GeekBrains

Базы данных. Интерактивный курс

Урок 10

NoSQL

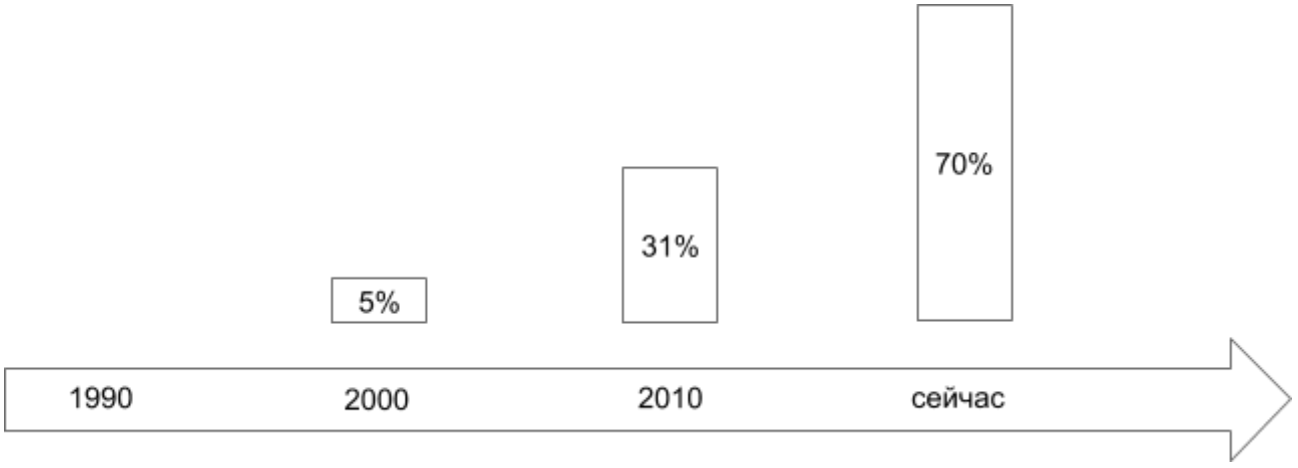
NoSQL-базы данных

Базы данных решают проблему: «я хочу сохранить некие данные и позже снова их просмотреть». Сейчас нет одного правильного решения этой проблемы, но зато есть много разных подходов, более или менее приемлемых, в зависимости от ситуации.

При реализации базы данных обычно приходится выбирать какой-то один из них. Достаточно сложно получить код, который был бы одновременно надежным, масштабируемым и очень быстрым. Попытка получить все сразу в одном программном продукте почти гарантированно ведет к неудачной реализации.

Поэтому часто выбор той или иной базы данных зависит от обстоятельств. Любая база данных предназначена для конкретного стиля использования. Задача разработчика заключается в том, чтобы разобраться, в каких обстоятельствах те или иные программные продукты проявят себя лучше всего.

Так было не всегда, много десятилетий реляционные базы данных считались хорошим компромиссом для решения почти всех задач. Разработка новых технологий и решений велась от существующих движков, количество игроков на рынке баз данных год от года сокращалось, компании поглощались, вытеснялись конкруентами.



Ситуация резко изменилась в начале 2000-х годов, когда интернет из дорогой игрушки превратился в новую электронную инфраструктуру, и многие операции по общению с госорганами, покупками, обучением перешли из офлайн-сферы в онлайн.

Как следствие, доля людей, которые используют интернет, постоянно росла. В ряде стран, в том числе и в России, она достигла 70–80 % от всего населения. Если раньше большие данные были относительно редким явлением, то сейчас они стали массовыми — даже небольшие компании стали сталкиваться с гигантским объемом информации.

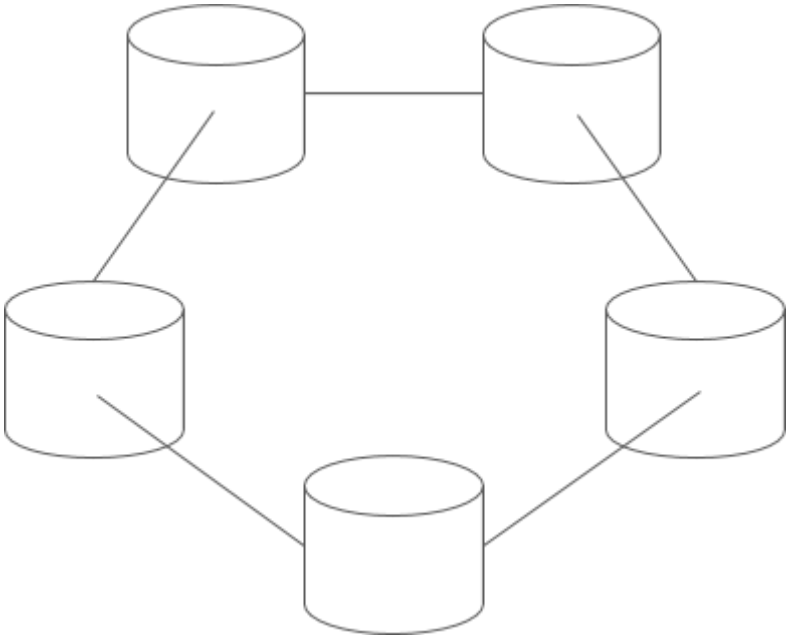
Не важно, что делать с этими потоками: обсчитывать, анализировать и искать корреляции, обучать нейронные сети или даже просто хранить массив данных.

Эти потоки просто появились, причем массово. Проблемы хранения и обработки, которые они стали вызывать, тоже стали массовыми, и возникают не только в больших корпорациях, но и в стартапах. Что это за проблемы?

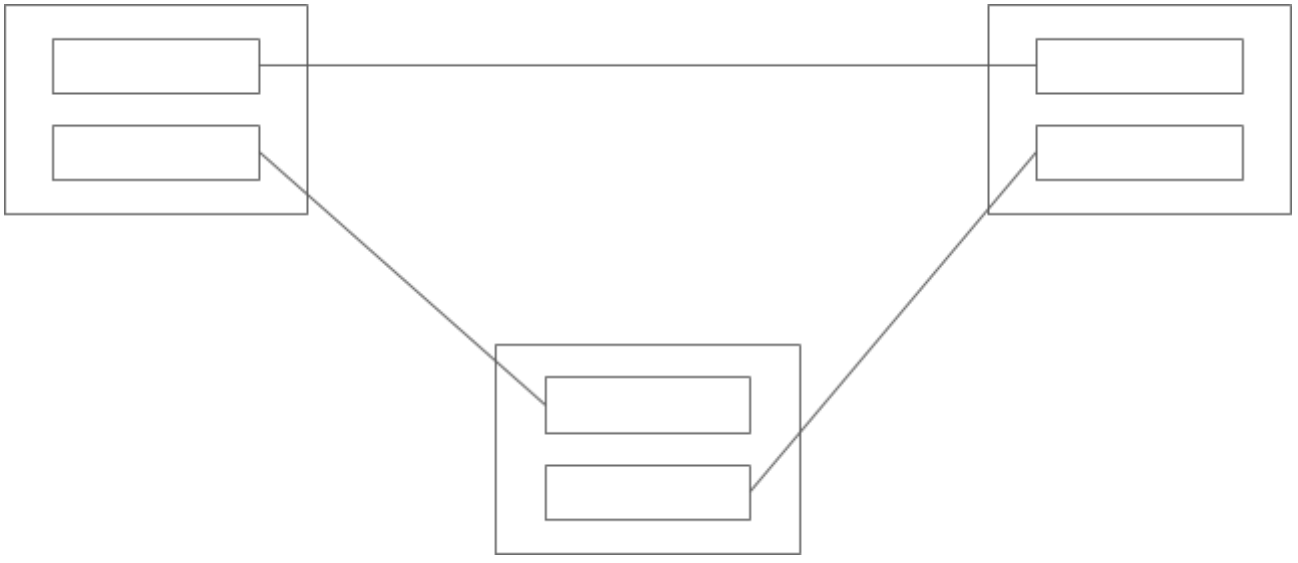
1. Одного, даже мощного, сервера стало не хватать для обсчета этих потоков данных. Базы данных стали распределенными. Увеличение количества компьютеров, участвующих в работе кластера, приводит к возрастанию отказов отдельных компонентов. В результате необходимо менять подход к

отдельным узлам, строить распределенные базы данных таким образом, чтобы потеря нескольких узлов не отражалась на целостности данных.

Распределенные приложения стали все чаще и чаще строиться при помощи кластерных решений, AWS, Kubernetes. В них любая нода может перестать быть доступной в любой момент. Например, нода может оказаться перегруженной или отказать, и контейнер будет выключен и развернут на другой. Т.е., в больших кластерных решениях контейнеры часто становятся недоступными без предупреждения. На это не были рассчитаны старые классические СУБД.

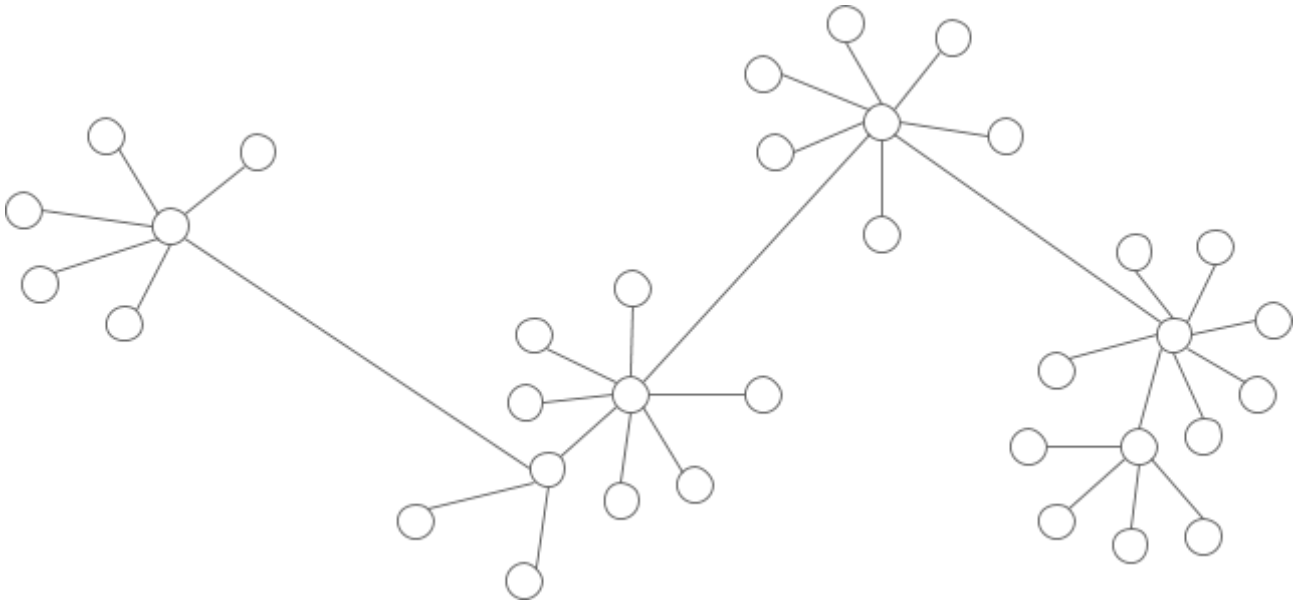


2. Одного сервера стало не хватать для хранения всей базы данных. Ее приходится разбивать на части, шарды, и хранить их на разных серверах. Меняются подходы для построения распределенных приложений.



3. Данных стало очень много, а их формат зачастую неудобен для быстрой обработки в рамках реляционных приложений.

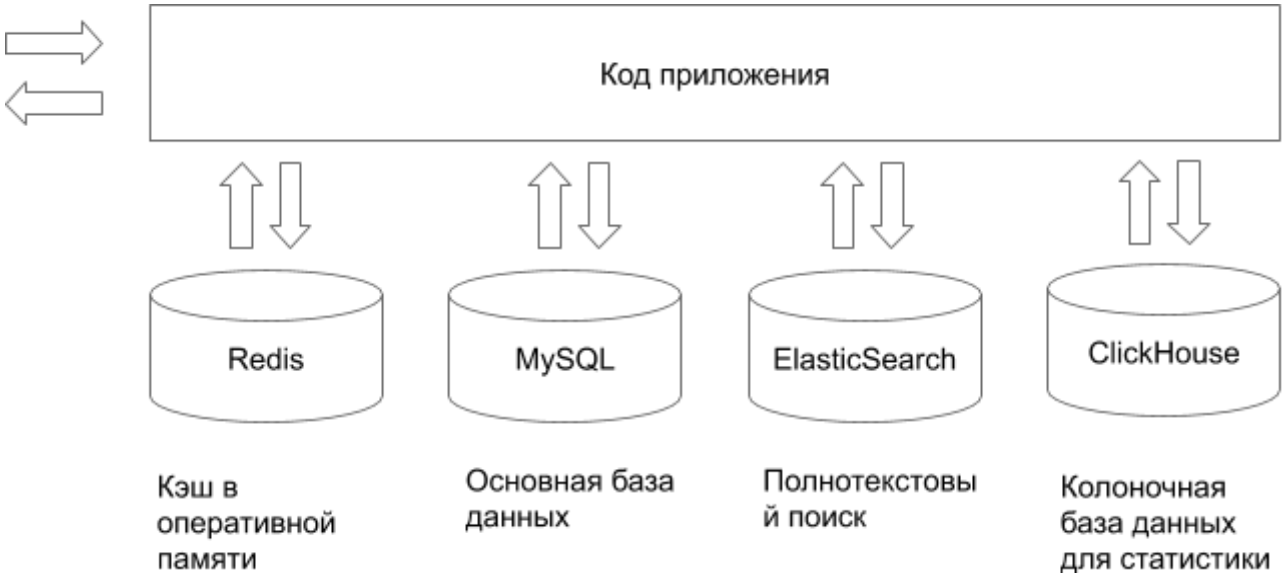
4. Сетевая природа интернета и связей пользователей в социальных сетях потребовала пересмотра реляционной модели, которая зачастую неудобна для организации деревьев или требует слишком объемных вычислительных мощностей. Потребовались другие форматы хранения.



5. Интенсивность потока настолько возросла, что потребовалось масштабировать не только чтение, но и прием данных. Резко увеличилась потребность в очень производительных базах данных.

Эта цепочка событий взорвала уже устоявшийся рынок баз данных. Системы управления базами данных стали меняться, а на рынке стали появляться новые игроки. Сейчас, спустя 40 лет, мы снова находимся в точке перелома, когда базы данных адаптируются к новым правилам игры.

Поэтому, если еще 20 лет назад СУБД от одного поставщика решала весь спектр задач долговременного хранения данных, то сейчас при создании приложения мы зачастую вынуждены пользоваться сразу несколькими СУБД, которые объединяются логикой приложения.



Например, часто требуется интегрировать в базу данных полнотекстовый поиск. Существующего в базах данных полнотекстового поиска иногда недостаточно и может использоваться специализированная база данных. При этом база данных полнотекстового поиска может быть не очень надежной, не поддерживать транзакции. Поэтому во многих приложениях, чтобы удовлетворить

все требования, необходимо комбинировать реляционную СУБД для долговременного хранения и СУБД полнотекстового поиска.

Поддержка транзакций и правил SQL может приводить к невысокой производительности СУБД долговременного хранения. Поэтому данные, которые меняются, не часто могут помещаться в специализированную СУБД для кеширования. Благодаря тому, что такая СУБД полностью расположена в оперативной памяти и спроектирована для максимально быстрой работы, можно добиться существенного увеличения производительности системы.

Обсчет статистики в основной базе данных может занимать длительное время, блокировать большие диапазоны строк в таблице. Поэтому для хранения и обсчета статистики лучше использовать колоночные СУБД. Они не позволяют обновлять и удалять данные, поэтому не предназначены для выполнения роли основной базы данных. Однако они позволяют в режиме реального времени получать агрегационные данные. Так как работа будет производиться в отдельной базе данных, то эти вычисления не будут затрагивать остальные хранилища и влиять на производительность системы.

Новые базы, которые пришли на смену реляционным, стали называть NoSQL-базами данных. Термин стал популярен в 2009 году и стал обозначать нереляционные базы данных. Такие базы данных часто поддерживают SQL-подобный синтаксис, поэтому NoSQL сейчас часто расшифровывают, как «Не только SQL».

Базы данных «ключ-значение». В них вместо таблиц используется ключ, с которым ассоциируется один или несколько атрибутов. Все данные ключа хранятся вместе и часто дублируются. К таким базам данных относятся Riak, Dynamo DB, Memcached и Redis.

Столбцовые базы данных. Если в реляционной модели содержимое строк хранится вместе, то в столбцовых базах данных вместе хранятся данные одного столбца. Каждый столбец таблицы зачастую хранится в отдельном файле.

Это позволяет более эффективно сжимать данные и сортировать столбцы независимо от других данных таблицы. К таким базам данных относятся Cassandra, HBase, Clickhouse.

Документоориентированные базы данных. В них единица хранения — целый документ, часто в формате JSON, XML или YML. Большинство документоориентированных БД не навязывают какой-либо схемы для данных в документах. В документ можно добавлять произвольные ключи и значения. Поэтому документоориентированные БД обычно называют бессхемными (**schemaless**) или неструктурированными. К таким база данных относятся CouchDB и MongoDB.

Графовые базы данных. В них данные представлены в виде графа со множеством вершин и соединяющих их ребер. Основной упор в таких базах данных делается на связях, они часто применяются для построения социальных сетей. К таким базам данных относятся FlockDB, Neo4j, Polyglot.

Объектные базы данных. В них данные хранятся не в виде отношений, состоящих из строк и столбцов, а в виде объектов, что упрощает работу с базой из объектно-ориентированного приложения. В результате с базой данных можно работать без хранимых процедур и средств объектно-реляционного отображения (ORM). К таким базам данных относятся db4o, InterSystems Cache.

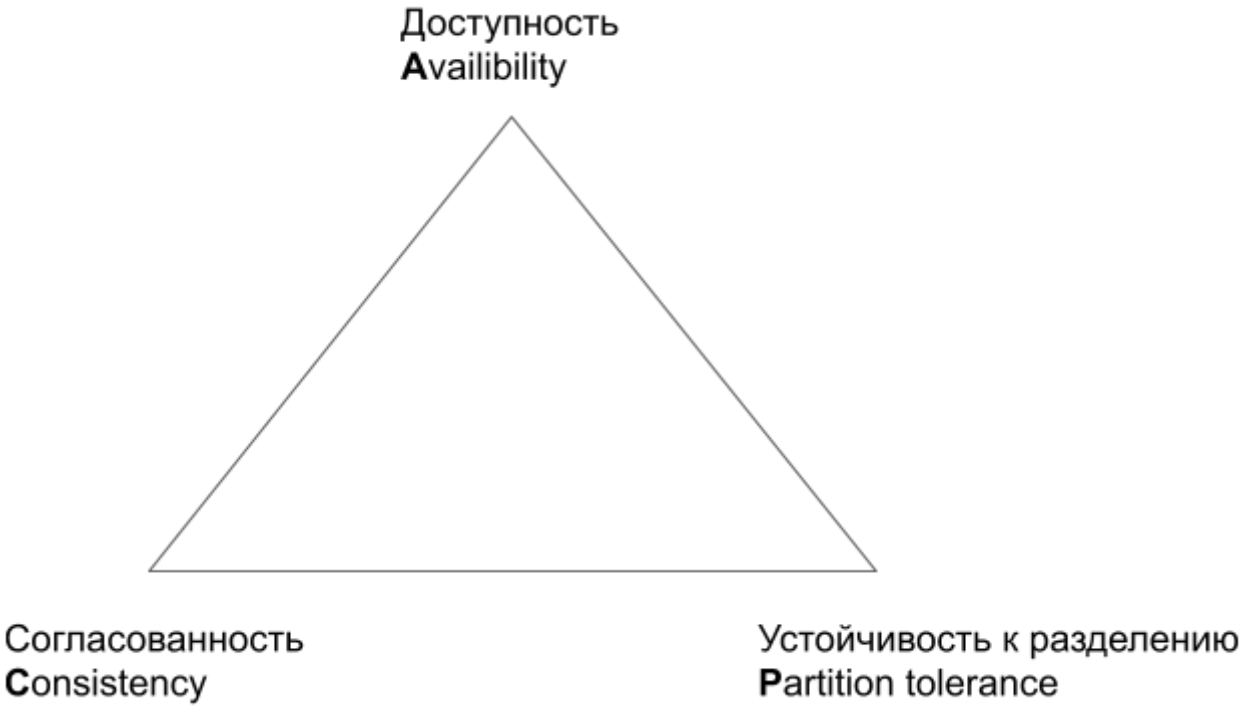
Базы данных полнотекстового поиска. В таких базах данных в качестве основной структурной единицы часто выбирается документ и используется неструктурированная система хранения. Основной упор в таких база[данных делается на поисковый механизм и обеспеченит гибкости настройки релевантности поиска. К базам данных этого типа относятся Solr и ElasticSearch.

Таким образом, в современном мире больших данных и распределенных систем реляционную модель необходимо дополнять новыми решениями.

CAP-теорема

Реляционные СУБД проектировались во времена, когда одна база данных убиралась на одном сервере, максимум в одном дата-центре. В реальности мы часто имеем дело с несколькими серверами, а зачастую и несколькими дата-центрами.

NoSQL-базы данных решают самые разные задачи, но одна из главных — поиск компромисса между согласованностью, доступностью и способностью к разделению. Проблему сформулировал Эрик Брюер в 2000 году, поэтому ее часто называют теоремой Брюера, но более известна она под названием CAP-теоремы: по первым трем буквам Согласованности, Доступности и способности к разделению.



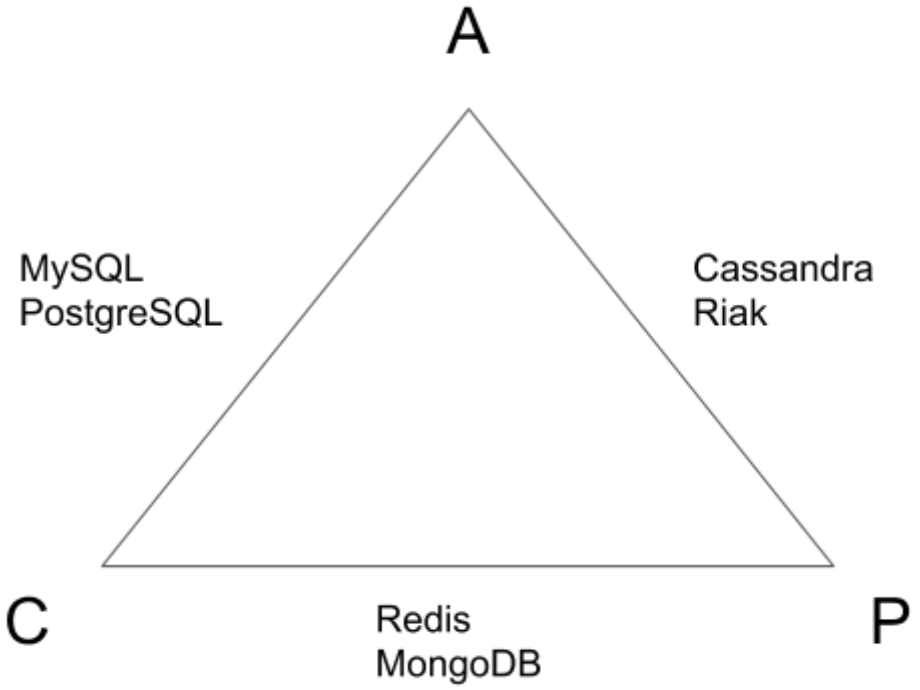
Согласованность означает, что любая операция чтения всегда возвращала самое последнее записанное значение. Например, когда мы настраиваем репликацию и слейв-сервер отстает от мастер-сервера, мы не обеспечиваем согласованность данных. Прочитав старые данные со слейв-сервера, мы можем получить не самый последний результат.

Доступность означает, что клиенты всегда имеют возможность читать и записывать данные.

Способность к разделению означает, что базу данных можно распределить по нескольким машинам, и она продолжит работу даже после отказа сегментов сети.

CAP-теорема утверждает, что в любой системе можно гарантировано обеспечивать выполнение только двух из этих трех требований. Теорема была доказана в 2002 году Сетом Гильбертом и Ненси Линч.

В современном мире у нас приложение почти всегда распределено в сети. При этом сетевые сбои весьма вероятны. Потери пакетов и сетевые задержки могут вызывать временное разделение сети, а значит, из компромиссов остаются либо согласованность, либо доступность.



У каждой базы данных есть своя специализация. Она может удовлетворять двум из трех условий CAP-теоремы. Обеспечение согласованности и доступности означает, что в случае разделения сети система блокируется, т. е., такие базы данных должны работать в одном дата-центре.

На нижней грани согласованности и способности к разделению, как правило, вводится сегментирование или шардирование данных. Однако в случае отказа часть данных может быть недоступна.

Если отдается предпочтение доступности и способности к разделению, могут возвращаться неточные данные. Зато база данных всегда доступна, даже в случае разделения сети. Не всегда базы данных можно однозначно отнести к одной из граней CAP-теоремы, многое зависит от режима, в котором они доступны и от характера сбоев.

Redis

Redis — исключительно быстрый однопоточный сервер для хранения данных в оперативной памяти. Данные в нем хранятся по принципу «ключ-значение».

Очень часто Redis используется для кеширования. Однако он может использоваться и в качестве очереди, и для обмена Pub/Sub-сообщениями.

За счет того, что процессору не приходится переключаться между потоками, при большом количестве соединений достигается огромная производительность.

Причем Redis — это не просто хранилище «ключ-значение», значения могут быть как обычными строками, так и коллекциями (массивами, хешами, множествами), над которыми можно осуществлять операции.

Допускается сохранение данных на жесткий диск, что разрешает проблему холодного старта, когда после запуска должно пройти некоторое время, прежде чем заполнится кеш и страницы будут отдаваться клиенту быстро.

Ключам допускается назначение времени жизни, тем самым можно очищать старые данные в фоновом режиме.

Redis поддерживает транзакции, позволяя объединять несколько команд в блоки, результат выполнения которых сохраняется только в том случае, если все они завершились успешно.

Redis предоставляет механизм репликации. Более того, можно построить Sentinel-кластер из нескольких Redis-серверов, в котором участники будут следить за работоспособностью друг друга. В случае выхода из строя master-сервера, оставшиеся работоспособные ноды могут «проголосовать» и выбрать новый master-сервер. В результате выход из строя одного или нескольких участников кластера не приводит к отказу сервиса.

Благодаря встроенному механизму подписки (Pub/Sub) на основе Redis можно строить очереди заданий. Допустим, при загрузке изображения из него нужно нарезать десяток вариантов с различными размерами. Эта операция может занять длительное время и лучше ее выполнить в фоновом режиме, не задерживая пользователя. В этом случае задание можно поместить в очередь и завершить обслуживание запроса. В очереди, как правило, на отдельном сервере, задание будет рано или поздно выполнено.

Redis допускает создание скриптов на языке программирования Lua. На Lua также разрабатываются расширения для веб-сервера Nginx. В результате можно помещать данные непосредственно в Redis и забирать их из Nginx.

За счет того, что в обслуживании конечных клиентов участвуют исключительно быстрые сервера Nginx и Redis, можно получить исключительную производительность при обслуживании огромного количества одновременных запросов.

В небольшом ролике нам не удастся затронуть все возможности и варианты использования Redis. Однако введение его в проект предоставляет широкие возможности по масштабированию и построению отказоустойчивых сервисов.

Redis — одна из самых популярных NoSQL-баз данных в современных проектах.

Клиент redis-cli

Для доступа к redis-серверу используется клиент **redis-cli**:

```
redis-cli
```

По умолчанию клиент пытается соединиться с локальным сервером по порту 6379. Для выхода из клиента используется команда **QUIT** или **EXIT**.

Помимо **redis-cli**, в составе утилит установленного Redis можно найти **redis-benchmark**, которая измеряет производительность ключевых команд на текущем сервере.

```
$ redis-benchmark -n 100000
```

Как видно, производительность сервера Redis может достигать свыше 100 000 RPS (операций в секунду), даже на запись. Столь впечатляющие результаты достигаются за счет EventLoop-механизма, когда соединения обрабатывает один поток в неблокирующем режиме.

Справочная система

Клиент **redis-cli** содержит удобную справочную систему. Воспользоваться ею можно при помощи команды **HELP**, после которой следует указать название команды.

Например, запросить справочную информацию по команде **PING** можно следующим образом:

```
HELP PING
```

Первое поле указывает название команды, в данном случае **PING**, и список возможных параметров. Поле **summary** кратко описывает назначение команды, а **since** — версию Redis, начиная с которой команда доступна.

Поле **group** указывает группу, к которой относится команда.

Воспользуемся символом **@**: если разместить после него название группы, можно получить список всех команд, входящих в группу.

```
HELP @connection
```

В справочной системе работает функция автодополнения: для получения подсказки достаточно набрать команду **HELP @**, многократно нажимая клавишу **<TAB>**, можно перебирать доступные группы. Ниже приводится список доступных групп:

- **@generic** — команды общего назначения;
- **@string** — команды для работы со строками;
- **@list** — команды для работы со списками;
- **@set** — команды для работы с множествами;
- **@sorted_set** — команды для работы с сортированными множествами;
- **@hash** — команды для работы с хешами;
- **@pubsub** — команды для организации подписчиков;
- **@transactions** — команды транзакционного механизма;
- **@connection** — команды управления соединением с сервером;
- **@server** — команды управления сервером;
- **@scripting** — автоматизация обработки данных;
- **@hyperloglog** — команды для работы с вероятностным алгоритмом подсчета уникальных элементов;
- **@cluster** — команды для обслуживания кластера redis-серверов;
- **@geo** — команды для работы с гео-координатами.

Операции с ключами

Самый простой способ вставить новое значение в базу данных — воспользоваться командой **SET**:

```
SET key 'Hello world!'
```

В качестве первого аргумента команда принимает ключ, а в качестве второго — значение. Извлечь вставленное командой значение можно при помощи команды **GET**:

```
GET key
```

Команда принимает в качестве аргумента ключ и возвращает полученное значение.

Команда **MSET** позволяет вставить за один раз сразу несколько значений, при этом ключи и значения отделяются друг от друга пробелом:

```
MSET fst 1 snd 2 thd 3 fth 4
```

Давайте извлечем вставленные командой **GET** значения:

```
GET fst
GET fth
```

При помощи команды **MGET** можно извлекать сразу несколько значений:

```
MGET fst snd thd fth
```

Так как структура хранимых данных предельно проста, полностью обновить запись «ключ-значение» можно при помощи команды создания нового значения — **SET**.

```
SET "key" "old"
GET key
SET "key" "new"
GET key
```

Однако Redis допускает и более сложные команды обновления значений. Так, с помощью команды **APPEND** можно добавить в конец существующей строки новое значение:

```
SET key "hello"
APPEND key " world!"
GET key
SET count 0
```

При помощи команды **INCR** можно увеличить целочисленное значение на единицу:

```
INCR count
GET count
```

При помощи **INCRBY** можно увеличить целочисленное значение на произвольное целое значение:

```
INCRBY count 5
GET count
```

Команде **INCRBY** можно передавать отрицательное значение. В этом случае будет осуществляться вычитание.

```
INCRBY count -3
GET count
```

Впрочем, для вычитания существуют специальные команды **DECR** и **DECRBY**.

```
SET count 10
DECR count
GET count
DECRBY count 5
GET count
```

Специальная команда **INCRBYFLOAT** позволяет прибавлять и удалять цифры с плавающей точкой:

```
GET count
INCRBYFLOAT count 0.5
GET count
INCRBYFLOAT count -1.3
```

Для удаления пары «ключ-значение» предназначена команда **DEL**, которая принимает в качестве параметра ключ удаляемой пары.

```
DEL key
GET key
```

Для извлечения данных из Redis всегда требуется ключ, поэтому важно знать список всех ключей, которые хранятся в базе данных.

Для получения такого списка используется команда **KEYS**, которая в качестве единственного аргумента принимает шаблон поиска. Если в качестве шаблона указать звездочку *****, будет возвращен список всех доступных ключей:

```
KEYS *
```

Звездочку можно использовать в составе более сложных шаблонов, например в следующем шаблоне извлекаются все ключи, начинающиеся с символа **f**:

```
KEYS f*
```

Для переименования ключа предназначена команда **RENAME**, которая принимает в качестве первого аргумента названия ключа переименовываемой пары, а в качестве второго — новое имя, которое ему назначается:

```
RENAME fst snd
GET snd
GET fst
```

Время жизни ключа

Одна из основных специализаций Redis — быстрый кеш, расположенный в оперативной памяти. В связи с этим особую роль получает актуальность кеша, срок жизни которого обычно не очень велик.

```
SET timer "one minute"
EXPIRE timer 60
```

Для задания срока хранения ключей предназначена команда **EXPIRE**, в качестве первого параметра которой передается имя ключа, а в качестве второго — время его жизни в секундах. Если срок жизни ключа не истек, то команда **EXISTS** возвращает значение 1, в противном случае возвращается 0.

```
TTL timer
```

Чтобы выяснить, сколько секунд осталось до истечения срока жизни ключа, можно воспользоваться командой **TTL**. Ограничение на срок хранения можно отменить, воспользовавшись командой **PERSIST**:

```
PERSIST timer
TTL timer
```

Типы ключей

Redis поддерживает несколько типов данных:

- строки — последовательность символов, заключенных в кавычки;
- числа — целые и с плавающей точкой, позволяющей прибавлять и вычитать значения;
- список — фактически массивы данных;
- хеш — хранит пары «ключ-значение», то есть помимо ключа к самому хешу, каждый элемент, который входит в его состав, также снабжается своим ключом;
- множество — неупорядоченная коллекция уникальных элементов, дублирующие значения в которой отбрасываются автоматически;
- отсортированное множество — точно так же, как в хешах, этот тип коллекции хранит пару ключ-значение, только в качестве ключа выступает числовое значение, задающее порядок следования элементов, что роднит коллекцию со списком. Как и традиционные множества, отсортированные множества сохраняют только уникальные значения.

Коллекционные типы

Вложение коллекций не допускается, то есть коллекции не могут выступать в качестве элементов других коллекций.

В любой момент можно узнать тип значения при помощи специальной команды **TYPE**.

```
SET key "hello world!"
TYPE key
```

Для создания хэша можно воспользоваться командой **HSET**, которая принимает в качестве первого параметра ключ хеша, в качестве второго параметра ключ пары, а в качестве третьего — значение. Ниже создается хеш **admin** с регистрационными данными пользователя.

```
HSET admin login "root"
HSET admin pass "password"
HSET admin register_at "2017-09-01"
```

Создать представленный выше хэш можно при помощи одной команды **HMSET**, которая позволяет задать сразу все пары «ключ-значение»:

```
HMSET admin login "root" pass "password" register_at "2017-09-01"
```

Для чтения элементов хэша можно воспользоваться командой **HGET**:

```
HGET admin login
```

Или извлечь все содержимое хэша при помощи команды **HVALS**:

```
HVALS admin
```

Проверить существование поля с заданным именем можно с помощью команды **HEXISTS**:

```
HEXISTS admin login
HEXISTS admin none
```

Кроме того, в любой момент можно запросить все ключи хэша при помощи команды **HKEYS**:

```
HKEYS admin
```

С помощью команды HGETALL можно извлечь все содержимое хеша, включая ключи и значения:

```
HGETALL admin
```

Выяснить количество элементов в хэше можно с помощью команды **HLEN**:

```
HLEN admin
```

Для вставки значений в множество можно воспользоваться командой **SADD**, первый параметр которой обозначает имя коллекции, а второй — вставляемое значение:

```
SADD email support@softtime.info
SADD email igor@softtime.info
SADD email support@softtime.info
```

Команда **SADD** позволяет вставлять в коллекцию сразу несколько значений:

```
SADD email igorsimdyanov@gmail.com igor@simdyanov.ru igor@softtime.ru
i.simdyanov@rambler-co.ru
```

Сколько бы повторяющихся значений не было вставлено в коллекцию **email**, содержать она будет только уникальные значения, в чем можно убедиться, воспользовавшись командой **SMEMBERS**:

```
SMEMBERS email
```

Выяснить количество элементов в множестве позволяет команда **SCARD**:

```
SCARD email
```

Для удаления элемента из коллекции предназначена команда **SREM**:

```
SREM email igor@softtime.info
```

Для извлечения случайного значения из множества можно воспользоваться командой **SPOP**:

```
SPOP email
```

Сильная сторона множеств — возможность поиска объединения и пересечения нескольких множеств. Для демонстрации этих возможностей создадим дополнительную коллекцию **subscribers**, также содержащую список электронных адресов:

```
SADD subscribers igor@simdyanov.ru igor@softtime.ru
SMEMBERS subscribers
```

Для поиска общих электронных адресов коллекции **email** и **subscribers** можно воспользоваться командой **SINTER**:

```
SINTER email subscribers
```

Для поиска по множеству email-адресов, не входящих во множество **subscribers**, можно воспользоваться командой **SDIFF**:

```
SDIFF email subscribers
```

При помощи команды **SUNION** можно объединить оба множества **email** и **subscribers** в одно (дубликаты автоматически отбрасываются).

```
SUNION subscribers email
```

Базы данных

Как и в СУБД, в Redis имеются базы данных, однако у них нет названия и они нумерованные. По умолчанию утилита **redis-cli** открывает базу данных 0, для того, чтобы переключиться на другую базу данных, например 1, следует воспользоваться командой **SELECT**:

```
localhost:6379> SET key value
localhost:6379> GET key
localhost:6379> SELECT 1
localhost:6379[1]> GET key
localhost:6379[1]> SET key value1
localhost:6379[1]> SELECT 0
localhost:6379> GET key
```

Количество баз данных по умолчанию ограничено 16, однако это значение можно поменять в конфигурационном файле сервера, изменив значение директивы **databases**.

MongoDB

MongoDB — это документоориентированная база данных с открытым кодом. Данные в ней хранятся в виде JSON-документов, языком запроса выступает JavaScript. В результате, например для отображения страницы сайта, нет необходимости собирать данные из нескольких таблиц: JSON-документ уже сразу может содержать всю необходимую информацию для формирования страницы. Это позволяет добиваться значительной производительности.

С другой стороны, структура документа никак не регламентируется. Можно записывать данные в одном формате, а при необходимости — изменить его и начать записывать дополнительные поля или убрать те ключи, которые больше не требуются.

Структуру документа диктует приложение, а не база данных. В этом и преимущество MongoDB — вы можете обрабатывать данные неизвестной заранее структуры, — и ее недостаток — в любой момент структура документа может быть изменена, а в выборке документы одной природы могут иметь разную организацию.

MongoDB поддерживает репликацию и шардирование, поэтому прекрасно масштабируется на чтение.

```
{
  "url": "http://example.com",
  "tags": ["nosql", "mongodb", "databases"],
  "comments": [
    { "user": "Ольга", "content": "Круто!" },
    { "user": "Александр", "content": "Здорово" },
  ]
}
```

JSON — это сокращение от JavaScript Object Notation. Сами JSON-документы состоят из ключей и значений и допускают произвольную глубину вложения, поэтому MongoDB чрезвычайно популярна при разработке на JavaScript и Node.js.

Документ — это набор, состоящий из ключей и значений. Значение может быть представлено простым типом: строкой, числом, датой, а также массивом и другим документом.

На рисунке выше ключ **tags** принимает в качестве значения массив, а **comments** — массив JSON-документов. Такой подход сильно отличается от принятого в реляционной модели, когда мы добиваемся нормализации и под каждый из представленных типов данных (тегов, комментариев, пользователей) создаем отдельную таблицу.

Преимущества документа заключается в том, что вся необходимая информация хранится вместе и доступна сразу, не требуя операции соединения таблиц. В этом же заключается недостаток: данные денормализованы и часто дублируются.

Для работы с MongoDB необходим запущенный сервер MongoDB. Убедиться, что он запущен, можно, поискав среди процессов операционной системы **mongod**.

```
ps aux | grep mongod
```

После этого получить доступ к базе данных можно при помощи консольного клиента **mongo**, который входит в состав дистрибутива:

```
mongo
```

Давайте запросим текущую версию сервера

```
db.version()
```

Запросы к базе данных выглядят как вызовы методов объектов языка JavaScript. Обратите внимание, что, если вы вызываете метод без указания круглых скобок, вы получаете его исходный код:

```
db.version
```

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

Переключить текущую базу данных можно при помощи команды **use**:

```
use shop
```

База данных автоматически появляется, как только в нее будет вставлен хотя бы один документ. Запросить список баз данных можно при помощи команды:

```
show dbs
```

Как видим в этом списке базы данных shop нет, так в ней пока нет ни одного документа. Давайте что-нибудь вставим:

```
db.shop.insert({name: 'Ольга'})
```

Повторно вызываем команду **show dbs** и видим, что база данных **shop** появилась в списке:

```
show dbs
```

Чтобы найти только что вставленное значение, следует воспользоваться методом **find()**:

```
db.shop.find()
{ "_id" : ObjectId("5b49de717aa5dc224acb5ff8"), "name" : "Ольга" }
```

Метод **find()** возвращает вставленный документ вместе с добавленным идентификатором объекта. В любом документе должен быть первичный ключ, который хранится в поле **_id**. Его можно вставлять самостоятельно при условии, что он имеет уникальное значение. Однако, как правило, его не указывают при вставке и MongoDB назначает уникальное значение этому полю самостоятельно.

Давайте вставим еще одно значение:

```
db.shop.insert({name: 'Александр'})
```

Теперь в коллекции должно быть два документа, давайте проверим это, выполнив команду **count**:

```
db.shop.count()
```

Да, так и есть.

```
db.shop.find()
```

Как и раньше, можно запросить все документы при помощи метода **find()**. Однако **find()** может принимать селектор запроса, JSON-объект, с которым сравниваются все документы коллекции:

```
db.shop.find({name: 'Ольга'})
```

Если соответствие находится, запись попадает в результирующий ответ, если же документ не соответствует шаблону — он отбрасывается.

Для обновления используется метод **update**, которому задается минимум два аргумента. Первый определяет, какие документы обновлять, второй — как следует модифицировать отобранные документы.

Давайте обновим документ пользователя Ольги, добавив ей электронный адрес:

```
db.shop.update({name: 'Ольга'}, {$set: { email: 'olga@gmail.com' }})
```

Обратите внимание на ключ **$set**, который начинается с символа доллара: фактически это обозначение команды по вставке нового значения.

Если мы заходим удалить только что вставленное значение, мы можем воспользоваться ключом **unset**:

```
db.shop.update({name: 'Ольга'}, {$unset: { email: '' }})
db.shop.find()
```

Вставлять можно не только простые значения, но и коллекции:

```
db.shop.update({name: 'Ольга'}, {$set: { contacts: { email: ['olga@gmail.com',
'olga@mail.ru'], skype: 'olgashop' }}})
db.shop.update({name: 'Александр'}, {$set: { contacts: { email:
['alex@gmail.com'], skype: 'alexander' }}})
db.shop.find()
```

Если мы захотим искать по вложенным документам, используем специальную вложенную нотацию. Давайте найдем пользователей со Skype **alexander**:

```
db.shop.find({ 'contacts.skype': 'alexander' })
```

Как видим, мы можем обратиться к ключу вложенной структуры. Точка между **contacts** и **skype** означает, что нужно найти ключ **contacts**, который указывает на вложенный объект **skype**, а затем сравнивать значение этого вложенного ключа с указанным в запросе.

Допустим мы ходим добавить Александру новый электронный адрес alex@mail.ru. Можно было бы снова воспользоваться оператором **$set**, но это означало бы, что надо переписать и отправить на сервер весь массив электронных адресов. Так как нам нужно добавить в массив только один элемент, мы можем использовать оператор **$push**:

```
db.shop.update({name: 'Александр'}, {$push: { 'contacts.email': 'alex@mail.ru'
}})
```

Для удаления данных из коллекции предназначен метод **remove()**:

```
db.shop.remove({name: 'Ольга'})
db.shop.find()
```

Если мы не будем задавать методу **remove** селектор, то получим сообщение об ошибке:

```
db.shop.remove()
```

Для удаления всей коллекции можно воспользоваться методом **drop()**:

```
db.shop.drop()
db.shop.find()
show dbs
```

Обратите внимание, так как в коллекции **shop** были удалены все записи, такая коллекция больше не выводится в отчете команды **show dbs**.

Документы могут быть довольно объемные, поэтому в консоли с ними может быть неудобно работать. В случае MongoDB следует обратить внимание на клиенты с графическим интерфейсом, например Robomongo.

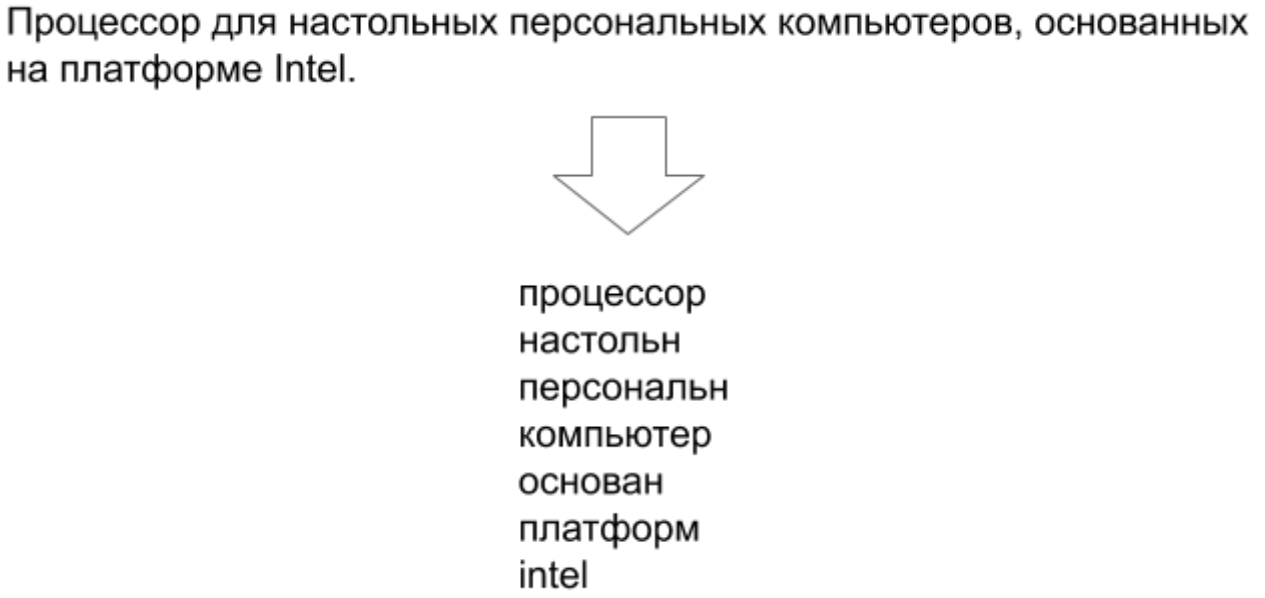
СУБД полнотекстового поиска ElasticSearch

Пользователям все чаще приходится работать с большими объемами информации. Поэтому поле поиска — зачастую первое место, куда обращаются пользователи, чтобы приступить к изучению проблемы и найти ответы.

Поисковые машины, такие как Google или Яндекс, являются точками входа почти для всех пользователей Интернет. Пользователи ожидают от приложений такого же быстрого и точного ответа, как в больших поисковых системах. Поэтому релевантность поиска в современных приложениях приобретает большую важность.

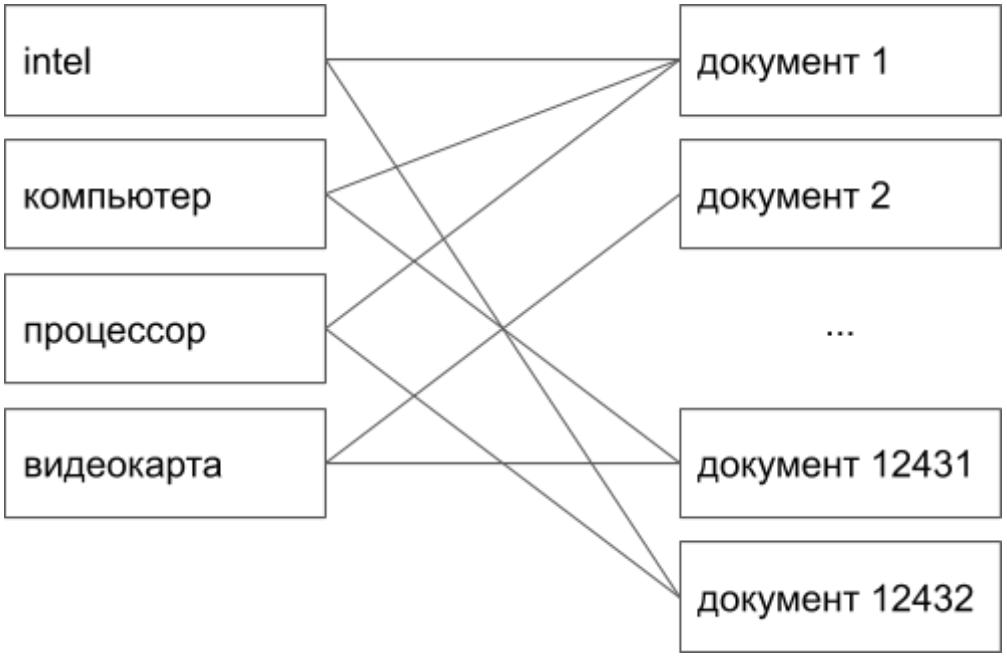
Полнотекстовый поиск

Базы данных полнотекстового поиска, как правило, документоориентированные, т. е., документ — это минимальная единица содержимого для хранения, поиска и возврата. Именно документ — главная цель поиска.



Поля документа подвергаются анализу, в ходе которого текстовые поля преобразуются в лексемы или токены. Как правило, они переводятся в нижний регистр, суффиксы и приставки отбрасываются, как и слишком часто встречающиеся слова, например предлоги. Часто встречающиеся слова нарушают релевантность поиска. Даже если слово не общеупотребительное, но слишком часто встречается в документах базы данных, ему назначается низкий вес, чтобы снизить его влияние на результат запроса. Если слово встречается слишком часто, например, более чем в 50 % документов — оно отбрасывается.

Поэтому чем больше ваша база данных, тем лучше работает полнотекстовый поиск. Если у вас в базе данных всего два документа, слова буду встречаться как раз в 50 % документов и полнотекстовый поиск будет работать из рук вон плохо.

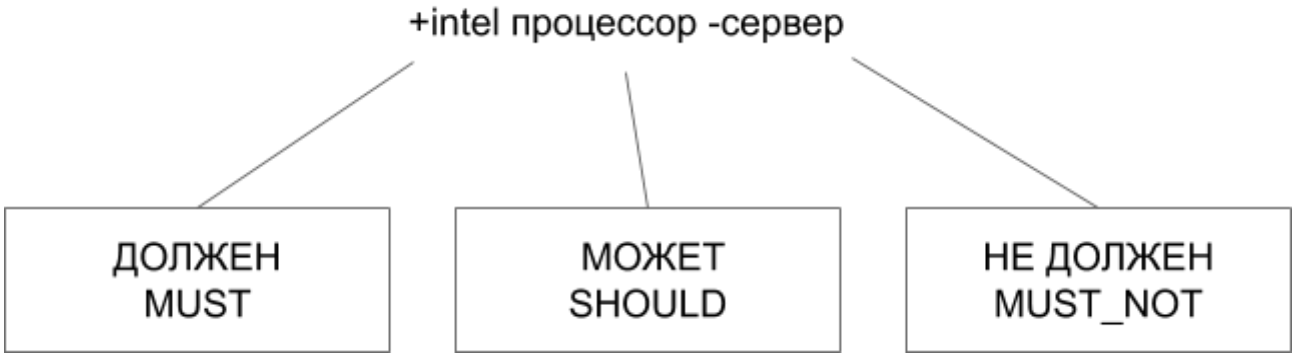


По завершению анализа документы индексируются. Лексемы, извлеченные на этапе анализа, сохраняются отдельно и используются в дальнейшем для организации поиска по документу, также сохраняется исходный непреобразованный текст, чтобы иметь возможность вернуть его

пользователю в результатах поиска. Лексемы и документы связываются обратным индексом, в котором по термину всегда можно найти документ.

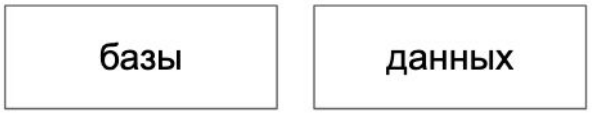
Когда вы ищете одно слово, все довольно просто: достаточно найти в обратном индексе слово и вернуть все документы, на которые ссылается элемент индекса. Если у вас несколько слов, вам потребуется использовать логический поиск, объединяя слова логическим И или ИЛИ.

В случае И, все слова обязательно должны присутствовать в документе. В случае ИЛИ в документе должно присутствовать одно из слов. Может применяться логическое НЕ, если в результат не должны попадать документы с выбранными лексемами.



В ElasticSearch вместо И, ИЛИ и НЕ используются похожие операции — МОЖЕТ, ДОЛЖЕН и НЕ ДОЛЖЕН.

- ДОЛЖЕН — требует обязательного присутствия совпадения с термином внутри документа, иначе документ не будет считаться соответствующим запросу;
- МОЖЕТ указывает, что документ может содержать или не содержать совпадения с искомым термином, но при наличии совпадения документ получает более высокий ранг;
- НЕ_ДОЛЖЕН требует, чтобы содержащий совпадения с термином документ не считался соответствующим запросу, даже если в нем имеются совпадения с операторами ДОЛЖЕН и НЕ_ДОЛЖЕН.



базы данных играют ключевую роль для построения современного приложения

данные о военных базах довольно трудно найти в открытых источниках

базы данных играю ключевую роль для построения современного приложения

данные о военных базах довольно трудно найти в открытых источниках

Относительное расположение слов в запросе часто имеет большое значение. Например, поиск по фразе «База данных» должен возвращать документы с описанием SQL- или NoSQL-баз данных. Если механизм поиска не будет учитывать взаимное расположение терминов, вместо документов, посвященных программному обеспечению, он вернет множество документов, посвященных статистическим данным, военным базам и т. д. Это не совсем то, то ожидает пользователь, поэтому полнотекстовый поиск не только предоставляет возможность поиска по фразе, но и учитывает расстояние между словами — чем оно меньше, тем документ считается более релевантным.

Процессор Intel Core i3-8100	0.999
Процессор Intel Core i5-7400	0.956
Процессор AMD FX-8320E	0.843
AMD FX-8320	0.812

Чтобы наиболее подходящие документы оказывались в начале поиска, документы сортируются по релевантности. Релевантность, в свою очередь, определяется функцией ранжирования. Для каждого документа функция ранжирования вычисляет оценку, определяющую, насколько полно документ соответствует запросу.

Чем чаще термин встречается в документе, тем более полно он соответствует запросу. Однако чем чаще термин встречается во всех документах, тем меньше у него вес в оценке.

СУБД ElasticSearch

Одна из самых популярных баз данных для полнотекстового поиска — ElasticSearch, которая использует Java-библиотеку Lucene и считается промышленным сервером.

Сервер поддерживает кластеризацию и шардирование, обеспечивает высокую производительность и горизонтальное масштабирование. Для доступа к нему используется REST-подобный интерфейс, т.

е., в основе запросов лежит протокол HTTP, поэтому обращаться к серверу можно непосредственно из браузера.

Чтобы воспользоваться ElasticSearch, нам потребуется сервер, который должен быть запущен на вашем компьютере:

```
ps aux | grep elastic
```

По умолчанию сервер ожидает запросов по порту 9200. Проверить работоспособность сервера мы можем, отправив HTTP-запрос на по адресу **localhost:9200**.

```
curl 'http://localhost:9200'
```

Итак, мы получаем информацию о сервере и его версии. Здесь мы воспользовались утилитой **curl** для отправки HTTP-запросов из консоли. Для этого можно воспользоваться и браузером. Когда мы обращаемся к ресурсу, используя адресную строку, мы отправляем ему HTTP-запрос методом **GET**.

Однако многие запросы к ElasticSearch требуют не только GET-запросов: например, заполнение индекса идет с использованием HTTP-запросов типа **POST**. Поэтому большую часть запросов мы будем осуществлять через консоль с использованием утилиты **curl**.

Для начала давайте вставим что-нибудь в базу данных ElasticSearch:

```
curl -H 'Content-Type: application/json' -X PUT
'http://localhost:9200/shop/products/1?pretty' -d'
{
  "name" : "Intel Core i5-7400",
  "description" : "Процессор для настольных персональных компьютеров, основанных
на платформе Intel.",
  "price" : "12700.0",
  "tags" : [
      "комплектующие",
      "процессоры",
      "Intel"
  ],
  "created_at" : "2018-10-10T20:35:12+00:00"
}'
```

Итак, мы вставили первый документ в индекс **shop** и тип **productions**. Если проводить аналогию с реляционными базами данных: индекс соответствует понятию базы данных, а тип — таблице.

```
curl 'http://localhost:9200/shop/products/1?pretty'
```

Чтобы извлечь содержимое документа, мы можем воспользоваться следующим GET-запросом. Мы получаем JSON-документ, в поле **_source** которого мы видим оригинальный документ.

Все ключи, начинающиеся с символа подчеркивания, относятся к служебным. Если нам не нужна мета-информация, мы можем запросить конкретное поле, например, **_source**:

```
curl 'http://localhost:9200/shop/products/1/_source?pretty'
```

Можем детализировать запрос еще больше и запросить конкретное поле документа, например, **description**:

```
curl 'http://localhost:9200/shop/products/1/?_source=description&pretty'
```

Давайте вставим еще несколько записей, соответствующих процессорам:

```
curl -H 'Content-Type: application/json' -X PUT
'http://localhost:9200/shop/products/2' -d'
{
  "name" : "Intel Core i3-8100",
  "description" : "Процессор для настольных персональных компьютеров, основанных
на платформе Intel.",
  "price" : "7890.00",
  "tags" : [
      "комплектующие",
      "процессоры",
      "Intel"
  ],
  "created_at" : "2018-10-10T20:40:23+00:00"
}'

curl -H 'Content-Type: application/json' -X PUT
'http://localhost:9200/shop/products/3' -d'
{
  "name" : "AMD FX-8320E",
  "description" : "Процессор AMD.",
  "price" : "4780.00",
  "tags" : [
      "комплектующие",
      "процессоры",
      "AMD"
  ],
  "created_at" : "2018-10-10T20:42:06+00:00"
}'
```

И материнским платам:

```
curl -H 'Content-Type: application/json' -X PUT
'http://localhost:9200/shop/products/4' -d'
{
  "name" : "ASUS ROG MAXIMUS X HERO",
  "description" : "Материнская плата Z370, Socket 1151-V2, DDR4, ATX",
  "price" : "19310.00",
  "tags" : [
      "комплектующие",
      "материнские платы",
      "ASUS"
  ],
  "created_at" : "2018-10-10T20:40:23+00:00"
```

```
}'

curl -H 'Content-Type: application/json' -X PUT
'http://localhost:9200/shop/products/5' -d'
{
  "name" : "Gigabyte H310M S2H",
  "description" : "Материнская плата H310, Socket 1151-V2, DDR4, mATX",
  "price" : "4790.00",
  "tags" : [
      "комплектующие",
      "материнские платы",
      "Gigabyte"
  ],
  "created_at" : "2018-10-10T20:44:36+00:00"
}'
```

Для извлечения документов можно использовать search-запрос. При помощи параметра **size** мы можем ограничить выборку только двумя элементами:

```
curl 'http://localhost:9200/shop/products/_search?pretty&size=2'
```

Можем отправлять и более сложные запросы, например ориентируясь на тэги. Так, мы можем извлечь все документы, у которых есть тэг «процессоры»:

```
curl -H 'Content-Type: application/json'
'http://localhost:9200/shop/products/_search?pretty' -d'
{
  "query": {
      "bool": {
      "filter": {
      "term": {
      "tags": "процессоры"
      }
      }
      }
  }
}'
```

Однако стоит нам ошибиться в названии тэга и мы получим нулевой результат. Давайте заменим «процессоры» на «процессор»:

```
curl -H 'Content-Type: application/json'
'http://localhost:9200/shop/products/_search?pretty' -d'
{
  "query": {
      "bool": {
      "filter": {
      "term": {
      "tags": "процессор"
```

```
        }
        }
        }
    }
}'
```

Ничего не найдено.

Когда мы используем ключевое слово **term** — мы ищем точное совпадение, для включения механизма полнотекстового поиска нам необходимо задействовать match-поиск. Каждый тип имеет свою схему — **mapping**, которая генерируется автоматически при индексации документа. Посмотреть текущий маппинг можно при помощи запроса **_mapping**.

```
curl 'http://localhost:9200/shop/products/_mapping?pretty'
```

Сейчас он не поддерживает русский язык, давайте включим такую поддержку. Для начала удалим существующий индекс:

```
curl -X DELETE 'http://localhost:9200/shop?pretty'
```

И явно создадим новый **mapping** с поддержкой русского языка:

```
curl  -H 'Content-Type: application/json' -X PUT
'http://localhost:9200/shop?pretty' -d'
{
  "settings": {
      "analysis": {
      "filter": {
      "ru_stop": {
      "type": "stop",
      "stopwords": "_russian_"
      },
      "ru_stemmer": {
      "type": "stemmer",
      "language": "russian"
      }
      },
      "analyzer": {
      "default": {
      "char_filter": [
            "html_strip"
      ],
      "tokenizer": "standard",
      "filter": [
            "lowercase",
            "ru_stop",
            "ru_stemmer"
      ]
      }
      }
```

```
        }
    },
    "mappings": {
        "products": {
        "properties": {
        "description": {
        "type": "text"
        },
        "name": {
        "type": "text"
        },
        "price": {
        "type": "double"
        },
        "tags": {
        "type": "text"
        },
        "created_at": {
        "type": "date"
        }
        }
        }
    }
}'
```

Давайте воспользуемся полнотекстовым поиском по всему документу:

```
curl -H 'Content-Type: application/json'
'http://localhost:9200/shop/products/_search?pretty' -d'
{
  "query": {
      "query_string": {
      "query": "процессор"
      }
  }
}'
```

Итак, у нас были найдены процессоры Intel и AMD.

Возможности ElasticSearch велики и допускают настройку всех аспектов полнотекстового поиска. Они заведомо превосходят аналогичные решения в реляционных СУБД. Поэтому ElasticSearch будет довольно часто встречаться вам в современных проектах.

Колоночная СУБД ClickHouse

С момента появления базы данных традиционно используются для обработки коммерческих операций или коммерческих транзакций. Например, продажи товара или услуги, размещения заказа у поставщика, выплаты зарплаты и т. п. Хотя базы данных теперь используются в том числе и областях, в которых не происходит перехода денег из рук в руки, термин «транзакция» до сих пор используется для обозначения единой операции чтения и записи.

Сейчас БД применяются для множества самых разных областей: комментариев в сообщениях блогов, действий в играх, контактов в адресной книге и т. д. Порядок работы с ними во многом остался сходным с обработкой коммерческих операций, коммерческих транзакций.

Приложение обычно ищет небольшое количество записей по какому-либо ключу. На основе вводимых пользователем данных вставляются или обновляются записи. Такие операции принято называть «обработка транзакций в реальном времени» (OLTP).

Однако БД все шире используются для аналитической обработки данных. Поведение таких запросов совершенно другое. Обычно аналитический запрос должен просматривать огромное количество записей, в каждой из которой читается лишь несколько столбцов. Результатом обычно являются какие-то агрегированные показатели (например, количество, сумма или среднее значение):

- Какова общая выручка за октябрь?
- Каков средний чек покупателя за прошлый год?
- Какую марку товара чаще покупают с другой маркой товара?

Такие операции принято называть «аналитической обработкой данных в реальном времени» OLAP

Сначала для обработки обоих типов запросов использовались одни и те же базы данных, тем более SQL предоставляет гибкие возможности для составления агрегационных запросов. Тем не менее, в конце 1980-х — начале 1990-х годов возникла такая тенденция: компании стали разделять базы данных для транзакций и для анализа. Последние стали называть складами данных.

Обычно от баз данных, обрабатывающих транзакции, ожидают высокой доступности и скорости обработки транзакций, поэтому их стали оберегать от тяжелых аналитических запросов, которые зачастую могут выполняться часами, блокируя широкий диапазон данных.

Склад данных представляет собой отдельную БД, которую аналитики могут опрашивать так, как им заблагорассудится, не влияя при этом на транзакционную базу данных. Как правило, он содержит копию основной базы данных, зачастую в режиме read only. Чем больше компания и чем больше у нее данных, тем более вероятно, что у нее появится склад данных, в который может стекаться информация из одной, а зачастую и нескольких транзакционных баз данных.

Изначально для транзакционной базы данных и склада данных использовались одни и те же реляционные решения. Однако постепенно внутренее устройство баз данных стало различаться, так как запросы, на которые они ориентированы, совершенно разные.

Поэтому в настоящий момент эти две роли выделятся в специализированные базы данных. Как правило, в качестве складов данных выступают колоночные базы данных.

Коммерческие решения:

- Vertica.
- ParAccel.
- Teradata.
- SAP HANA.

Свободные склады данных:

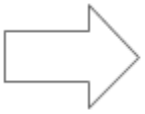
- HBase.
- Cassandra.
- Hypertable.
- Amazon SimpleDB.
- ClickHouse.

В реляционных таблицах можно хранить большие объемы данных, они прекрасно обрабатывают миллионы строк и гигабайтные таблицы. Однако для гигантских таблиц с триллионами строк и петабайт данных эффективное хранение и выполнение запросов становится непростой задачей.

В большинстве реляционных баз данные располагаются построчно. Все значения из одной строки таблицы хранятся рядом друг с другом.

Документоориентированные БД устроены аналогично: весь документ обычно хранится в виде непрерывной последовательности байт. Чтобы выполнить аналитический запрос, часто нужны только несколько столбцов таблицы. Однако построчно-организованная СУБД будет извлекать все столбцы с диска в оперативную память, те столбцы которые не нужны, будут отброшены, усилия и память, потраченные на их извлечение, пропадут. Тут только два столбца, но когда их в таблице сотни, это существенно отражается на производительности. Идея столбцовых хранилищ проста: нужно хранить рядом значения не из одной строки, а из одного столбца.

id	name	birthday_at
1	Ольга	1993-10-12
2	Иван	NULL
3	Сергей	NULL



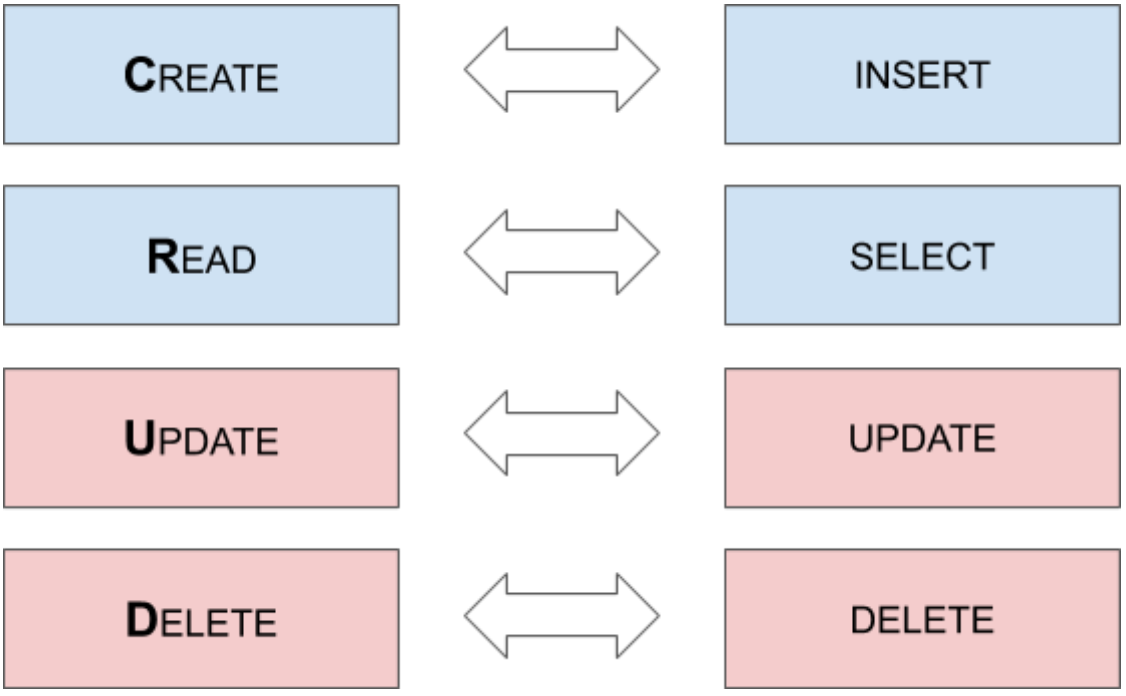
name
Ольга
Иван
Сергей

birthday_at
1993-10-12

Если каждый столбец хранится в отдельном файле, то запросу требуется только прочитать и выполнить синтаксический разбор необходимых столбцов, что может сэкономить массу усилий.

За счет раздельного хранения столбцов, данные прекрасно подвергаются сжатию. Зачастую количество различных значений в столбце невелико по сравнению с числом строк. У интернет-магазина могут быть миллионы торговых операций, но только сотни различных товаров.

Большая часть нагрузки в больших аналитических запросах относится к операциям чтения. Хранение сжатых данных в виде отдельных столбцов ускоряет выполнения этих запросов. Однако операции записи усложняются.



Поэтому в столбцовых базах данных операции обновления и удаления данных зачастую не предусмотрены. Мы можем выполнять только вставку и чтение.

В качестве колоночной СУБД мы рассмотрим Clickhouse, который был разработан компанией Яндекс для использования в проекте «Яндекс.Метрика».

Clickhouse — классическая колоночная СУБД, поддерживающая сжатие данных. Многие колоночные СУБД полностью располагаются в оперативной памяти. Clickhouse позволяет держать данные на жестком диске, что более разумно для склада данных.

Запросы выполняются в параллельном режиме на многоядерных процессорах. Пользователям предоставляется язык запросов, следующий традициям SQL. Конечно, речи о поддержке стандарта SQL не идет, все-таки мы имеем дело не с реляционной базой данных. Однако поддерживаются соединения и подзапросы. Допускается индексирование данных и приближенные вычисления. Clickhouse подавляет репликацию и шардирование.

База данных разработана российской компанией Яндекс, и на официальном сайте clickhouse.yandex есть детальная документация на русском языке.

Для соединения с сервером ClickHouse используется клиент **clickhouse-client**:

```
SELECT 1
```

ClickHouse поддерживает разделение данных по отдельным базам данных. Получить текущий список баз данных можно при помощи команды **SHOW DATABASES**.

```
SHOW DATABASES;
```

Создать базу данных можно при помощи команды:

```
CREATE DATABASE shop;
```

Давайте убедимся, что база данных успешно создана:

```
SHOW DATABASES;
```

Мы видим новую базу данных **shop**, выбрать ее в качестве текущей можно так же, как в случае с MySQL — командой **USE**:

```
USE shop
```

Давайте создадим таблицу с посещениями **visits** с посещениями по дням:

```
CREATE TABLE `visits` (
  `VisitDate` Date,
  `Hits` UInt32
) ENGINE = MergeTree(VisitDate, (VisitDate), 8192);
```

Синтаксис ClickHouse очень похож на SQL, для создания таблицы мы используем команду **CREATE TABLE**. Название таблицы и столбцов заключаем в обратные кавычки. Для задания разных типов

целочисленного столбца мы используем тип **Int**, суффиксы 8, 16, 32 и 64. по количеству бит в числе, префикс **U** сообщает, что число должно быть беззнаковым. Мы можем использовать и календарные типы, например **Date** для столбца **VisitDate**. Точку с запятой в конце запроса можно не указывать.

Давайте вставим данные за октябрь 2018 года:

```
INSERT INTO visits VALUES
('2018-10-01', 45324),
('2018-10-02', 72241),
('2018-10-03', 69572),
('2018-10-04', 93242),
('2018-10-05', 92201),
('2018-10-06', 12579),
('2018-10-07', 17242),
('2018-10-08', 45298),
('2018-10-09', 67932),
('2018-10-10', 74590),
('2018-10-11', 67275),
('2018-10-12', 45475),
('2018-10-13', 10792),
('2018-10-14', 11243),
('2018-10-15', 78056),
('2018-10-16', 65345),
('2018-10-17', 94335),
('2018-10-18', 89345),
('2018-10-19', 52532),
('2018-10-20', 11437),
('2018-10-21', 13673),
('2018-10-22', 78234),
('2018-10-23', 67233),
('2018-10-24', 92345),
('2018-10-25', 82564),
('2018-10-26', 60234),
('2018-10-27', 14234),
('2018-10-28', 15865),
('2018-10-29', 72784),
('2018-10-30', 76335),
('2018-10-31', 89483);
```

Для подсчета количества записей в таблице можно воспользоваться функцией **COUNT()**:

```
SELECT COUNT() FROM visits;
```

Для извлечения первых 5 строк таблицы можно использовать следующий запрос:

```
SELECT * FROM visits LIMIT 5;
```

Как видим, синтаксис SELECT-запроса полностью эквивалентен SQL

```
SELECT * FROM visits LIMIT 2\G
```

Поддерживается даже вертикальный режим вывода при помощи модификатора **\G**, который был заимствован из MySQL.

Можно использовать и более сложные запросы. Например, давайте подсчитаем количество посещений по неделям, для этого воспользуемся функцией **toRelativeWeekNum** для преобразования даты **VisitDate** в номер недели:

```
SELECT toRelativeWeekNum(VisitDate) AS week, Hits FROM visits LIMIT 2;
```

Обратите внимание, так же как и в SQL мы можем использовать ключевое слово **AS** для назначения псевдонима столбца. Теперь мы можем сгруппировать данные по полю **week** и воспользоваться функцией **SUM** для подсчета количества посещений в неделю.

```
SELECT toRelativeWeekNum(VisitDate) AS week, sum(Hits) AS hits
FROM visits
GROUP BY week;
```

Давайте преобразуем полученные результаты в диаграмму при помощи функции **bar**:

```
SELECT
  toRelativeWeekNum(VisitDate) AS week,
  sum(Hits) AS hits,
  bar(hits, 0, 500000, 20) AS bar
FROM
  visits
GROUP BY
  week;
```

Функция получает в качестве первого аргумента текущее числовое значение. В данном случае это поле **hits**, полученное при помощи функции **sum** как сумма посещений за неделю. Два следующих аргумента задают минимальное и максимальное значения диаграммы, а последнее — ширину столбца.

Посмотреть список текущих таблиц можно при помощи запроса **SHOW TABLES**:

```
SHOW TABLES;
```

Как видим, у нас пока только одна единственная таблица **visits**. Удалить таблицу можно при помощи команды **DROP TABLE**.

```
DROP TABLE visits;
```

Используемые источники

1. Клеппман М. Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб.: Питер, 2018. — 640 с.: ил.
2. Редмонд Эрик , Уилсон Джим Р. Семь баз данных за семь недель. Введение в современные базы данных и идеологию NoSQL. — М: ДМК Пресс — 384с.
3. Фаулер, Мартин, Садаладж, Прамодкумар Дж. NoSQL: новая методология разработки нереляционных баз данных. — Пер. с англ. — М.: ООО "И.Д. Вильямс", 2013. — 192 с.
4. Робинсон Ян, Вебер Джим, Эифрем Эмиль. Графовые базы данных: новые возможности для работы со связанными данными. — 2-е изд. — М.: ДМК Пресс, 2016. — 256 с.
5. Карпентер Д., Хьюитт Э. Cassandra. Полное руководство. 2-е изд. — М.: ДМК Пресс, 2017. — 400 с.
6. Бэнкер Кайл. MongoDB в действии. — М.: ДМК Пресс, 2017. — 394с.
7. https://redis.io/documentation
8. https://clickhouse.yandex/docs/ru/