в которой существует база данных, например, папку XYZ (папка).

Теперь выполнить команду: **mongorestore -db XYZ**

Имя базы данных, куда будут импортированы данные, — это имя каталога в папке db, в которой, в свою очередь, должны быть файлы копии в форматах .bson и .json.

Полный список опций можно посмотреть на [официальном сайте](https://www.mongodb.com/docs/database-tools/mongorestore/#options) **www.mongodb.com/docs/database\_tools**

**Репликация** – это способ хранить идентичные копии данных на нескольких серверах, рекомендуется для всех рабочих развертываний.

В MongoDB репликация настраивается путем создания набора реплик.

Набор реплик – это группа серверов с одним первичным узлом, который получает операции записи, и несколькими вторичными узлами, где хранятся копии данных первичного узла.

Если первичный узел дает сбой, вторичные узлы могут выбрать новый первичный узел из их числа.

**Шардинг** обозначает процесс распределения данных по разным серверам.

Для описания этой концепции также иногда используется термин разделение.

Размещая подмножество данных на каждой машине, становится возможным хранить больше данных и обрабатывать бóльшую нагрузку, без необходимости наличия больших или более мощных машин – достаточно просто бóльшего количества менее мощных компьютеров.

Шардинг можно использовать и для других целей, включая размещение данных с более частым доступом для более производительного оборудования или разделение набора данных на основе географии, чтобы найти подмножество документов в коллекции (например, для пользователей, базирующихся в конкретной локали) рядом с серверами приложений, с которых к ним чаще всего обращаются.

Вручную шардинг можно осуществить практически с помощью любой СУБД. При таком подходе приложение поддерживает соединения с несколькими различными серверами баз данных, каждый из которых является полностью независимым. Приложение управляет хранением различных данных на разных серверах и выполняет запросы к соответствующему серверу, чтобы получить данные обратно. Такой вариант может хорошо работать, но его становится трудно поддерживать при добавлении или удалении узлов из кластера или при изменении распределения данных либо шаблонно в нагрузки.

MongoDB поддерживает автоматический шардинг, который пытается абстрагировать архитектуру от приложения и упростить администрирование такой системы.

Шардинг позволяет создавать кластер из множества серверов (шардов) и распределять коллекцию по ним, поместив подмножество данных в каждый шард. Это дает возможность приложению расти за пределами ресурсов автономного сервера или набора реплик.

В ходе репликации создается точная копия данных на нескольких серверах, поэтому каждый сервер – это зеркальное отражение любого другого сервера. И наоборот, каждый шард содержит различное подмножество данных.

Одна из целей шардинга – сделать так, чтобы для приложения кластер из 2, 3, 10 или даже сотен шардов выглядел как одна машина.

Чтобы скрыть эти детали от приложения, мы запускаем перед шардами один или несколько процессов маршрутизации под названием mongos. Процессы mongos хранят «оглавление таблиц», которое сообщает им, в каком шарде какие данные содержатся. Приложения могут подключаться к этому маршрутизатору и отправлять запросы в обычном режиме. Маршрутизатор, зная, какие данные на каком шарде находятся, может пересылать запросы соответствующему шарду (шардам). Если есть ответы на запрос, маршрутизатор собирает их и, если необходимо, объединяет и отправляет обратно в приложение.

Шардинг является наиболее сложным способом настройки MongoDB как с точки зрения разработки, так и с точки зрения эксплуатации. Тут множество компонентов, которые нужно настраивать и мониторить, и данные перемещаются по кластеру автоматически. Следует освоиться с автономными серверами и наборами реплик, прежде чем пытаться развернуть или использовать разделенный кластер. Кроме того, как и в случае с наборами реплик, рекомендуемые средства настройки и развертывания разделенных кластеров – это платформы OpsManager или Atlas.

**Примеры, когда MongoDB подходит для проекта**:

1. Каталог товаров в электронной коммерции.
2. Блоги и системы управления контентом, особенно те, где много контента, в том числе видео и изображений.
3. Аналитика в реальном времени и высокоскоростное журналирование, кэширование данных и кейсов, когда важна высокая масштабируемость системы.
4. Хранение данных датчиков и устройств.
5. Работа с большими данными для машинного обучения и исследований в ритейле и других отраслях.
6. Ведение данных на основе местоположения, то есть геопространственных данных.
7. Социальные сети, новостные форумы и другие похожие сценарии.
8. Слабосвязанные данные без четкой схемы хранения.
9. Стартапы и развертывание новых проектов, где структура данных пока неизвестна.

**Примеры, когда MongoDB лучше не использовать**:

1. Транзакционные системы, приложения, требующие транзакций на уровне базы данных, например, банковские приложения.
2. Проекты, где модель данных определена заранее.

Хранение сильно связанных данных.

**Выбор базы данных**

MySQL — наиболее популярная реляционная база данных, а MongoDB — наиболее популярная нереляционная база данных.

1. При выборе базы данных важно учитывать, какой есть *опыт* и *предпочтения разработчиков*.

Если есть команда, которая долго работала с SQL-базами данных и понимает реляционную алгебру, может быть сложно заставлять разработчиков использовать нереляционные базы данных, такие как MongoDB, где нет даже полноценной транзакции.

И наоборот: если есть команда, которая использует и хорошо знает MongoDB, SQL-язык может быть для неё сложен.

При этом надо рассматривать как оригинальную разработку, так и дальнейшее *сопровождение*, и *администрирование* базы.

2. **MySQL** — это проверенная технология, используется крупными компаниями более 15 лет. Так как используется стандарт SQL, есть возможность достаточно простой миграции на другие SQL-базы данных, если необходимо. Есть возможность транзакций. Поддерживаются сложные запросы, включая аналитику.

**MongoDB** использует гибкий JSON-формат документов. Для некоторых задач это удобнее, чем колонки в SQL-базах данных. Сложные запросы в MongoDB обычно создаются с использованием приложений.

В MongoDB встроена достаточно простая масштабируемость с использованием технологии шардинга.

**Шардинг** обозначает процесс распределения данных по разным серверам.

Размещая подмножество данных на каждой машине, становится возможным хранить больше данных, без необходимости наличия мощных машин – достаточно просто бóльшего количества менее мощных компьютеров.

Шардинг можно использовать и для того, чтобы размещать данные, требующего частого доступа на более производительном оборудовании или разделять наборы данных на основе географии рядом с серверами приложений, с которых к ним чаще всего обращаются.

Вручную шардинг можно осуществить практически с помощью любой СУБД. Такой вариант может хорошо работать, но его становится трудно поддерживать при добавлении или удалении данных.

MongoDB поддерживает ***автоматический*** шардинг, который позволяет упростить администрирование системы.

# 3. MySQL — реляционная база данных. Можно легко отображать связи между таблицами. Нормализуя данные, можно заставлять изменения данных происходить атомарно в одном месте. Данные всегда в таблицах. С одной стороны, это просто, с другой — некоторые структуры данных не всегда удобно размещать в таблицу, нам может быть неудобно с этим работать.

# У MongoDB структура данных основана на *документах*. Данные многих веб-приложений отображать очень просто.

Пример. Пусть надо хранить контакт-листы с телефонами. Есть данные, которые хорошо кладутся в одну реляционную табличку: Фамилия, Имя и т.д. Но телефонов или email-адресов у одного человека может быть несколько. Если их хранить в отдельных таблицах, потом это всё собирать с помощью JOIN, что менее удобно, чем хранить это всё в одной коллекции, где находятся иерархические документы.

4. **MySQL** достаточно гибок в администрировании, у него есть много разных подходов. Множество вариантов порождает сложность.

В **MongoDB** минимизировано администрирование, что приводит к потере гибкости.

5. Под **репликацией** понимается размещение и обслуживание серверов базы данных на нескольких машинах.

Репликация **MySQL** – это процесс, позволяющий легко поддерживать несколько копий данных MySQL путем их автоматического копирования из базы данных master (ведущей) в slave (ведомую). Это упрощает резервное копирование данных, помогает анализировать их без использования главной БД, а также используется в качестве средства масштабирования.

В **MongoDB** реализовано две модели 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.

6. Типичный пример, где используется **MySQL**-решение — это сайт электронной коммерции. Когда идёт вопрос о деньгах, нужны полноценные транзакции. Для чего хорошо подходит реляционная структура, которая была проработана.

**MongoDB** часто задействуется как бэкенд больших онлайн-игр. Потому что важна масштабируемость. Если какая-то игра хорошо выстрелит, её приходится масштабировать значительно больше, чем предполагалось.

## ****Популярные СУБД****

### ****PostgreSQL (www.postgresql.org)****

СУБД имеет большую функциональность и высокую производительность — например, она без проблем может работать с большими данными под высокой нагрузкой.

Язык запросов — SQL, но его можно поменять через расширения на PL/Python, PL/Java и PL/Perl. И ещё одно преимущество PostgreSQL — в ней нет лимита по размеру баз данных и числу записей в таблицах.

Postgres Pro (компании Postgres Professional) — разрабатываемый в России дистрибутив, содержащий усовершенствования, впоследствии, как правило, поступающие в основной проект. Данная версия PostgreSQL сертифицирована рядом крупных российских организаций. В рамках российского импортозамещения рассматривается как замена Oracle (руководитель разработки Postgres Professional Александр Коротков)

### ****MySQL****

Интерфейс программы позволяет работать с таблицами разных форматов. MySQL работает онлайн и вмещает до 50 миллионов элементов. По функциональности она уступает PostgreSQL. При этом её можно интегрировать с другими СУБД.

MySQL использовали для сайтов и интернет-магазинов такие компании, как Twitter, Alibaba, Meta, Wikipedia.

После того как MySQL купила компания Oracle, пользователи стали переживать, что в скором времени база данных может стать платной. Но пока она остаётся бесплатной.

### ****Microsoft SQL Server****

Эта СУБД добавляет автоматизацию задач — например, можно задать скрипт, который будет управлять памятью. Ещё Microsoft SQL Server позволяет удобно хранить сложные структуры данных и быстро искать их.

СУБД совместима с другими программами Microsoft — например, Excel и Access. С ними можно сделать интеграцию и выгружать данные оттуда, а также изменять их онлайн.

В качестве языка запросов Microsoft SQL Server использует язык SQL.

### ****SQLite (****[****sqliteexpert.com****](http://www.sqliteexpert.com/)****)****

Компактная СУБД, которая не использует серверы и другие утилиты. Все данные хранятся на одном устройстве.

На SQLite можно написать простой сайт или приложение, у которого будет ограничен трафик и объём данных. СУБД работает на любых устройствах — смартфонах, компьютерах, ТВ и других, куда можно загрузить библиотеку. Она не нуждается в администрировании, а её язык запросов — C.

### ****MongoDB****

Данные представлены в виде текстовых документов, которые записаны в формате JSON.

СУБД оптимизирована для распределённой работы, но также поддерживает локальное хранение данных.

MongoDB используют такие компании, как Meta, Google, Twitter, Forbes, IBM, а также многие интернет-магазины.

### ****Redis (redis.io)****

NoSQL база данных типа ключ-значение. Redis можно использовать в облаке — полностью готовую к работе и оптимально настроенную. Она легко масштабируется и управляется.

В Redis можно перенести данные из другой базы данных с помощью автоматизированного сервиса.

### ****Oracle Database (www.oracle.com/database/)****

Oracle DB располагается на сервере вместе с базой данных. Поэтому, чтобы работать с ней, нужен специальный интерфейс приложения-клиента. Пользователь управляет пересылкой и получением данных от сервиса.

Oracle DB обеспечивает высокую безопасность и лёгкий доступ для пользователей. Ещё она позволяет снизить нагрузку на клиентские компьютеры. При этом сервер для СУБД должен быть помощнее.

<https://metanit.com/nosql/mongodb/>

[Merrick – Medium](https://merrick-krg.medium.com/)

## Аудит ИС.

С каждым годом увеличивается зависимость организаций от используемых в них информационных систем. Развитие аппаратно-программного обеспечения и существенный его количественный рост всё чаще вызывает необходимость периодической ревизии ИТ-инфраструктуры с целью последующей оптимизации её функционирования и структуры.

По мнению специалистов, одним из важных моментов эффективного управления инфраструктурой ИТ, а также определения и устранения возникающих проблемных ситуаций является её аудит. Для предметных областей, связанных с ИТ, наибольший интерес представляет информационный аудит.

***Информационный аудит*** – это проверка и оценка практики использования ИТ-систем в организации, осуществляемая специализированной независимой организацией.

Выделяют две причины необходимости проведения такого аудита.

***Первая причина*** – информационным технологиям большинства Российских компаний свойственен эволюционный путь создания и дальнейшего их развития. Он характеризуется тем, что информационные системы включаются в инфраструктуру ИТ или модернизируются по мере возникновения необходимости и (или) по мере возможности (в том числе и финансовой). В итоге в ИТ-инфраструктуре формируется сложная (порой разнородная) и поэтому плохо управляемая совокупность программно-технических и системных платформ. Даже если ИС прошла все стадии создания, последующие изменения бизнес-процессов или введение новых приложений могут привести к тому, что параметры её программно-аппаратных платформ перестанут соответствовать требованиям бизнеса.

***Вторая причина*** связана с зависимостью успешности бизнеса от способности управленцев вовремя получать и быстро обрабатывать нужную информацию. Очевидно, что современный руководитель любого звена не в состоянии одновременно быть компетентным в различных областях, обеспечивать получение и обработку разноплановой информации.

Система аудита ИС может быть разделена на три составные части:

     ***внутренний контроль***, осуществляемый организацией, эксплуатирующей ИС;

     ***внешний контроль***, осуществляемый внешней организацией в рамках разделения полномочий организации, эксплуатирующей ИС, и вышестоящего ведомства или организации;

     ***независимый информационный аудит***.

При этом в общем случае выделяют следующие виды аудита ИТ – это аудит:

     процессов управления службой ИТ;

     структуры службы ИТ;

     информационной системы;

     ТЗ и проектной документации на создание ИС;

     систем резервирования данных.

***Аудит процессов управления службой ИТ*** включает ряд специализированных аудитов, среди которых выделим аудиты процесса управления инцидентами, изменениями и конфигурациями, а также управления доступностью и информационной безопасностью. В результате оценки действующего процесса управления на соответствие передовому мировому опыту, представляются предварительные рекомендации по совершенствованию и предложения по структуре проекта совершенствования (реорганизации) процесса управления.

***Аудит информационной системы***используется для определения (оценки степени) соответствия сервисов текущим и планируемым в будущем потребностям бизнеса организации и услугам, предоставляемым информационной системой сотрудникам организации. Результатом такого аудита являются предварительные рекомендации по совершенствованию (модернизации) и предложения по структуре проекта совершенствования (модернизации) ИС.

Анализ обращений пользователей к ИС включает: сбор статистики обращений пользователей, её хранения и анализ этих действий (кто из пользователей, к какой информации, как часто обращался, какие выполнял операции, время выполнения запросов, анализ причин безуспешных, в том числе и аварийных, обращений к ИС).

Анализ эффективности функционирования ИС и развития системы включает: анализ показателей функционирования системы (время обработки, объём памяти, стоимостные показатели), её реорганизации и реструктуризации, изменения состава, развития программных и технических средств.

В процессе проведения информационного аудита предполагается получить ответы на вопросы, связанные с полезностью, доступностью, целостностью и конфиденциальностью аудируемой системы.

***Полезность*** выявляется путём определения как и на сколько используемая система способствует достижению целей, стоящих перед аудируемым лицом и в интересах указанных заинтересованных лиц, а также соответствия издержек создания и эксплуатации системы приобретаемым выгодам.

***Доступность*** определяется установлением возможности системы обеспечивать её использование (доступность) уполномоченными лицами в требуемые моменты с получением необходимой информации вовремя и в адекватных форматах.

***Целостность*** системы устанавливается в процессе выявления возможностей пользователей получать точную, непротиворечивую и своевременную информацию с соблюдением принципов транзакционности обработки данных и журналирования изменений в данных.

***Конфиденциальность*** системы определяется путём установления доступности к информации в системе только её авторизованных пользователей.

При ответе на эти выше вопросы также принимается во внимание, не противоречат ли требования полезности, доступности, целостности и конфиденциальности нормативным правовым актам и соответствующим регламентам использования ИКТ-системы.

Результаты аудита используются почти всеми задачами администрирования ИС.

С целью обеспечения непротиворечивости получаемой информации доступ к подсистеме аудита должен быть ограничен. Причём лицо, ответственное за подсистему аудита не должно иметь административных полномочий по управлению системой и данным, по которым аудит ведётся. Такое разделение позволит существенно повысить уровень безопасности: если человек знает, что его действия протоколируются, то он воздержится от попыток совершения каких-либо манипуляций с информацией во вред компании. По этой же причине он оказывается защищённым от давления со стороны третьих лиц совершить нечто противоправное.

Для определения лиц и организаций, имеющих право осуществлять аудит ИС, рекомендуется разработать и принять *обязательные требования к информационным аудиторам*и определить случаи их применения.

Считается, что проведение информационного аудита должно стать необходимым этапом получения бюджетных средств для создания или развития ИТ-систем.