



Лекция «NoSQL системы»

Михайлова Елена Георгиевна
Графеева Наталья Генриховна

Санкт-Петербург
2019

NoSQL: введение

В предыдущем модуле мы рассмотрели функциональность, которая является общепринятой для систем управления данными, называемых реляционными или традиционными. Однако, в последнее десятилетие наряду с развитием традиционных систем управления данными появилось новое направление систем с достаточно амбициозным названием NoSQL. Эти системы позиционируют себя как альтернативные системы хранения, которые в состоянии хранить и обрабатывать гигантские объемы данных. Из-за большого разнообразия существующих сегодня NoSQL систем трудно понять, что они на самом деле собой представляют, и, тем более, дать рекомендации по использованию их для конкретных прикладных задач. Попробуем описать это явление подробнее с примерами реально существующих систем. Но начнем с истории.

История NoSQL

УНИВЕРСИТЕТ ИТМО

Автор термина - Йохан Оскарсон.

2009 год – термин впервые используется на конференции по нереляционным базам данных.

Not Only SQL



The illustration shows a speaker standing at a podium with a presentation board behind them. The board displays a pie chart and a bar chart. An audience of several people is seated in rows, facing the speaker. The overall scene depicts a professional conference or seminar setting.

Происхождение термина NoSQL приписывается Йохану Оскарсону (Johan Oskarsson), который использовал его на конференции по нереляционным базам данных в 2009 году. Сам термин NoSQL понимается как <NotOnlySQL>. Термин используется в качестве главной классификации, подразумевающей большое разнообразие хранилищ данных, многие из которых не основаны на реляционной модели данных, в том числе узко ориентированных на очень специфические модели данных, например, графы.

Цели создания NoSQL-хранилищ

УНИВЕРСИТЕТ ИТМО



- ✓ Работа на кластерах из недорогих компьютеров в облачной среде.
- ✓ Легкое масштабирование.
- ✓ Хранение и обработка слабоструктурированных и неструктурированных данных.

Большинство систем NoSQL создавались с расчетом на работу на кластерах из недорогих компьютеров в облачной среде, легкое масштабирование и хранение слабоструктурированных и не структурированных данных. Разберемся, что стоит за каждым из этих понятий.

Работа на кластерах в облачной среде



Работа на кластерах в облачной среде означает, что идеологи NoSQLсистем изначально хотели уменьшить стоимость используемых ресурсов и упростить доступ к ним. Для уменьшения стоимости было предложено разрабатывать NoSQLсистемы как распределенные и использовать для хранения и обработки данных кластеры, т.е. группы компьютеров, объединенные высокоскоростными каналами связи и представляющие с точки зрения пользователей единый аппаратный ресурс.

Работа на кластерах в облачной среде



Легкий доступ к мощностям и ресурсам должны были обеспечивать разнообразные Интернет-сервисы. Очевидно, что поддержка транзакций и строгой согласованности данных в условиях распределенных систем требует значительного числа синхронных взаимодействий. Однако есть ряд задач, когда объективно нет необходимости в поддержке транзакций. Например, аналитические приложения, в которых нет регулярных обновлений и для которых информация, поступающая в последние часы, минуты и секунды, не является существенной. Тем более поддержка транзакций не актуальна для хранения изображений, видеороликов, файлов и т.п. Именно для такого рода задач в первую очередь и создавались NoSQLсистемы.

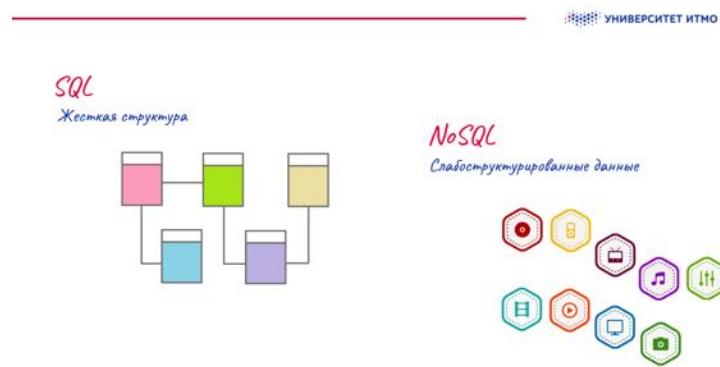
Легкая масштабируемость

$$\frac{\text{прирост производительности}}{\text{прирост используемых ресурсов}} \approx 1$$



Масштабируемость — следующий чрезвычайно важный аспект систем обработки данных, если для них предусматривается сценарий работы под большой нагрузкой. Систему называют **масштабируемой**, если она способна увеличивать производительность пропорционально дополнительным ресурсам.

Масштабируемость принято количественно оценивать через отношение прироста производительности системы к приросту используемых ресурсов. Системы, в которых это отношение близко к единице принято называть **легко масштабируемыми**. Именно это свойство должно учитываться разработчиками NoSQL при выборе архитектуры систем, так как для потенциальных приложений NoSQL систем характерны сценарии работы с высокой нагрузкой.



Последний из упомянутых аспектов – **возможность хранения и обработки в NoSQL системах слабоструктурированных и неструктурированных данных**. Напомним, что когда вы собираетесь хранить данные в реляционной базе, Вы сначала определяете жесткую схему (структуру) базы данных, т.е. указываете, какие существуют таблицы и столбцы, и задаете типы данных, которые могут содержаться в этих столбцах.

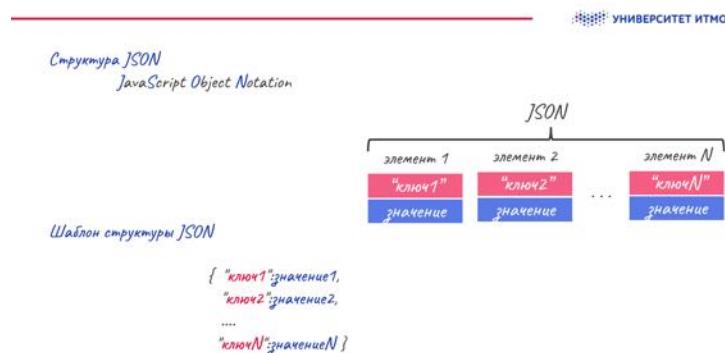
Неструктурированные или слабоструктурированные данные, напротив, не требуют обязательного задания схемы данных. Примерами неструктурных данных могут служить файлы видео, аудио и изображениями, содержимое писем и т.п. Типичными примерами данных слабоструктурированных форматов являются структуры в форматах XML (eXtensible Markup Language) и JSON (JavaScript Object Notation). На приведенном слайде вы можете видеть примеры таких слабоструктурированных данных.

A small cartoon illustration of a lion cub walking towards the left.

Примеры слабоструктурированных данных	
XML	JSON
<pre><Cartoon> <CartoonName>The Lion King</CartoonName> <Year>1994</Year> <Country>USA</Country> <Duration>88</Duration> <FilmDirectors> <Name>Roger Allers</Name> <Name>Rob Minkoff</Name> </FilmDirectors> </Cartoon></pre>	<pre>{ "CartoonName": "The Lion King", "Year": 1994, "Country": "USA", "Duration": 88, "FilmDirectors": ["Roger Allers", "Rob Minkoff"] }</pre>

В приведенном примере видно, что оба формата представляют одни и те же данные – о мультфильме Король Лев. Однако правая структура (в формате JSON) более компактна и выразительна. Именно поэтому она чаще других используется для представления данных в NoSQL.

системах. Рассмотрим более формально, что собой представляет структура формата JSON. Структуру (или документ) формата JSON можно представить как контейнер, состоящий из элементов. Каждый элемент в таком контейнере — это некоторая структурная единица, состоящая из ключа и значения.



При этом значение напрямую связано с ключом и вместе они образуют, так называемую, пару ключ-значение. Для того чтобы получить значение в таком объекте, необходимо знать его ключ.

Ключ отделяется от значения с помощью знака двоеточия (:). Для отделения одного элемента (пары ключ-значение) от другого используется знак запятая (,). Вся структура заключается в фигурные скобки ({}).

Пример записи структуры JSON

```
{
  "CartoonName": "The Lion King",
  "Year": 1994,
  "Country": "USA",
  "Duration": 88,
  "FilmDirector": [
    "Roger Allers",
    "Rob Minkoff"
  ]
}
```

строка
число
строка
число
массив

Значение ключа в JSON может быть записано в одном из следующих форматов: string (строка), number (число), object (объект), array (массив), boolean (логическое значение true или false), null (специальное значением, которое указывает на отсутствие значения любого другого типа). На слайде приведен пример структуры JSON, состоящей из различных типов данных.

Следует обратить внимание на использование специального значения – null. Если какому-то ключу нет соответствующего значения, то упоминать соответствующую пару **ключ:null** можно, но никакой необходимости нет.

В NoSQL хранилищах JSON структуры объединяются в коллекции (аналог таблиц в реляционных базах). Однако в отличие от таблиц в коллекцию могут объединяться данные различной структуры. Например, следующие две структуры могут оказаться в одной коллекции, несмотря на то что количество пар ключ-значение в структурах различается.

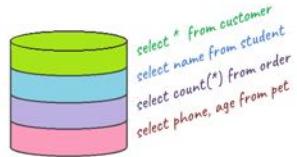
Пример JSON коллекции

NOSQL. Основные характеристики

NoSQL системы весьма разнообразны и решения, которые они используют, не ограничены никакими формальными рамками. Тем не менее, попытаемся перечислить некоторые характеристики, которые дают основания хранилищам причислять себя к системам типа NoSQL.

Языки запросов в NoSQL системах

- ✓ Декларативные (иногда SQL)
- ✓ Низкоуровневые (как правило)



Одно из самых больших заблуждений состоит в том, что наиболее общей характеристикой всех этих систем является отказ от использования языка SQL. Это утверждение не соответствует действительности, так как многие NoSQL системы используют SQL-подобные языки запросов. Хотя надо признать, что в списке NoSQL есть системы, которые, действительно отказались от использования декларативных запросов и формулируют запросы на языках низкого уровня.

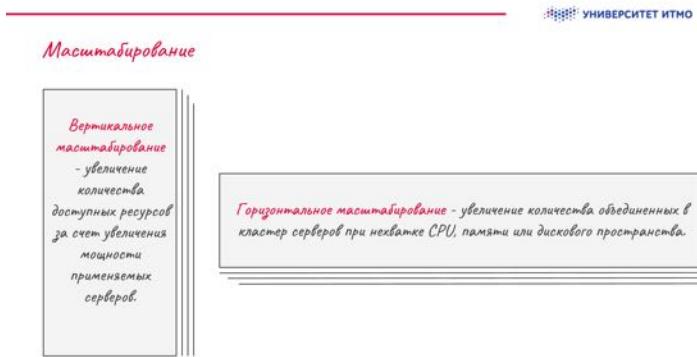
Основные модели данных

- ✓ Ключ-значение
- ✓ Документные
- ✓ Колоночные
- ✓ Граф-ориентированные

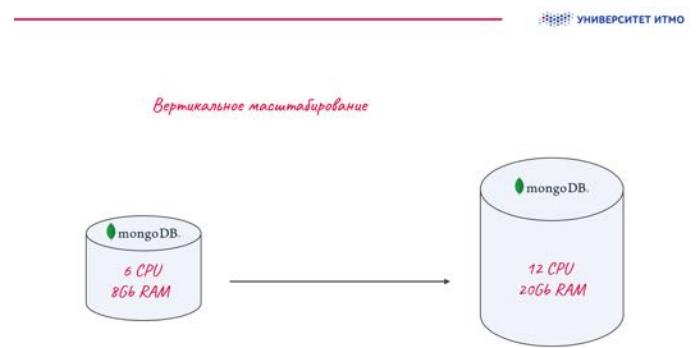


NoSQL системы используют, как правило, простые и, предназначенные для широкого круга задач. Современные NoSQL модели данных принято делить на

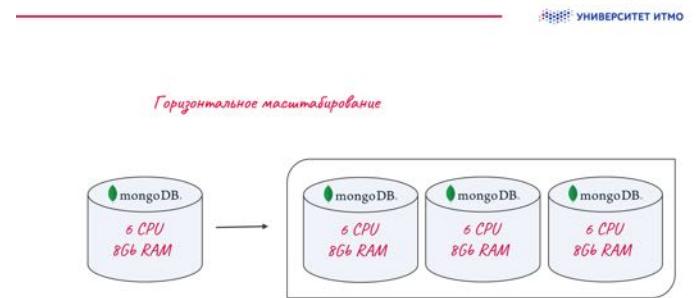
четыре категории, а именно: хранилища типа ключ-значение, документные хранилища, колоночные хранилища и граф-ориентированные. На самом деле моделей значительно больше, однако остальные не так широко распространены. Кроме того, последнее время многие системы позволяют использовать несколько моделей. В этом случае их называют мультимодальными или гибридными.



Для NoSQL систем характерно наличие возможностей для масштабирования как вертикального, так и горизонтального.

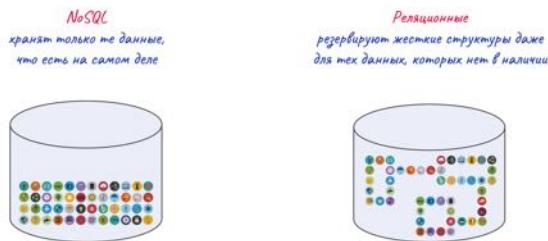


(Вертикальное масштабирование (scaling up) означает увеличение количества доступных ресурсов за счет увеличения мощности применяемых серверов.



Горизонтальное масштабирование означает разбиение системы на более мелкие структурные компоненты и разнесение их по отдельным серверам и увеличение количества серверов, параллельно выполняющих одну и ту же операцию.

Горизонтальная масштабируемость в этом контексте означает возможность не изменять мощности используемых серверов (что характерно для вертикального масштабирования), а добавлять к системе новые узлы для увеличения общей производительности. Этот способ масштабирования может потребовать внесения изменений в программное обеспечение на стороне приложения или СУБД. Большинство NoSQL систем изначально проектируется с расчетом на горизонтальное масштабирование и решает эти проблемы на стороне СУБД.



В базах данных NoSQL хранение данных происходит значительно проще, чем в традиционных. Сторонники неструктурированных баз данных подчеркивают их свободу и гибкость. Хранилища типа "ключ-значение" позволяют сохранять и выбирать данные по ключу и не задумываться о содержании данных. Документные хранилища данных, по существу, делают то же самое, но при этом позволяют манипулировать с фрагментами слабоструктурированных данных. Графовые хранилища свободно добавляют новые узлы и ребра в граф, а также новые свойства к узлам и ребрам. Колоночные хранилища также позволяют легко добавлять новые поля к отдельным записям и удалять старые, не вынуждая при этом проводить глобальную перестройку всей базы. Помимо обеспечения удобных изменений, неструктурированные базы данных облегчают обработку неоднородных данных, т.е. данных, в которых все записи имеют разные наборы полей. В базах с традиционной организацией схема вынуждает хранить все записи в одинаковом формате. Это может оказаться неудобным, если в разных строках хранятся разные данные. В этом случае множество неиспользуемых полей оказываются заполненными неопределенными или заданными по умолчанию значениями. Неструктурированные базы данных позволяют избежать этого, позволяя каждой записи хранить все, что требуется, - не больше и ни меньше.

Представление данных в виде агрегатов

FilmDirector		SongWriterName	
<i>idFilmDirector</i>	Name	<i>idSongWriter</i>	Name
1	Roger Allers	1	Elton John
2	Rob Minkoff	2	Hans Zimmer

CartoonFilmDirector		CartoonSongWriter	
<i>idCartoon</i>	Name	<i>idCartoon</i>	Name
1	1	1	1
2	2	2	2

Представление данных в виде агрегатов

FilmDirector		SongWriterName	
<i>idFilmDirector</i>	Name	<i>idSongWriter</i>	Name
1	Roger Allers	1	Elton John
2	Rob Minkoff	2	Hans Zimmer

CartoonFilmDirector		CartoonSongWriter	
<i>idCartoon</i>	Name	<i>idCartoon</i>	Name
1	1	1	1
2	2	2	2

<i>idCartoon</i>	Name	Year	Country	Duration
1	1	1994	USA	88

В отличие от реляционной модели, которая хранит сущности в нормализованном виде в различных таблицах, большинство NoSQL хранилищ оперируют с этими сущностями как с целостными объектами и хранят их в денормализованном виде.

Представление данных в виде агрегатов

FilmDirector		SongWriterName	
<i>idFilmDirector</i>	Name	<i>idSongWriter</i>	Name
1	Roger Allers	1	Elton John
2	Rob Minkoff	2	Hans Zimmer

CartoonFilmDirector		CartoonSongWriter	
<i>idCartoon</i>	Name	<i>idCartoon</i>	Name
1	1	1	1
2	2	2	2

<i>idCartoon</i>	Name	Year	Country	Duration
1	1	1994	USA	88

```
{
  "_id": 1,
  "CartoonName": "The Lion King",
  "Year": 1994,
  "Country": "USA",
  "Duration": 88,
  "FilmDirectorName": ["Roger Allers", "Rob Minkoff"],
  "SongWriterName": ["Elton John", "Hans Zimmer"]
}
```

В примере на слайде продемонстрировано реляционное представление для базы “мультифильмы-режиссеры-композиторы” и соответствующий ему агрегат в формате JSON.

Обратите внимание, что в отличие от реляционной модели, где данные “размазаны” по пяти таблицам в агрегате NoSQL данные о мультифильме объединяются со списком режиссеров и композиторов в один логический объект. Такая денормализация нужна, чтобы не запрашивать имена режиссеров и композиторов при извлечении фильма. Основной смысл такого объединения – минимизация количества соединений между различными объектами. Этим демонстрируется главное правило проектирования структуры данных во многих NoSQL базах — они должны подчиняться требованиям приложения и быть

максимально оптимизированы под наиболее частые запросы. Если режиссеры регулярно извлекаются вместе с фильмом — имеет смысл их включать в общий объект, если же большинство запросов работает только с режиссерами — значит, лучше их вынести в отдельную сущность. Очевидно, что работа с большими денормализованными объектами чревата многочисленными проблемами при выполнении произвольных запросов к данным, когда запросы не укладываются в структуру агрегатов. В этом случае при получении требуемой информации нам придется извлекать массу информации, которая абсолютно не нужна. К сожалению, это компромисс, на который приходится идти в распределенной системе: мы не можем проводить нормализацию данных как в обычной односерверной системе, так как это создаст необходимость объединения данных с разных узлов и может привести к значительному замедлению работы базы.



Долгое время согласованность (Consistency) данных была “священной” для разработчиков баз данных. Все реляционные базы обеспечивали тот или иной уровень изоляции транзакций. С приходом огромных массивов информации и распределенных систем стало ясно, что обеспечить для них изолированность транзакций с одной стороны и получить высокую доступность и быстрое время отклика с другой — невозможно.



Более того, даже обновление одной записи не гарантирует, что любой другой пользователь моментально увидит изменения в системе, ведь изменение в распределенной системе может произойти, например, в мастер-узле, а реплика асинхронно скопируется на узлы, с которыми работают другие пользователи. В таком случае пользователи увидят результат через какой-то, пусть минимальный,

но все же промежуток времени. Это свойство принято называть “согласованностью в конечном счете” (Eventual Consistency) и это то, на что идут сейчас все крупнейшие интернет-компании мира, включая Facebook и Amazon. Последние, например, декларируют, что максимальный интервал, в течение которого пользователь может видеть несогласованные данные, составляет не более секунды. Пример такой ситуации показан на слайде.



Два пользователя с разницей в несколько миллисекунд забронировали один и тот же номер. Второй пользователь умудрился попасть в момент заказа именно в тот временной интервал, когда данные на несколько миллисекунд оказались несогласованными. Такое событие крайне маловероятно, но оно все же случилось! Что делать в этом случае? Разумеется, бизнес, который осознанно выбрал приложение и хранилище, обеспечивающие быстрый отклик системы для конечного пользователя и свойство “согласованности в конечном счете” (EventualConsistency) должен предусматривать какие-то сценарии отката операции как с точки зрения приложения, так и самого бизнеса. В приведенном примере –в гостиницах принято держать “пул” свободных номеров на непредвиденный случай, а в приложениях выполнять откат несогласованного заказа. На самом деле слабая поддержка согласованности не означают, что ее нет вообще. NoSQL системы поддерживают ее в той мере, в какой она необходима для целей конкретных приложений.

BASE-свойства

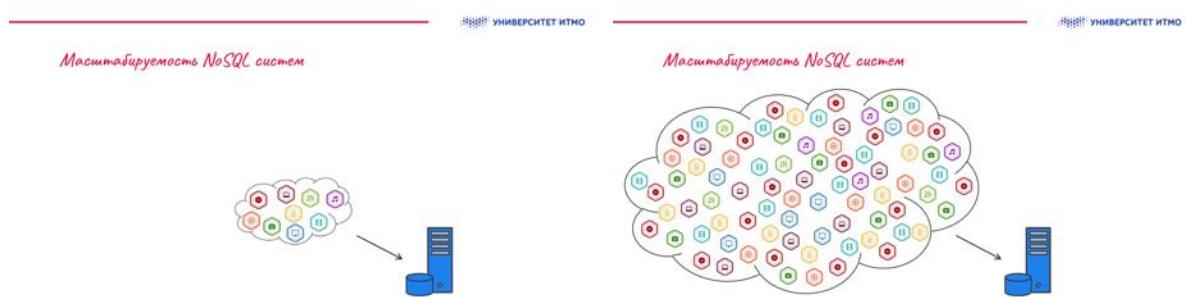
- ✓ Basic Availability - данные доступны всегда
- ✓ Soft State - данные могут находиться в рассогласованном состоянии
- ✓ Eventual Consistency - в конечном счете данные в хранилище окажутся в согласованном состоянии

Многие NoSQL системы поддерживают, так называемые, BASE-свойства (Basic Availability, SoftState, Eventual Consistency). В этом названии BasicAvailability

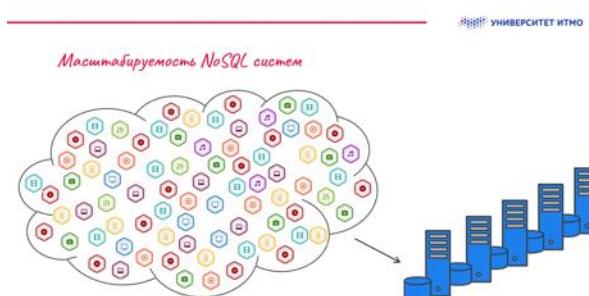
означает, что данные доступны всегда, когда к ним происходит обращение, даже если часть из них в настоящий момент недоступна; *SoftState* означает, что данные могут находиться в рассогласованном состоянии какой-то период времени, а *EventualConsistency* означает, что в конечном счете, после некоторого периода времени данные в хранилище окажутся в согласованном состоянии. Тем не менее, некоторые NoSQL хранилища данных, например Couch DB обеспечивают полную поддержку согласованности, как это принято в традиционных СУБД.

В следующем фрагменте лекции мы обсудим технологии, используемые для NoSQL систем.

NoSQL: Технологии



Лавинообразный рост количества данных обострил проблему масштабируемости данных - вычислительные мощности компьютеров не могут расти бесконечно, да и цена нескольких простых серверов меньше, чем цена одного высокопроизводительного сервера.



Как мы уже упоминали ранее, в такой ситуации хорошим выходом становится горизонтальное масштабирование, в котором несколько независимых серверов соединяются вместе в кластер и каждый из них обрабатывает только часть запросов. Такая архитектура делает возможным быстрое увеличение мощности кластера путем добавления нового сервера. NoSQL хранилища изначально проектируются как распределенные системы с таким расчетом, что все процедуры репликации, распределения данных и обеспечения отказоустойчивости выполняются самой NoSQL системой.

Технологии для обеспечения распределенной работы

- ✓ Репликация
- ✓ Фрагментация (шардинг)
- ✓ MapReduce



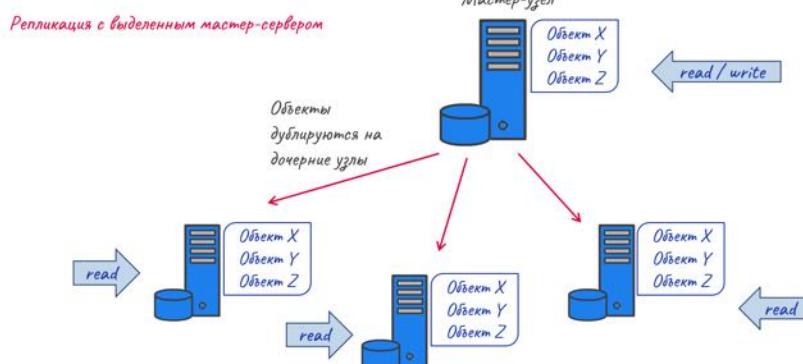
Ключевые моменты NoSQL систем по организации распределенной работы реализуются с помощью технологий **репликации, фрагментации (или шардинга) и технологии MapReduce**. Рассмотрим эти технологии подробнее.

Репликация — это копирование данных при их обновлении на другие сервера.

- Репликация с выделенным мастер-сервером.
- Репликация одноранговая.

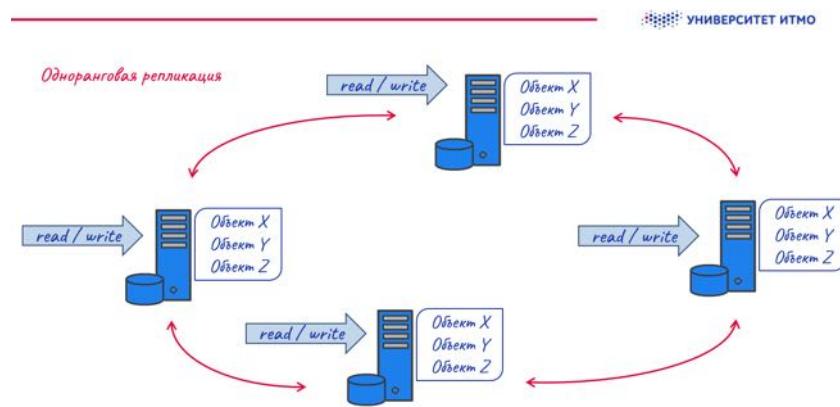


Репликация — это копирование данных при их обновлении на другие сервера. Именно этот механизм позволяет добиться большей отказоустойчивости и масштабируемости системы. Принято выделять два вида репликации: с выделенным мастер-сервером (master-slave) и одноранговую (peer-to-peer).

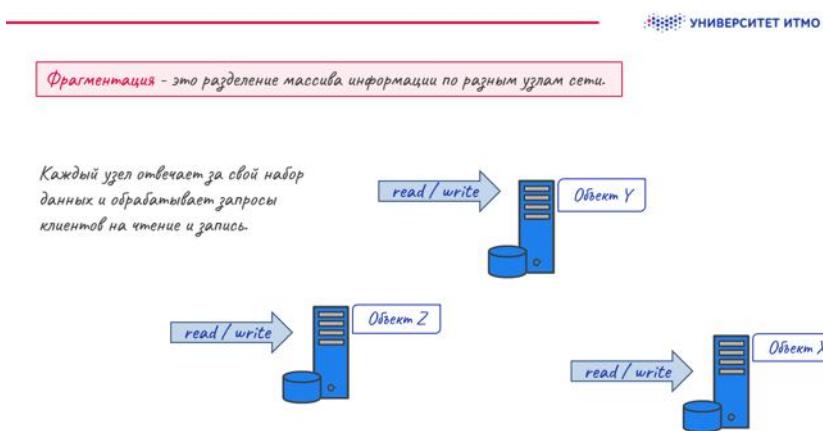


Первый тип подразумевает наличие одного мастер-сервера и нескольких дочерних серверов. Запись может производиться только на мастер-сервер, а он в свою очередь передаёт изменения на дочерние машины. Этот тип репликации даёт хорошую масштабируемость на чтение (чтение может происходить с любого узла

сети), но не позволяет масштабировать операции записи - запись идёт только на один мастер-сервер. Такой вариант организации репликации предполагает сложности в случае неисправности мастер-сервера, так как в таком случае должен происходить автоматический или ручной выбор нового мастер-сервера из оставшихся.

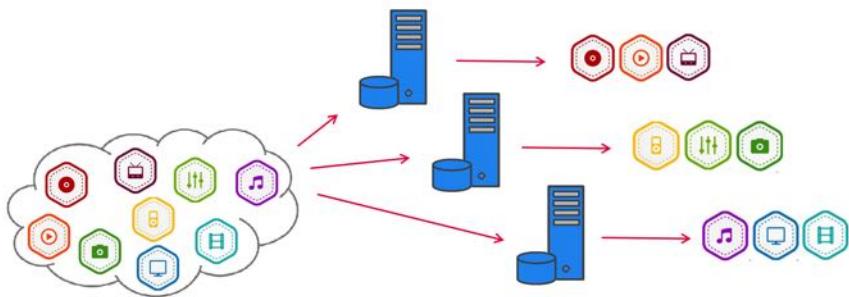


Второй тип – одноранговая сеть (peer-to-peer) - предполагает, что все узлы равны в возможности обслуживать запросы на чтение и запись. Информация об обновлении данных передаётся от сервера к серверу по кругу.



Следующая технология – фрагментация. Фрагментация (sharding) — это разделение массива информации по разным узлам сети - когда каждый узел отвечает только за определенный набор данных и обрабатывает запросы на чтение и запись, относящиеся только к этому набору данных. Эта технология и раньше использовалась при работе с реляционными базами данных, но в достаточно сыром виде. Создавались независимые базы данных, по которым приложение распределяло запросы пользователей. Концепция NoSQL предполагает, что ответственность за этот механизм возлагается на саму базу данных, и фрагментация производится автоматически.

MapReduce – это технология эффективного распараллеливания задач по кластеру.



Последняя из упомянутых технологий – технология **MapReduce**. Для подавляющего большинства NoSQL систем характерно использование технологии MapReduce. MapReduce – это технология эффективного распараллеливания задач по кластеру. Эта технология сегодня имеет множество разнообразных реализаций. Но есть классическая идея, которую мы попытаемся изложить. Рассмотрим простой пример.

Быть или не быть, вот в чем вопрос.
To be, or not to be, that is the question

Достойно ль
Whether 'tis nobler in the mind to suffer

Смиряться под ударами судьбы,
The slings and arrows of outrageous fortune

Иль надо оказать сопротивление...
Or to take arms against a sea of troubles



Представьте, что вам нужно узнать частоту использования тех или иных слов в произведениях Шекспира. Вы, конечно, можете начать просматривать все его произведения и выписывать пары (слово, частота слова в предложении) для каждого встретившегося слова. Когда Вы дойдете до конца последнего произведения – Вы сможете проаггрегировать результаты и определить, какое слово сколько раз встретилось.

Быть или не быть, вот в чем вопрос.
To be, or not to be, that is the question

Достойно ль
Whether 'tis nobler in the mind to suffer

Смиряться под ударами судьбы,
The slings and arrows of outrageous fortune

Иль надо оказать сопротивление...
Or to take arms against a sea of troubles



Но, очевидно, что на эту работу Вам придется потратить много времени. Вы можете попробовать попросить своих друзей помочь Вам и раздать каждому из них по произведению. Однако друзей должно быть достаточно много, так как Шекспир написал 38 пьес, 154 сонета, а также еще несколько других поэтических произведений.

Быть или не быть, вот в чем вопрос.
To be, or not to be, that is the question

Достойно ль
Whether 'tis nobler in the mind to suffer

Смиряться под ударами судьбы,
The slings and arrows of outrageous fortune

Иль надо оказать сопротивление...
Or to take arms against a sea of troubles

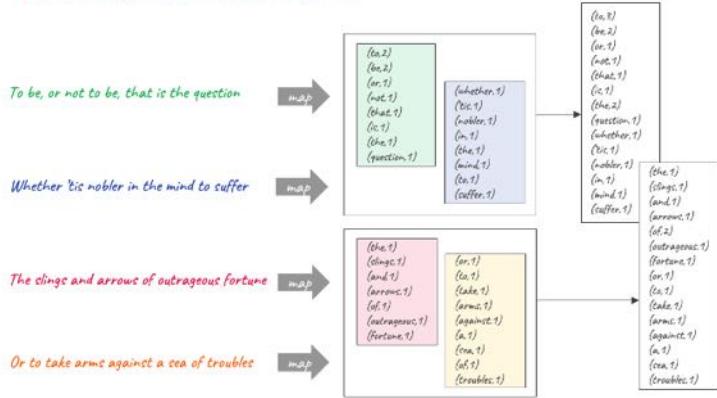


Кроме того, даже если Вам удастся найти такое количество друзей, то желательно, чтобы они предоставили результат анализа произведений не в виде длинных списков слов, а в агрегированном виде, где про каждое слово уже будет известно, сколько раз оно встретилось в произведении. А уже Вы, собрав все предварительные результаты агрегирования, сможете провести финальную агрегацию для каждого слова.

Собственно, в этом простом примере и изложена вся технология MapReduce. Как этот пример может быть трактован с точки зрения современных технологий?

Когда Вы собрали друзей - Вы сформировали кластер для проведения распределенных вычислений.

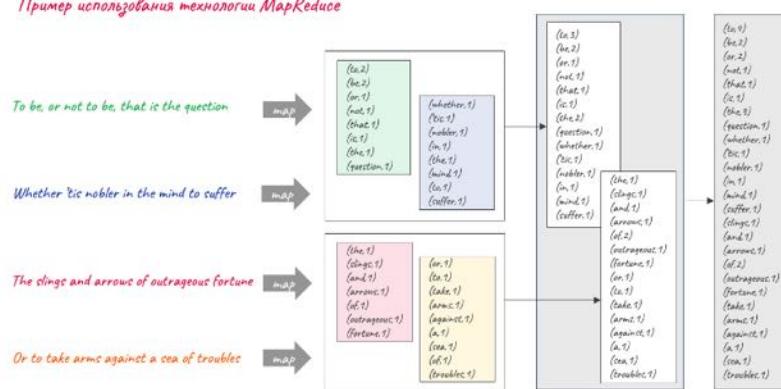
Пример использования технологии MapReduce



Когда Вы разделили произведения Шекспира среди своих друзей - Вы поступили как мастер-узел кластера, который распределяет данные для вычислений на кластере.

Когда Ваши друзья выписывали пары <слово, частота> (т.е. формировали пары ключ-значение) - они выполняли фазу Map (отображение) из технологии MapReduce.

Пример использования технологии MapReduce



Когда Ваши друзья, а затем Вы агрегировали результаты по каждому слову - вы выполняли фазу Reduce (свертка) из технологии MapReduce.

Рисунок на слайде демонстрирует основные этапы технологии MapReduce на примере анализа сонета Шекспира.

NoSQL: примеры NoSQL систем

Модели данных и методы работы с данными систем класса NoSQL чрезвычайно разнообразны. Количество таких систем уже трудно сосчитать. На слайде вы видите ссылку, которая представляет один из наиболее полных перечней NoSQLсистем.

В этом списке числится более 200 систем. Почему их так много? Причина состоит в том, что характерной особенностью этих систем является ориентация на узкий класс задач. А прикладных задач действительно много и становится все больше. Так какие из систем являются наиболее востребованными? Как понять, что следует изучать? Каким-нибудь случайным образом выбрать их из списка?



Нет, мы не будем полагаться на случай. Вспомним, что системы NoSQL разбиваются на разного рода группы, среди которых особо выделяют следующие 4 категории: колоночные, ключ-значение, документные и графовые.

Категории NoSQL хранилищ

Сервис DB-Engines Ranking
<https://db-engines.com/en/ranking>

Нужно посмотреть рейтинг!



Попытаемся найти наиболее рейтинговых представителей этих категорий.

DB-ENGINES
Knowledge Base of Relational and NoSQL Database Management Systems

Home | DB-Engines Ranking | Systems | Encyclopedia | Blog | Search | Vendor Login

provided by solid.IT

Select a ranking

- Complete ranking
- Relational DBMS
- Key-value stores
- Document stores
- Graph DBMS
- Time Series DBMS
- Object oriented DBMS
- RDF stores
- Search engines
- Wide column stores
- Multivalue DBMS
- Native XML DBMS
- Event stores
- Content stores
- Navigational DBMS

Special reports

- Ranking by database model
- Open source vs.

RSS Feed

DB-Engines Ranking

The DB-Engines Ranking ranks database management systems according to their popularity. The ranking is updated monthly.

Read more about the [method](#) of calculating the scores.

trend chart

352 systems in ranking, September 2019

Rank	DBMS	Database Model	Score		
Sep 2019	Sep 2019	Sep 2018	Sep 2019	Sep 2019	Sep 2018
1.	Oracle	Relational, Multi-model	1346.66	+7.18	+37.54
2.	MySQL	Relational, Multi-model	1279.07	+23.39	+498.60
3.	Microsoft SQL Server	Relational, Multi-model	1085.06	-8.12	+33.78
4.	PostgreSQL	Relational, Multi-model	482.25	+0.91	+75.82
5.	MongoDB	Document	410.06	+5.50	+51.27
6.	IBM Db2	Relational, Multi-model	171.56	-1.39	-6.50
7.	Elasticsearch	Search engine, Multi-model	149.27	+0.19	+6.67
8.	Redis	Key-value, Multi-model	141.90	-2.18	+0.96

Обратимся к сервису DB-EnginesRanking, который по определенным алгоритмам (а именно: упоминание в разнообразных публикациях и т.п.) формирует рейтинги наиболее популярных систем разных категорий.

Выберем в интерфейсе категорию ключ-значение.

Рейтинг систем категории **ключ-значение**

67 systems in ranking, February 2019

Rank	DBMS			Database Model	
Feb 2019	Jan 2019	Feb 2018	Feb 2019	Jan 2019	Feb 2018
1.	Redis	Key-value store	149.45	+0.43	+22.43
2.	Amazon DynamoDB	Multi-model	54.95	-0.15	+15.07
3.	Memcached	Key-value store	29.45	-0.09	+0.51
4.	Microsoft Azure Cosmos DB	Multi-model	24.86	+0.47	+8.67
5.	Hazelcast	Key-value store	8.89	+0.27	+0.22

Наиболее популярными представителями систем категории класс-значение согласно рейтингу DB-EnginesRanking на момент записи этой лекции являются: Redis, Amazon Dynamo DB (мультимодельная система), Memcached. Утверждается, что эти системы позволяют добиваться высокой производительности, могут легко масштабироваться, но эффективный поиск в таких моделях возможен только по уникальному ключу. При этом сам запрашиваемый объект может быть как последовательностью байтов, так и более сложной структурой. Авторы систем декларируют высокую производительность, однако надо признать, что для объективного сравнения производительности систем категории ключ-значение и традиционных СУБД необходимо произвести сравнение на основе сопоставимых операций. Операции, выполняемые в традиционных СУБД с помощью одного высокоуровневого запроса, в системах типа ключ-значение могут потребовать десяток, а то и сотню низкоуровневых операций. Сравнение на основе мелких операций типа ключ-значение представляется не совсем справедливым, так как сильной стороной высокоуровневых запросов является именно массированная, а не единичная обработка данных. К серьезным недостаткам этих систем можно отнести полное отсутствие отношений между сущностями. Это означает, что система управления хранилищем не может проконтролировать целостность отношений и соответствующая функциональность целиком ложится на приложения. Тем не менее, для некоторого класса задач этого вполне достаточно и такие хранилища используются весьма эффективно (например, для хранения изображений, видеороликов, файлов и т.п.).

УНИВЕРСИТЕТ ИТМО

Рейтинг документных хранилищ

46 systems in ranking, February 2019									
Rank	DBMS			Database Model	Score			Feb 2019	Jan 2019
	Feb 2019	Jan 2019	Feb 2018		Feb 2019	Jan 2019	Feb 2018		
1.	1.	1.	MongoDB	Document store	395.09	+7.91	+56.67		
2.	2.	2.	Amazon DynamoDB	Multi-model	54.95	-0.15	+15.07		
3.	3.	3.	Couchbase	Document store	35.58	+0.98	+3.83		
4.	4.	4.	Microsoft Azure Cosmos DB	Multi-model	24.86	+0.47	+8.67		
5.	5.	4.	CouchDB	Document store	20.00	+0.68	-0.30		

Выберем в качестве фильтра – документные хранилища и увидим следующую картину. Среди систем, ориентированных на обработку документов, согласно рейтингу DB-EnginesRanking заслуживают упоминания: MongoDB, CouchBase и уже упомянутая ранее мультимодельная система Amazon Dynamo DB. В рамках документарной модели эти системы хранят объекты (документы) в формате JSON (Java Script Object Notation) или BSON (Binary JSON). Форматы JSON и BSON допускают атрибуты простых типов, массивы, а также вложенные объекты. Эти системы поддерживают индексы на полях документов и позволяют строить сложные запросы. Полноценных транзакций в них также нет. Тем не менее, операции обновления на уровне одного документа обычно являются

атомарными. Такие хранилища данных эффективно применяются в системах управления содержимым, издательском деле, документальном поиске и т. п.



Рейтинг систем, ориентированных на обработку графов

Rank			DBMS	Database Model	Score		
Feb 2019	Jan 2019	Feb 2018			Feb 2019	Jan 2019	Feb 2018
1.	1.	1.	Neo4J	Graph DBMS	47.86	+1.06	+8.04
2.	2.	2.	Microsoft Azure Cosmos DB	Multi-model	24.86	+0.47	+8.67
3.	3.	3.	Datastax Enterprise	Multi-model	9.15	+0.34	+3.20
4.	4.	4.	OrientDB	Multi-model	6.05	+0.31	+0.11
5.	5.	5.	ArangoDB	Multi-model	4.36	+0.07	+0.89

Выберем в качестве фильтра – графовые системы. Это системы, ориентированные на обработку графов и предназначены для хранения узлов графов и связей между ними. Как правило, такого рода системы позволяют задавать для узлов и связей еще и набор произвольных атрибутов и выбирать узлы и связи по этим атрибутам. Кроме того, системы поддерживают алгоритмы обхода графов и построения маршрутов. Согласно рейтингу DB-EnginesRanking список граф-ориентированных хранилищ возглавляют: Neo4j, Microsoft Azure Cosmos DB, Datastax Enterprise, OrientDb и ArangoDB. При этом последние четыре хранилища также позиционируются как мультимодельные. Граф-ориентированные хранилища эффективно используются для задач, связанных с анализом социальных сетей, выбором маршрутов и т.п.



Рейтинг колоночных хранилищ

Rank			DBMS	Database Model	Score		
Feb 2019	Jan 2019	Feb 2018			Feb 2019	Jan 2019	Feb 2018
1.	1.	1.	Cassandra	Wide column store	123.37	+0.39	+0.59
2.	2.	2.	HBase	Wide column store	60.28	-0.12	-1.43
3.	3.	3.	Microsoft Azure Cosmos DB	Multi-model	24.86	+0.47	+8.67
4.	4.	4.	Datastax Enterprise	Multi-model	9.15	+0.34	+3.20
5.	5.	6.	Microsoft Azure Table Storage	Wide column store	4.78	+0.66	+1.45

Особое место среди NoSQL – систем занимают колоночные хранилища данных. Их также принято относить к NoSQL. Группу колоночных хранилищ

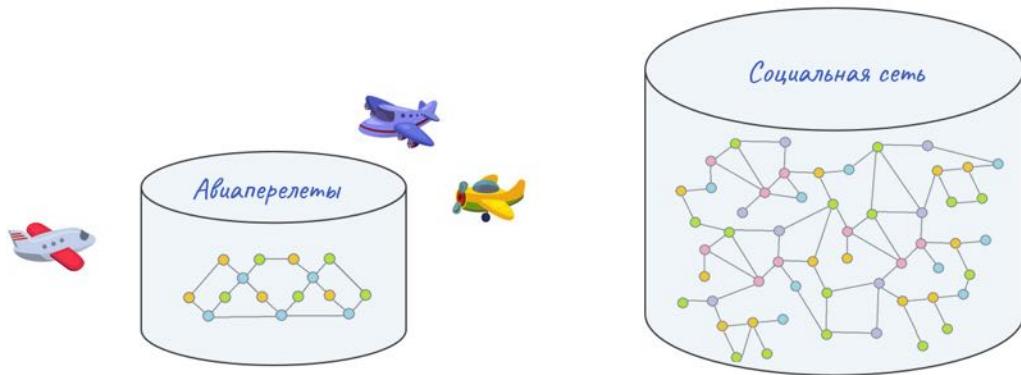
согласно рейтингу DB-Engines Ranking на момент написания лекции возглавляют: Cassandra, HBase, Microsoft Azure Cosmos DB. Основная особенность хранилищ этого типа заключается в том, что данные представляются в виде таблиц, а хранение и фрагментация этих данных возможны не по строкам, как это принято в традиционных СУБД, а по столбцам. Кроме того, для многих систем этого класса характерно наличие SQL-подобных языков высокого уровня.

Категория хранилищ	Характерные представители
Документные хранилища	 mongoDB
Ключ-значение	 redis
Графовые хранилища	 neo4j
Колоночные хранилища	 cassandra

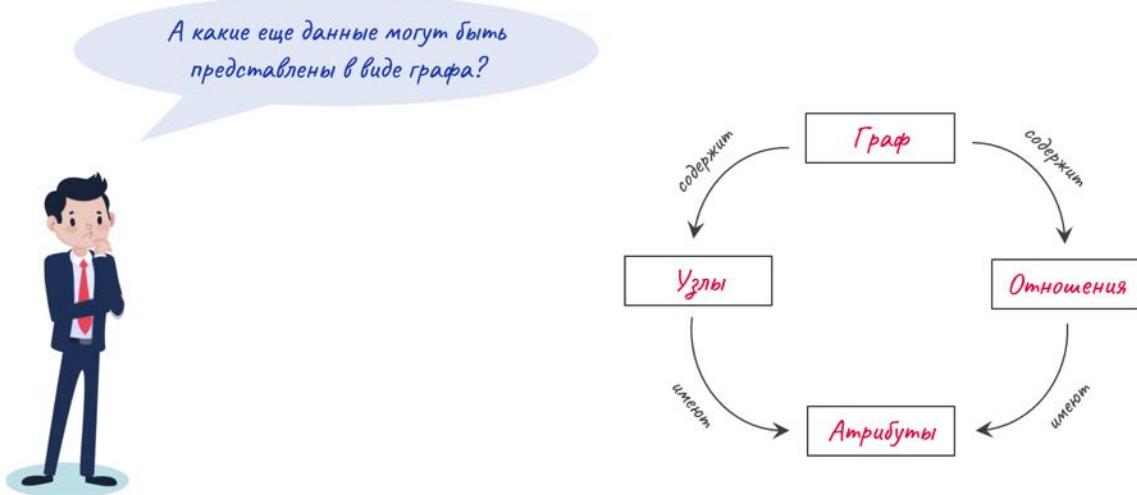
На протяжении десятилетия NoSQL движение набирает популярность. Появляются новые системы, которые позиционируют себя как NoSQL. Какое-то время еще можно было понять принцип работы той или иной системы на основе того, к какой категории она относится (графовая, ключ-значение, документная или колоночная). Однако, даже это деление на категории становится условным, так как многие из хранилищ допускают несколько способов хранения данных и, как следствие, становятся мультимодельными системами. И, тем не менее, попробуем разобраться в принципах работы NoSQL систем на конкретных примерах. В качестве примеров в следующих разделах рассмотрим наиболее ярких представителей каждой категории.

Графовые базы данных (Graph Databases)

Графовые базы данных



Графовые базы данных используют графовую модель для представления и хранения данных. Если в реляционной модели данные хранятся в виде жесткой структуры с предопределенной схемой, то в графовой базе данных никакой предопределенной схемы нет. Скорее сама схема является отражением тех данных, которые поступили в базу. Чем разнообразнее данные – тем сложнее схема.

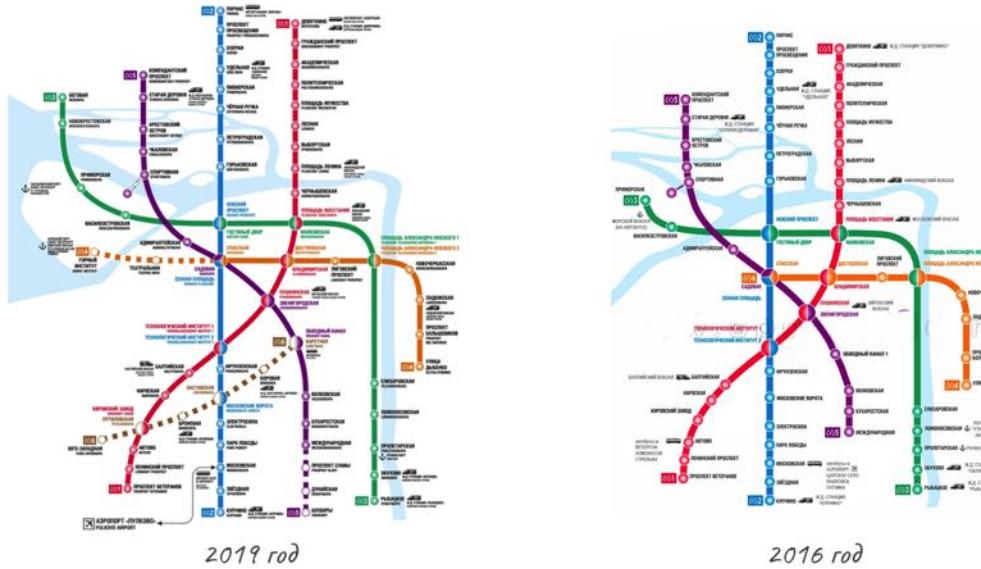


Графовые базы данных позволяют хранить сущности и отношения между ними. Сущности моделируются узлами графа, которые имеют свои атрибуты. Отношения моделируются ребрами, которые также могут иметь свои атрибуты. Ребра имеют направление. Организация графа позволяет один раз записать данные, а затем интерпретировать их разными способами в соответствии с отношениями. Кроме того, к графу легко добавляются новые узлы и новые отношения. В виде графовых баз данных чаще всего представляются данные, которые изначально по сути своей организованы в виде графа.



Схема метрополитена Санкт-Петербурга
<http://www.metro.spb.ru/map.html>

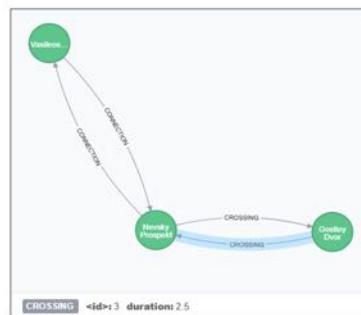
Рассмотрим пример данных, имеющих графовую структуру. Это схема метро Санкт-Петербурга. Еще недавно эта схема имела следующий вид:



Как известно, в 2018 году были открыты новые станции метро: Новокрестовская и Беговая. Схема стала выглядеть так:

Заметьте, что добавление новых станций никак не затронуло существующие части схемы. Всего лишь произошло добавление новых узлов и ребер к графу.

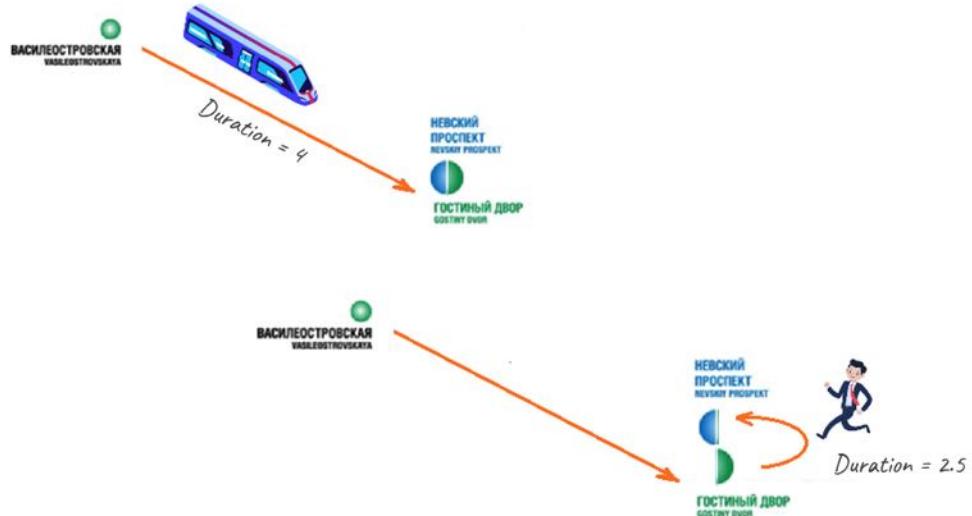
Пример фрагмента графовой базы данных



Экранная форма из Neo4j

Рассмотрим фрагмент небольшой графовой базы данных, содержащий только несколько станций метро. В базе данных, которую вы видите на экране, присутствуют узлы графа, представляющие станции метро. У каждого узла есть свойства. В нашем случае – это одно свойство Name. Узлы графа соединены ребрами. Ребра двух типов: CONNECTION и CROSSING. Содержательно, первый

типа связи означает, что станции, которые она соединяет, связаны одним перегоном (например, между метро Василеостровская и Приморская один перегон). Связь типа CROSSING означает, что возможна пересадка с одной станции на другую (например, со станции Гостиный двор возможна пересадка на станцию Невский проспект).



У каждого типа ребер есть и дополнительная нагрузка – свойство Duration, которое в первом случае означает время, за которое поезд метро преодолевает соответствующий перегон, а во втором случае – время, которое требуется пассажиру, чтобы перейти с одной станции на другую.

A table showing family members:

ID	Name	Mother_ID	Father_ID
...
1001	Мария	901	902
1002	Александр	805	806
1003	Ольга	901	902
1004	Виктор	704	710
1005	Елена	1001	1002
1006	Софья	1001	1002
1007	Валентин	1003	1004
...			

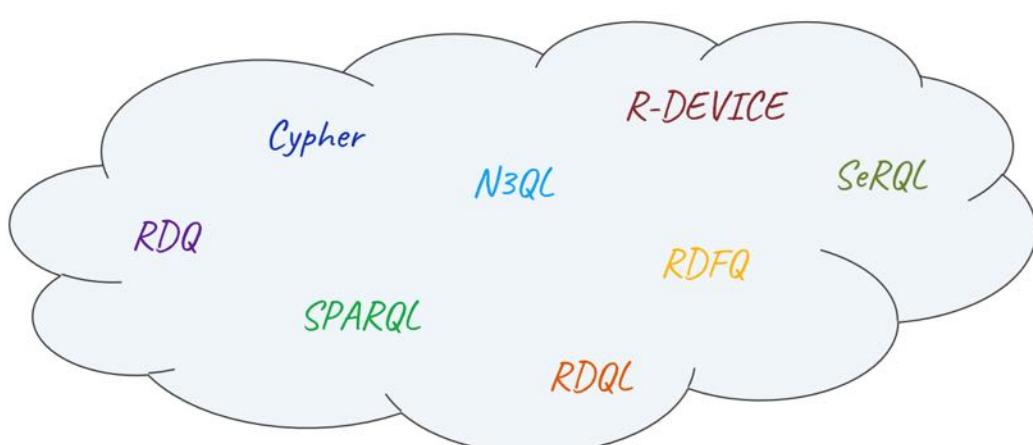
Конечно, и в традиционных реляционных СУБД тоже существуют возможности для хранения связей между однотипными и разнородными

объектами. Однако добавление новых связей в такие схемы является очень тяжелой операцией, требующей перемещения большого количества данных. В графовых базах добавление нового отношения является примитивной операцией, не затрагивающей все остальные данные.



В графовых базах запрос на выборку данных из графа принято называть обходом графа. Обход графа происходит быстрее, чем в реляционных базах потому, что связи (отношения) между узлами не надо вычислять, так как они напрямую присутствуют в структуре хранения графа.

Языки графовых баз данных



Существует много разнообразных графовых баз данных со своими встроенными командами для создания базы. Но, тем не менее, как бы они не отличались друг от друга, их объединяет то, что их основными командами при создании базы являются команды, позволяющие задавать сущности с атрибутами и отношения между ними. Для создания графовых баз и манипуляций над данными используются специализированные языки.



The screenshot shows the official website for the Cypher Graph Query Language. The header features the Neo4j logo and navigation links for Products, Solutions, Customers, Partners, Resources, Developers, and Download Now. A prominent blue bar displays the text '(Cypher)-[:LOVES]-(Graph Technology)'. Below this, the main content area is titled 'Cypher, The Graph Query Language' with a subtitle 'The Standard Query Language for Graph Database Technology'. The page explains that Cypher is a vendor-neutral open graph query language used across the graph ecosystem, featuring a declarative syntax for matching nodes and relationships. It contrasts Cypher with SQL, noting its declarative nature and the ability to perform actions like match, insert, update, or delete directly on graph data. A note mentions that two years ago, Neo4j, Inc. decided to open source the Cypher language to make it the most popular graph query language available to any technology provider with the aim of making Cypher the 'SQL for graphs.' A cookie consent banner at the bottom asks if the user wants to accept necessary cookies or use unnecessary cookies only.

<https://neo4j.com/cypher-graph-query-language/>

В СУБД Neo4J для этих целей используется язык Cypher. Посмотрим, как используя Cypher, можно задать графовую базу на примере создания фрагмента базы метрополитена.



Создание графовой базы данных

Пример (Neo4j), язык Cypher



Создание отдельного узла

(Vasileostrovskaya:station{name:'Vasileostrovskaya'})

Создание отношения

(Vasileostrovskaya)-[CONNECTION]->(NevskyProspekt)

Для создания отдельного узла используется следующая конструкция, которая позволяет задать узел в графе, его тип (station) и задать его свойство Name:

(Vasileostrovskaya:station{name:'Vasileostrovskaya'})

Следующая конструкция в квадратных скобках на экране используется для задания отношений:

(Vasileostrovskaya)-[:CONNECTION]->(NevskyProspekt)

Она позволяет указать имена соединяемых отношений, установить направленную связь между ними.

Мы должны создать отношение между узлами в двух направлениях, потому что направление отношения имеет значение. Узлам должны быть известны входящие и исходящие отношения, которые можно обойти в обоих направлениях. Отношения являются полноправными элементами графовых баз данных и, собственно, ценность этих баз данных в основном обусловлена наличием отношений. Отношения имеют не только тип, начальный и конечный узел, но и свои собственные свойства.



Создание графовой базы данных

Пример (*Neo4j*, язык *Cypher*)



Создание отдельного узла

(Vasileostrovskaya:station{name:'Vasileostrovskaya'})

Создание отношения

(Vasileostrovskaya)-[:CONNECTION]->(NevskyProspekt)

(Vasileostrovskaya)-[:CONNECTION {duration:4.2}]->(NevskyProspekt)

Используя эти свойства, в отношение можно внести дополнительную информацию, например, время прохождения перегона или перехода с одной станции метро на другую. Пример на экране демонстрирует, как можно задавать отношениям дополнительные свойства:

(Vasileostrovskaya)-[:CONNECTION {duration:4.2}]->(NevskyProspekt)

Создание графовой базы данных

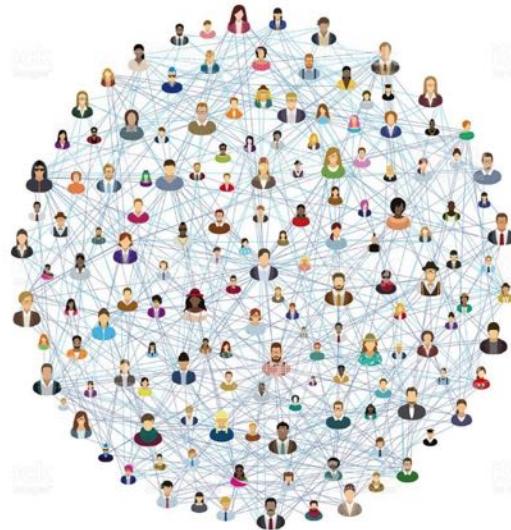
Создание трех станций и отношений между ними



```
CREATE (Vasileostrovskaya:station{name:'Vasileostrovskaya'}),
(NevskyProspekt:station{name:'Nevsky Prospekt'}),
(GostinyDvor:station{name:'Gostiny Dvor'}),
(Vasileostrovskaya)-[:CONNECTION {duration:4}]->(NevskyProspekt),
(NevskyProspekt)-[:CONNECTION {duration:4}]->(Vasileostrovskaya),
(NevskyProspekt)-[:CROSSING {duration:2}]->(GostinyDvor),
(GostinyDvor)-[:CROSSING {duration:2}]->(NevskyProspekt)
```

И, наконец, следующий пример демонстрирует весь код, позволяющий задать три станции метрополитена и отношения между ними.

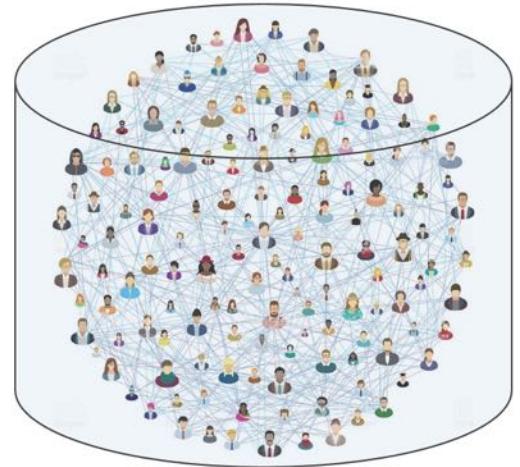
Масштабируемость



Как уже говорилось ранее, при масштабировании баз данных NoSQL широко используется фрагментация, в ходе которой данные разделяются и распределяются по разным серверам. В графовых базах данных фрагментацию сделать крайне трудно, поскольку они ориентированы не на агрегаты, а на отношения. Поскольку ключевым алгоритмом обработки графовых данных является алгоритм обхода графа, хранение связанных узлов на одном и том же сервере позволяет повысить эффективность обхода графа. Обход графа, узлы которого разбросаны по разным серверам, имеет низкую эффективность. В то же

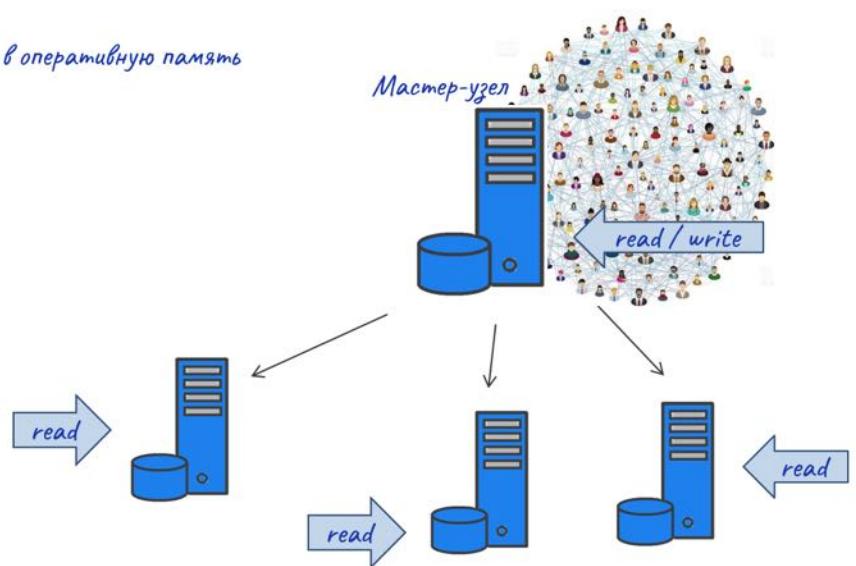
время для некоторых графовых баз (особенно связанных с социальными сетями) характерен лавинообразный рост объемов данных. С учетом этой особенности применяются 3 метода масштабирования графовых баз.

- ✓ Полная загрузка базы в оперативную память



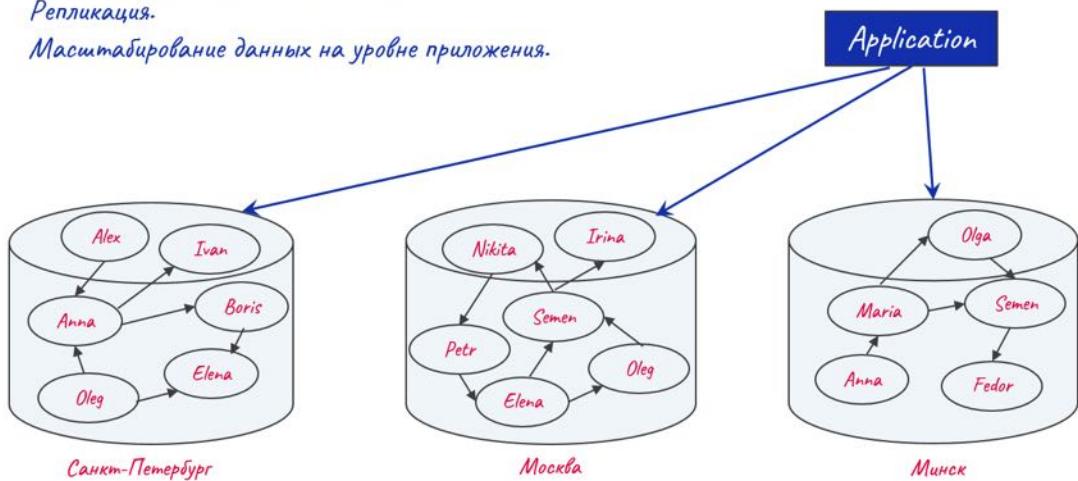
Первый метод – полная загрузка базы в оперативную память. Это в принципе возможно потому, что в настоящее время мощные сервера могут иметь большой объем оперативной памяти. Однако этот метод оказывается полезным только в тех случаях, когда все множество данных, с которым мы работаем, действительно, может поместиться в оперативную память сервера.

- ✓ Полная загрузка базы в оперативную память
- ✓ Репликация



Второй метод – репликация, т.е. выполнение операций записи данных на мастер-узле и добавление подчиненных узлов, обеспечивающих только чтение данных. Такой способ, предусматривающий однократную запись данных и их чтение с многих серверов, хорошо зарекомендовал себя на кластерах и оказался действительно полезным при работе с наборами данных, которые слишком велики, чтобы не помещаться в оперативной памяти одного сервера, но достаточно компактны, чтобы создавать их реплики на разных серверах.

- ✓ Полная загрузка базы в оперативную память.
- ✓ Репликация.
- ✓ Масштабирование данных на уровне приложения.



Третий способ – масштабирование данных на уровне приложения. Если база данных оказывается настолько велика, что оперативно создавать ее реплики оказывается невозможным, то можно выполнить фрагментацию данных на стороне приложения, используя знания о предметной области. Например, узлы социального графа, связанные с Москвой, можно создать на одном сервере, а узлы, связанные с Санкт-Петербургом, - на другом. Однако следует понимать, что в последнем случае узлы хранятся в физически разных базах данных.



*Назначение индексов в графовых базах
– быстрый поиск стартовой точки для обхода графа.*

Пример (Neo4j, язык Cypher)

Создание индекса

CREATE INDEX ON :Station(Name)

Специализированные языки, ориентированные на работу с графами, позволяют не только создавать графы, но и запрашивать свойства узлов, обходить граф и перемещаться по узлам и отношениям. Свойства узлов и отношений можно индексировать. Назначение индексов в графовых базах – быстрый поиск стартовой точки для обхода графа. В приведенном ранее примере базы данных метрополитена имеет смысл построить индекс по полю Name для узлов графа. В следующих примерах будем исходить из того, что такой индекс уже построен.

Пример (Neo4j, язык Cypher)



Поиск узлов, связанных отношением CROSSING с узлом «Гостиный двор»

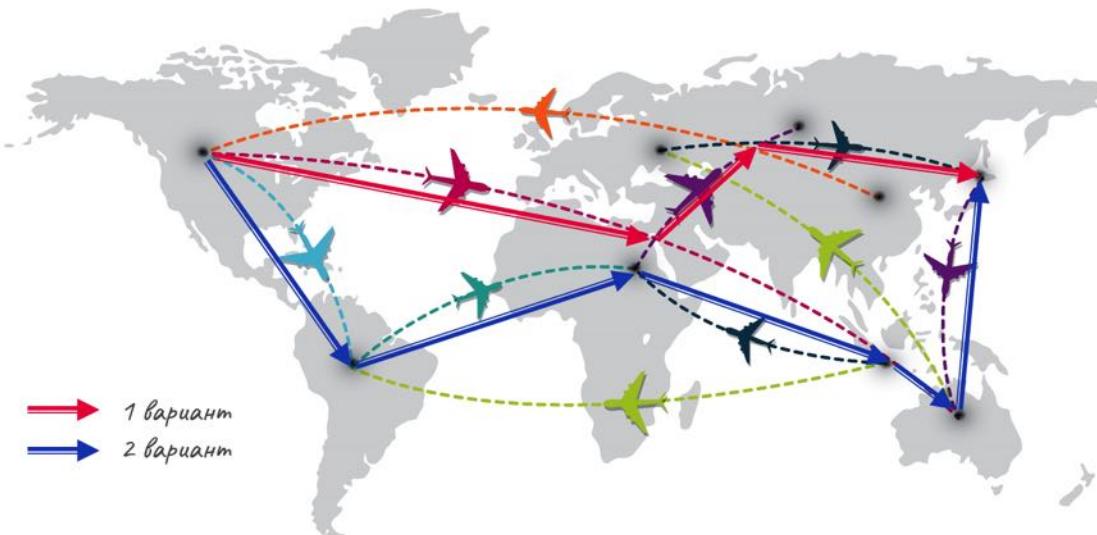
```
MATCH (node:station {name: 'Gostiny Dvor'})-[:CROSSING]->(cross_node)
RETURN node.name, cross_node.name
```

*Поиск узлов, до которых можно добраться из узла «Невский проспект»,
используя связи типа CONNECTION*

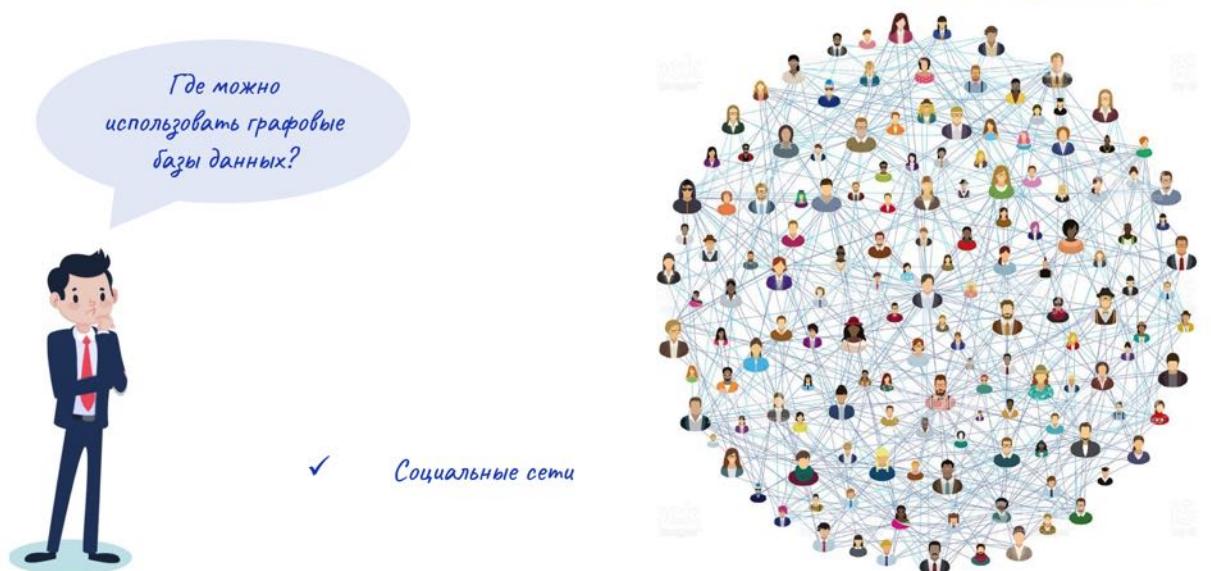
```
MATCH path = (node:station {name: "Nevsky Prospekt"})-[:CONNECTION*]->(end_node)
RETURN node.name, end_node.name, length(path)
```

Рассмотрим, как в языке Cypher выглядит поиск узлов, связанных отношением CROSSING с узлом “Гостиный двор”. Поиск узлов и их прямых отношений не

представляет особого интереса, так как это же можно делать в реляционных базах данных. Настоящая мощь графовых баз данных проявляется в ситуациях, когда необходимо обойти график на произвольную глубину начиная с заданной стартовой точки. Следующий пример демонстрирует, как можно найти все узлы, до которых можно добраться из узла «NevskyProspekt» используя связи типа CONNECTION.



Еще одно из преимуществ графовых баз данных заключается в разнообразии возможностей поиска путей между двумя узлами - можно определить, есть ли несколько путей, найти все пути или кратчайший путь.



Графовые базы данных можно эффективно использовать в социальных сетях. Они могут отражать дружеские связи между людьми и не только. Например,

они могут представлять сотрудников, их профессиональные навыки, а также сотрудничество между коллегами в разных проектах. Любая предметная область с богатыми взаимными связями подходит для описания с помощью графовой модели данных. Графовые базы данных особенно эффективны если в одной и той же базе данных существуют различные отношения между сущностями. Например, родственные, социальные, географические, коммерческие связи и т.п.



Где можно использовать графовые базы данных?

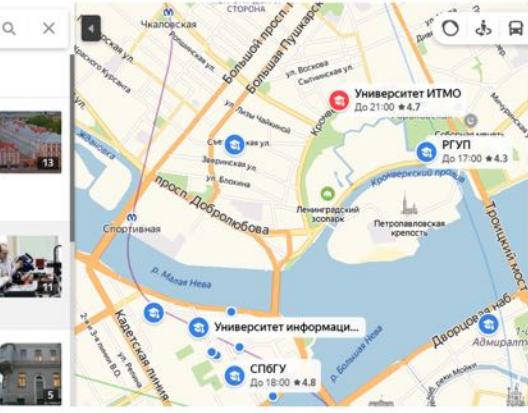
Университет

Открыто Круглосуточно

Санкт-Петербургский государственный университет
ВУЗ - Менделеевская линия, 2

Университет ИТМО
ВУЗ - Кронверкский просп., 49

СПбГИК
ВУЗ - Миллионная ул., 1



УНИВЕРСИТЕТ ИТМО

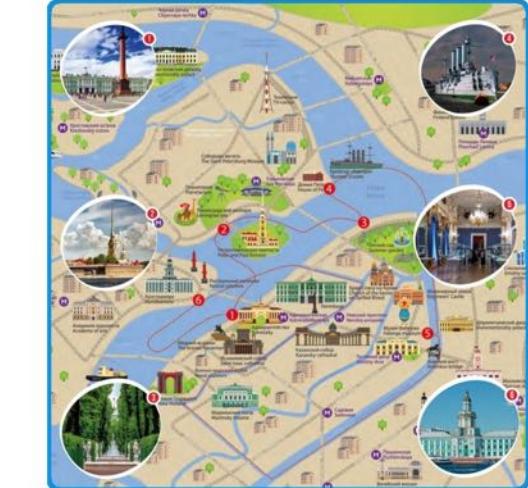
✓ Социальные сети

✓ Геолокационные сервисы

Любую географическую точку можно представить в виде узла графа. Отношения между узлами могут описывать расстояние. Использование таких баз позволяет службам доставки обеспечивать эффективную доставку товаров, а навигационным службам прокладывать оптимальный маршрут.



Где можно использовать графовые базы данных?



✓ Социальные сети

✓ Геолокационные сервисы

✓ Рекомендательные базы данных

Кроме того, снабдив узлы графа соответствующей информацией о характере объекта (ресторан, книжный магазин, фитнес-центр, поликлиника и т.п.) можно в зависимости от запроса выдавать рекомендации о том, куда именно рекомендуется пойти (например, в ресторан с наивысшим рейтингом) и как добраться до нужного объекта в зависимости от текущего положения пользователя.

Хранение в базах данных социальных связей между людьми и сведений о покупках, совершенных в интернет магазинах позволяет использовать их для выдачи рекомендаций типа "ваши друзья также купили этот товар" или "при заказе этого товара обычно также заказывают следующие товары". Кроме того, при заказе отеля можно рассылать рекомендации туристам о том, какие достопримечательности им рекомендуется посетить. Например, "Гости, приезжающие в Санкт-Петербург, обычно посещают Эрмитаж и Петродворец".

Основы Redis

The screenshot shows the official Redis website at <https://redis.io/>. At the top left is the Redis logo. The top right features the text "УНИВЕРСИТЕТ ИТМО" above a grid of small colored dots. The main content area includes a navigation bar with links to Commands, Clients, Documentation, Community, Download, Modules, and Support. Below the navigation is a brief introduction to Redis as an open-source in-memory data structure store. To the left, there's a table comparing Redis with other DBMS systems based on rank, DBMS type, and database model. On the right, there's a "Redis News" section with a list of recent posts and a "More..." link. At the bottom, there's a note about the website being open source and credits to Redis Labs.

Rank	DBMS	Database Model
Sep 2019	1. Redis	Key-value, Multi-model
Aug 2019	2. Amazon DynamoDB	Multi-model
Sep 2018	3. Microsoft Azure Cosmos DB	Multi-model
1. Redis	4. Memcached	Key-value
2. Amazon DynamoDB	5. Hazelcast	Key-value, Multi-model

Redis News

- Redis 5 RC5 and Redis 4.0.11 are out! Details here: <https://t.co/TvfUjOlk8g> Note that 4.0.11 fixes a few important (and very old) bugs.
- Redis 4.0.8 is out! TLDR: a single commit fixing a radix tree bug fixed 10 months ago, but escaped to 4.0. branch me... <https://t.co/9nRw7qDZU>
- Blog post: "An update on Redis Streams development": <https://t.co/GMT0uuCnP>
- Redis 4.0.7 is out, many bugs fixed, none critical, a few important improvements (for instance HyperLogLog dense/rpc...) <https://t.co/Nw4x08GVP>
- @VincenSimon64 if you don't need sharding Sentinel, otherwise Cluster.

More...

This website is open source software. See all credits.

Sponsored by redislabs

В этой лекции мы познакомимся с основами Redis. Почему именно Redis? Да просто потому, что именно его считают наиболее ярким представителем группы "ключ-значение".

Основные характеристики Redis

- ✓ Хранит все данные в оперативной памяти.
- ✓ Периодически сохраняет данные на диске.
- ✓ Представляет данные в виде структур пяти типов.



Часто Redis описывают как хранилище данных *in memory* (то есть в оперативной памяти). Да, действительно, Redis, держит все данные в оперативной памяти, однако периодически сохраняет эти данные на диске. Кроме того, Redis не просто хранит данных типа ключ-значение, а содержит данные гораздо более сложных структур. Данные в Redis могут быть представлены в виде пяти разных структур, и только одна из них, собственно, и есть в чистом виде структура типа ключ-значение. Понимание этих пяти структур, как они работают, какие методы предоставляют для взаимодействия с пользователем, и что можно сделать с их помощью, как раз и являются ключом к пониманию Redis. Но сначала разберемся с тем, какие именно структуры данных возможны в REDIS.

Маша Иванова, Саша Петров, Оля Смирнова, Вася Соколов, Коля Сидоров



Маша	Иванова
Саша	Петров
Оля	Смирнова
Вася	Соколов
Коля	Сидоров

1	Маша
2	Саша
3	Оля
4	Вася
5	Коля
6	Аня

Если обратиться к реляционным базам, то можно сказать, что базы данных предоставляют один универсальный тип структур данных - таблицы. Таблицы одновременно сложные и в то же время гибкие. Их универсальность заключается в том, что любые структуры данных можно смоделировать с помощью таблиц. Тем

не менее, они не идеальны. А именно: их не всегда достаточно просто приспособить для представления данных, и они не всегда достаточно быстро позволяют организовать доступ к требуемым данным.



Именно поэтому и появилась идея вместо универсальной структуры использовать специализированные структуры, ориентированные на конкретные прикладные задачи. Конечно, в этом случае, наверное, найдутся данные, которые мы не сможем представить в виде узкоспециализированных структур, однако для каких-то данных, мы выиграем в простоте и скорости доступа.

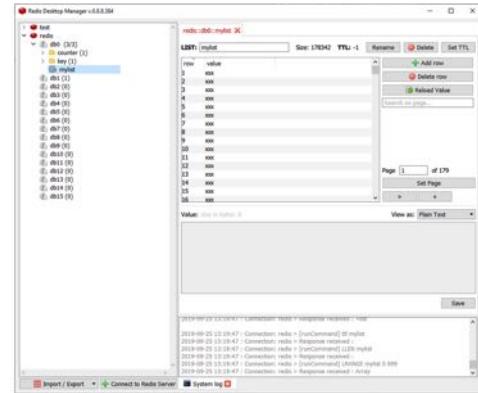
Сравнение структур данных реляционных баз и Redis

Реляционная база	Redis
таблицы	строки
	списки
	хеши
	множества
	упорядоченные множества

Использование специфичных структур данных для специфичных задач? Да, именно в этом и состоит подход Redis. Если в прикладной задаче вам необходимы не таблицы, а скаляры, списки, хеши или множества, то почему бы с самого начала не пытаться представлять их не в виде таблиц, а хранить как скаляры, списки, хеши и множества?

Доступ к Redis

- Загрузка Redis с официального сайта
<https://redis.io/download>

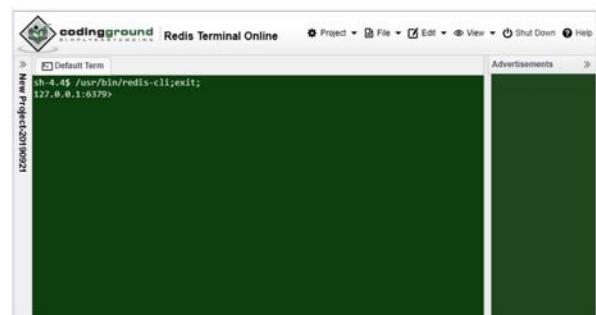


Как и где можно познакомиться со структурами данных и приемами работы в Redis чтобы получить минимальные практические навыки? Можно скачать дистрибутив Redis с официального сайта -<https://redis.io/download> (его адрес вы видите на слайде), установить его на своем компьютере и получить базу Redis в свое полное распоряжение.

Доступ к Redis

- Загрузка Redis с официального сайта
<https://redis.io/download>

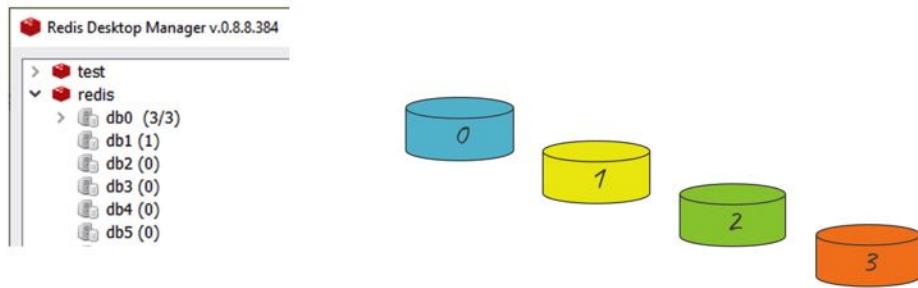
- Redis Terminal Online
https://www.tutorialspoint.com/redis_terminal_online.php



Другой способ, который годится для тех, кто использует Redis в учебных целях,- воспользоваться инструментом RedisTerminalOnline - https://www.tutorialspoint.com/redis_terminal_online.php (его адрес вы также видите на слайде). В этом случае Вы не сможете полностью распоряжаться базой по своему усмотрению, не сможете выполнять некоторые операции (например, конфигурирование базы), но, тем не менее, получите достаточно возможностей, чтобы ознакомиться с основными приемами работы с Redis.

Во всех примерах, которые последуют далее в этой лекции, мы использовали именно инструмент Redis Terminal Online.

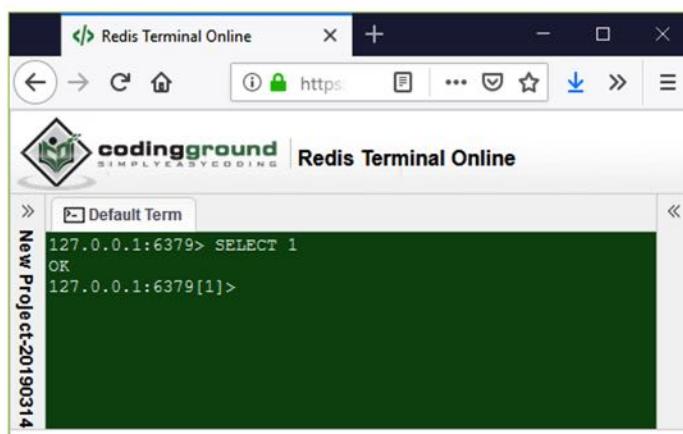
Идентификация баз в Redis



Redis использует знакомую всем концепцию базы данных. База – это набор данных. Типичное предназначение базы данных Redis — это группировка всей информации определенного приложения в одном месте и изоляция ее от других приложений.

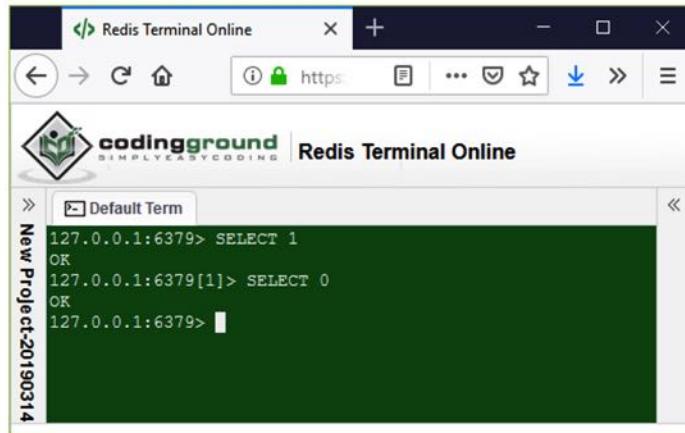
В Redis база данных идентифицируется целым числом, которое по умолчанию равняется 0 (а соответствующая база называется базой по умолчанию). Если необходимо сменить базу данных, то можно это сделать командой SELECT. Например, для переключения на базу с идентификатором 1 достаточно в командной строке ввести команду: SELECT 1.

Переключение баз в Redis

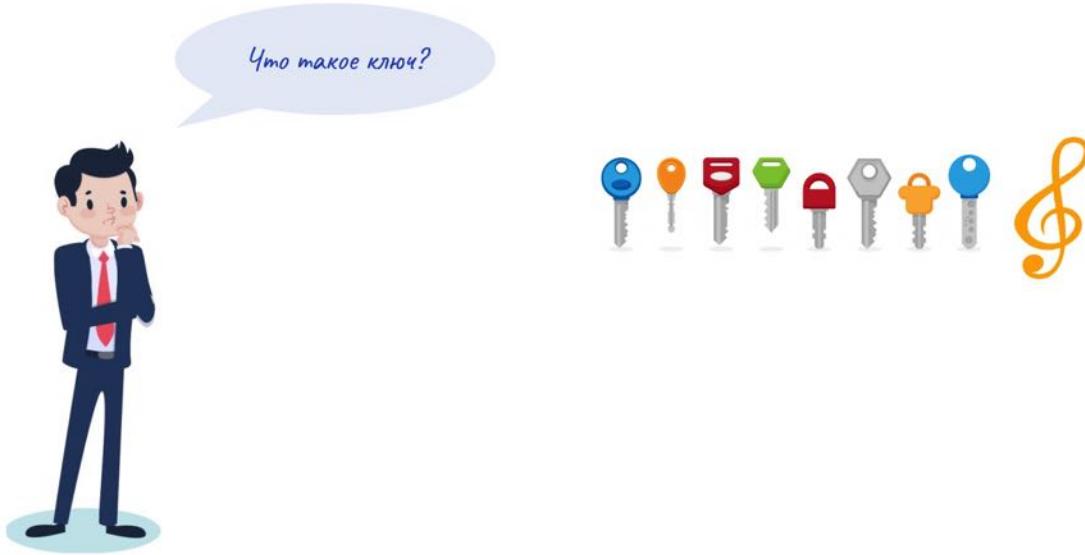


Redis должен ответить сообщением OK, а в терминале вы должны увидеть что-то типа того, что видите сейчас на слайде.

Доступ к Redis



Если требуется переключиться обратно на базу по умолчанию, просто введите в командной строке: SELECT 0.



Одним из основополагающих понятий Redis является понятие ключа. Несмотря на то, что Redis больше, чем просто хранилище типа ключ-значение, в его основе каждая из пяти используемых структур обязательно имеет ключ и значение. Поэтому очень важно разобраться в том, что такое ключи и что такое значения, перед тем как двигаться дальше.



Ключ - это идентификатор, который позволяет получить доступ к ассоциированному с ним значению. Ключ должен быть уникальным в базе.

Итак, что собой представляет ключ в паре “ключ-значение”? Прежде всего, он должен быть уникальным в базе. Это идентификатор, который позволяет получить доступ к ассоциированному с ним значению. Теоретически он может быть любым. Однако в практических реализациях всегда есть ограничения, связанные, как минимум, с его размером.

Примеры ключей



В качестве ключа можно использовать любую последовательность символов - от короткой строки текста до содержимого файла изображения. Даже пустая строка является допустимым ключом. Однако из соображений производительности следует избегать слишком длинного ключа. В хранилищах типа "ключ-значение" большое внимание уделяется выбору структуры самого ключа. Ключи можно генерировать с помощью специальных алгоритмов и задавать явно. В качестве ключа может выступать идентификатор пользователя, электронный адрес, номер телефона и т.п. Ключ можно формировать из даты, времени и идентификаторов приложений, работающих с базой.

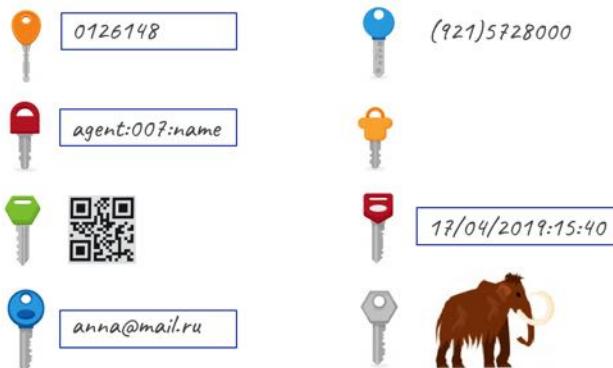
Примеры ключей

<тип_объекта:идентификатор_объекта:атрибут>

Часы:Секундомер_A15:Цена



Еще одна идея – формировать ключи в виде следующей структуры: <тип объекта:идентификатор объекта:атрибут>. Таким образом можно сохранять в хранилищах “ключ-значение” всю информацию об объектах и получать доступ к значениям атрибутов объектов. С помощью специальных методов можно задавать интервал времени, по истечении которого ключ станет недействительным. Последнее свойство особенно удобно для работы с сессионными объектами и корзинами покупателей.

Примеры ключей

На слайде Вы можете видеть примеры ключей. Они достаточно разнообразны. Это и просто последовательность символов, и имя агента 007, и электронный адрес, и номер телефона и даже простые дата и время. Любая из этих структур может быть ключом.

Примеры значений

Данные в Redis могут быть представлены в виде пяти разных структур, и только одна из них в чистом виде структура типа ключ-значение. Понимание этих пяти структур, как они работают, какие методы предоставляют для взаимодействия с пользователем, и что можно сделать с их помощью, как раз и является ключом к пониманию Redis.

Рис. 1. Схема движения.



3,1415926

Elena	(856) 759-0123
-------	----------------

`RETURN node.name, end_node.name, length(path)`

Значением в хранилище “ключ-значение” может быть что угодно, Например, длинный или короткий текст, число, программный код, изображение и т. п. Значение также может быть списком, множеством или даже другой парой ключ-значение, инкапсулированной в объект.

Некоторые хранилища позволяют указывать тип данных для значения. Например, можно указать, что значение должно быть целым числом. Другие хранилища не предоставляют эту функциональность и, следовательно, значение может быть любого типа. Однако не все команды применимы к значениям любого типа. Например, команда, которая увеличивает значение на 1, подразумевает, что ее операндом может быть только численное значение.

Справочник телефонов

Ключ	Значение
	(987) 446-7890
	(955) 567-8301
	(945) 678-9012
	(856) 759-0123

Рассмотрим некоторые примеры ассоциированных ключей и значений. Очень хорошо известный нам пример – справочник телефонов (или как сейчас говорят – контактов). Ключ-имя, а значение – номер телефона.

Мультфильмы

Ключ	Значение
cartoon:1:name	The Lion King
cartoon:1:year	1994
cartoon:1:views	5495742
cartoon:2:name	The Little Mermaid
cartoon:2:year	1989
cartoon:2:views	3295423



Еще один возможный пример – коллекция мультфильмов. Здесь ключ организован по принципу <тип объекта:идентификатор:атрибут> и в приведенном примере легко разобраться где хранится название мультфильма, где год выпуска, а где – количество просмотров.

Основные команды Redis

`SET <ключ><значение>` – задать значение по ключу

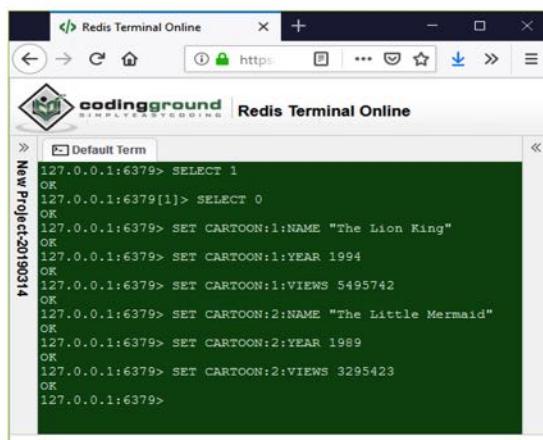


`GET<ключ><значение>` – выбрать значение по ключу



Для хранилищ типа “ключ-значение” характерно использование языков запросов низкого уровня (не декларативных). Запросы в хранилищах типа “ключ-значение” могут выполнять команды по ключу и ничего более. Результатом запроса является все значение. Часть значения вытащить с помощью запроса невозможно. Даже, если значением является слабоструктурированный документ JSON-формата, вытащить отдельные атрибуты документа на уровне запроса не удастся. Анализ выбранного документа может быть произведен только на уровне приложения.

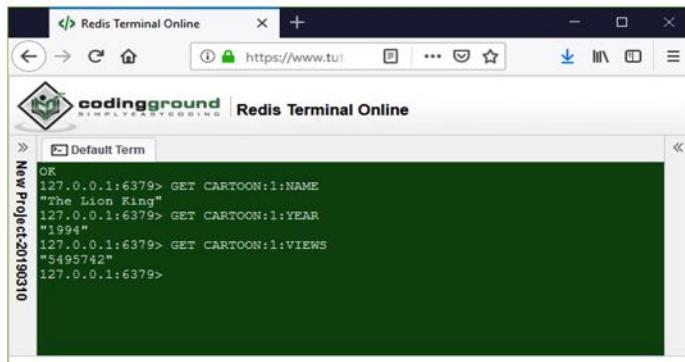
Создание пар ключ-значение с информацией по мультфильмам



```
Redis Terminal Online
https://redis.codingground.ru/
Default Term
127.0.0.1:6379> SELECT 1
OK
127.0.0.1:6379[1]> SELECT 0
OK
127.0.0.1:6379> SET CARTOON:1:NAME "The Lion King"
OK
127.0.0.1:6379> SET CARTOON:1:YEAR 1994
OK
127.0.0.1:6379> SET CARTOON:1:VIEWS 5495742
OK
127.0.0.1:6379> SET CARTOON:2:NAME "The Little Mermaid"
OK
127.0.0.1:6379> SET CARTOON:2:YEAR 1989
OK
127.0.0.1:6379> SET CARTOON:2:VIEWS 3295423
OK
127.0.0.1:6379>
```

И, тем не менее, этих, казалось бы, весьма ограниченных, возможностей для многих приложений вполне хватает. На слайде приведены несколько запросов в СУБД Redis. Составным ключам мультфильм-идентификатор-атрибут сопоставляются имена, годы выпуска и количество просмотров, соответствующих мультфильмам. Сопоставление производится с помощью команды SET.

Выборка атрибутов первого мультфильма



Redis Terminal Online | https://www.tu...

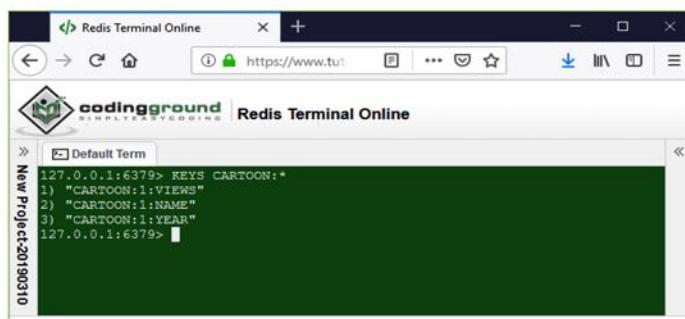
codingground | Redis Terminal Online

```
OK
127.0.0.1:6379> GET CARTOON:1:NAME
"the Lion King"
127.0.0.1:6379> GET CARTOON:1:YEAR
"1994"
127.0.0.1:6379> GET CARTOON:1:VIEWS
"5495742"
127.0.0.1:6379>
```

New Project 20190310

Есть и обратная операция. Ее действие также продемонстрировано на слайде. Это команда GET. Она позволяет достать из хранилища значение, соответствующее ключу. Можно достать имя мультфильма и любые другие значения.

Выборка всех ключей с префиксом «CARTOON»



Redis Terminal Online | https://www.tu...

codingground | Redis Terminal Online

```
127.0.0.1:6379> KEYS CARTOON:*
1) "CARTOON:1:VIEWS"
2) "CARTOON:1:NAME"
3) "CARTOON:1:YEAR"
127.0.0.1:6379>
```

New Project 20190310

Кроме того, в списке основных команд есть команда KEYS. Она позволяет выбирать значения ключей в соответствии с заданным шаблоном. На слайде вы можете видеть, как выбираются все ключи с префиксом CARTOON.

Профиль пользователей приложения

Ключ (username)	Значение				
Catarina	(987) 446-7890		UTC+3	rus	red
Annet	(955) 567-8301		UTC+1	fre	green
Adelaida	(945) 678-9012		UTC+3	rus	green
Elena	(856) 759-0123		UTC-5	eng	yellow

Где используются хранилища типа ключ-значение?



Где используются хранилища типа ключ-значение? Например, для хранения профилей пользователей приложения. Почти каждый пользователь приложения имеет уникальный атрибут userid, username или какой-то другой идентификатор, а также параметры окружения клиентской сессии, например, язык, цвет, часовой пояс и т.д. Все это можно поместить в один объект базы данных и получать параметры окружения пользователя с помощью одной операции GET.

Корзины заказов

Ключ (username)	Значение				
Catarina	1 * 120,00	1 * 250,50			
Annet	1 * 199,99	1 * 219,99	1 * 8,99		
Adelaida	2 * 21,75				
Elena	1 * 35,00	15 * 3.00			

Где используются хранилища типа ключ-значение?



Еще один распространенный способ использования – коммерческие сайты, связанные с корзинами заказов. Вся информация о покупках размещается в одном объекте с ключом userid. Именно для таких случаев хранилища ключ-значение являются очень хорошим решением.

Основы MongoDB



УНИВЕРСИТЕТ ИТМО

<https://www.mongodb.com/>

The database for modern applications

MongoDB is a general purpose, document-based, distributed database built for modern application developers and for the cloud era. No database is more productive to use.

Try MongoDB free in the cloud

Your Work Email

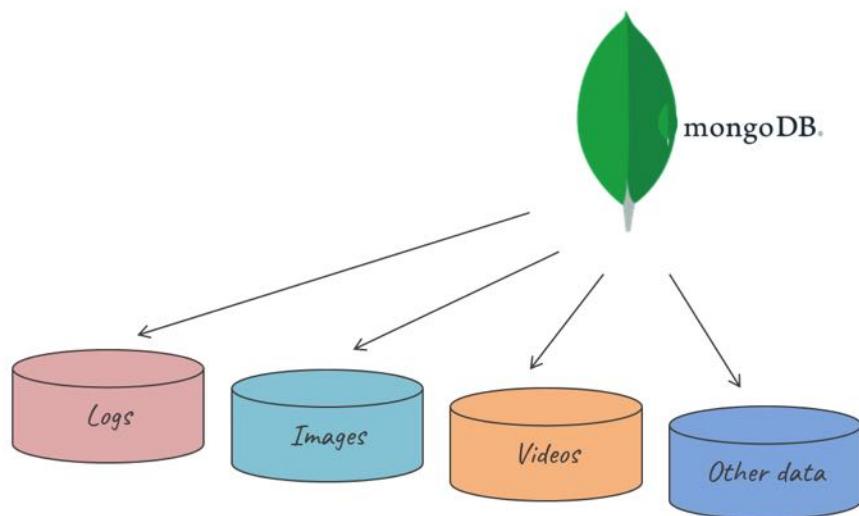
Get Started

Used by millions of developers to power the world's most innovative products and services

Rank	Sep 2019	Aug 2019	Sep 2018	DBMS
1.	1.	1.	1.	MongoDB
2.	2.	2.	2.	Amazon DynamoDB
3.	3.	3.	3.	Couchbase
4.	4.	4.	4.	Microsoft Azure Cosmos DB
5.	5.	5.	5.	CouchDB

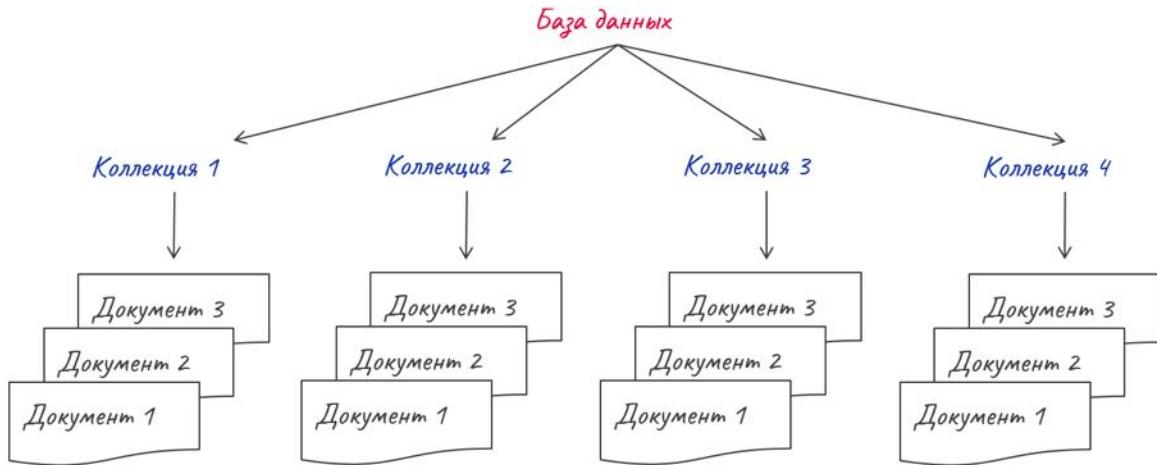
В этой лекции мы познакомим слушателей с основами документного хранилища данных - системой MongoDB. Почему именно MongoDB? Потому, что именно MongoDB (согласно упомянутому ранее рейтингу) принято считать наиболее ярким представителем группы документных хранилищ.

УНИВЕРСИТЕТ ИТМО



MongoDB представляет собой систему для управления документно-ориентированными базами данных, которая реализует новый подход к построению баз данных, где нет таблиц, схем, запросов SQL, внешних ключей и многих других объектов, присущих традиционным реляционным базам данных. MongoDB в некоторых случаях работает быстрее, обладает лучшей масштабируемостью, и, наконец, ее легче использовать. При этом важно понимать, что задачи бывают разные и методы их решения бывают разные. В какой-то ситуации MongoDB действительно улучшит производительность приложения, например, если надо хранить сложные по структуре данные. В другой же ситуации лучше будет

использовать традиционные реляционные базы данных. Вся система MongoDB может представлять не только одну базу данных, находящуюся на одном физическом сервере. Особенность архитектуры MongoDB позволяет расположить несколько баз данных на одном или нескольких физических серверах, и эти базы данных смогут легко обмениваться данными и даже сохранять целостность.

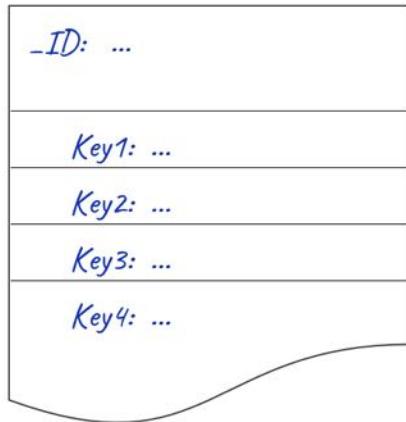


Базу данных MongoDB можно представить в виде иерархической структуры как это указано на слайде. Если в реляционных базах данные хранятся в таблицах, то в MongoDB база данных состоит из коллекций. Каждая коллекция имеет свое уникальное имя - произвольный идентификатор, состоящий не более чем 128 различных алфавитно-цифровых символов и знака подчеркивания. Коллекции в свою очередь состоят из документов. Именно документы хранят полезную информацию.

MongoDB	Реляционные базы
База данных	База данных
Коллекция	Таблица
Документ	Строка (запись) в таблице

В некотором смысле документы подобны строкам в реляционных базах, где строки хранят информацию об отдельных объектах. В отличие от строк документы могут хранить сложную по структуре информацию, которую принято называть слабоструктурированной.

Документ



Документ можно представить как хранилище ключей и значений. Ключ представляет собой простую метку, с которой ассоциирован определенный фрагмент данных из документа. Для каждого документа имеется свой уникальный ключ, который называется `_id`. И если явным образом не указать его значение, то MongoDB сгенерирует для него значение автоматически.

Формат хранения данных

JSON - Java Script Object Notation

BSON - Binary Java Script Object Notation

Как уже упоминалось ранее, одним из популярных стандартов обмена данными и их хранения является формат **JSON** (JavaScript Object Notation). JSON эффективно описывает сложные по структуре данные. Способ хранения данных в MongoDB в этом плане похож на JSON, хотя формально JSON не используется. Для хранения в MongoDB применяется формат, который называется **BSON** (БиСон) или сокращение от binary JSON. Формат BSON позволяет работать с данными быстрее: быстрее выполняется поиск и обработка. Хотя надо отметить, что BSON в отличие

от хранения данных в формате JSON имеет небольшой недостаток: в целом данные в JSON-формате занимают меньше места, чем в формате BSON, но с другой стороны, данный недостаток окупается скоростью. На слайде Вы можете видеть пример документа в формате JSON.

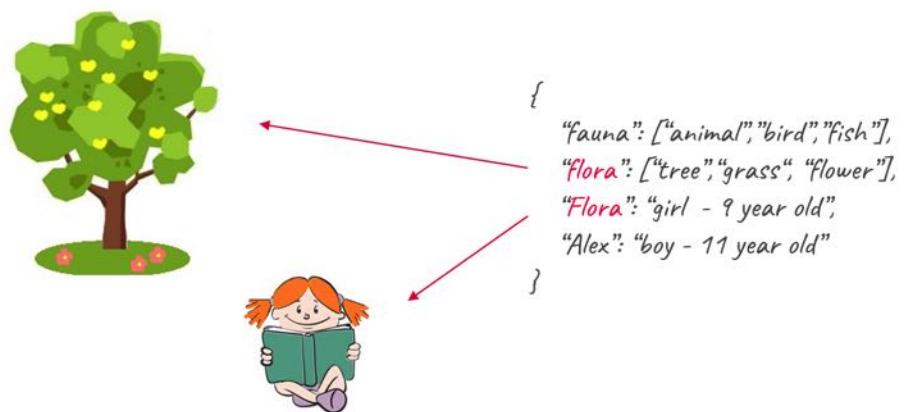
Пример документа MongoDB

```
{  
    "_id": 1,  
    "CartoonName": "The Lion King",  
    "Year": "1994",  
    "Country": "USA",  
    "Duration": "88",  
    "FilmDirectorName": ["Roger Allers", "Rob Minkoff"],  
    "SongWriterName": ["Elton John", "Hans Zimmer"]  
}
```



Всему документу соответствует уникальный идентификатор, т.е. ключ `_id` со значением, равным 1. Отдельные поля в документе также представлены в виде набора пар ключ-значение. Например, ключу `"CartoonName"`: соответствует одно значение `"TheLionKing"`, а ключу `"FilmDirectorName"` массив значений: `["RogerAllers", "RobMinkoff"]`.

Пример различающихся ключей



Следует обратить внимание на то, что ключи в MongoDB являются регистрозависимыми. Два ключа на приведенном слайде различаются.

Типы данных

<i>String</i>	строковый тип данных (для строк используется кодировка UTF-8)
<i>Integer</i>	используется для хранения целочисленных значений
<i>ObjectID</i>	тип данных для хранения <i>id</i> документа
<i>Array</i>	тип данных для хранения массивов элементов
<i>Boolean</i>	булевый тип данных, хранящий логические значения <i>TRUE</i> или <i>FALSE</i>
<i>Regular expression</i>	применяется для хранения регулярных выражений
...	

Ключи идентифицируют значения. Значения же могут различаться по типу данных. В приведенном ранее примере "CartoonName" соответствует строковому значению, «Year» – целочисленному, "FilmDirectorName" – массиву строковых значений. На слайде приведены наиболее используемые типы значений, среди которых вы можете видеть: строки, целые значения, логические, массивы и даже регулярные выражения.

Пример документа с различающимися значениями

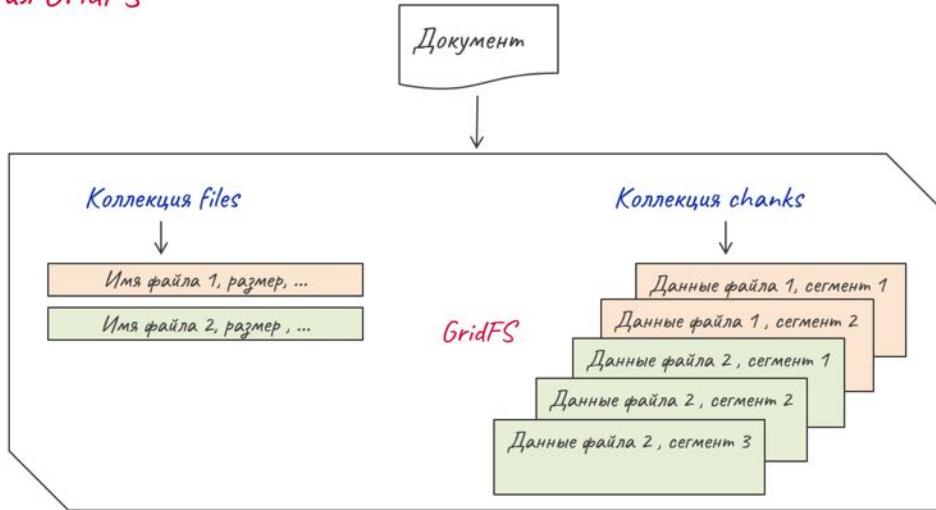
{ "Duration": "88" } - строка

{ "Duration": 88 } - число

Следует обратить внимание, что значения в документах обладают строгой типизацией, например, следующие два документа не будут идентичными. Если в

в первом случае для ключа "Duration" определена в качестве значения строка, то во втором случае значением является число. И именно поэтому значения этих ключей считаются различными.

Технология GridFS



Система MongoDB ориентирована на хранение документов. Однако эти документы иногда оказываются документами очень большого размера. MongoDB позволяет сохранять данные большого размера, однако при этом размер одного документа ограничивается 16 Мб. Но в MongoDB имеется специальное решение - технология **GridFS**, которая позволяет хранить данные по размеру больше, чем 16 Мб. Суть технологии GridFS заключается в использовании двух специальных коллекций. В первой коллекции, которая называется *files*, хранятся имена файлов, а также их метаданные, например, размер. А в другой коллекции, которая называется *chunks*, в виде небольших сегментов хранятся данные файлов, обычно сегментами по 256 Кб.

Язык запросов

Команда Выборки

```
db.<collection-name>.find([data_filter[projection]])
```

data_filter – конструкция, которая задает условие для выборки данных из коллекции

projection – конструкция, которая задает поля коллекции для выборки

Выборка данных из коллекций в MongoDB осуществляется при помощи команды `find`. Действие этой команды, в конечном счете, во многом аналогично обычному запросу `select` в реляционной базе. Так же, как и в команде `select`, ключевые моменты запроса задаются с помощью условия выборки (т.е. фильтра) и проекции. И, тем не менее, синтаксически они отличаются. На слайде приведен синтаксис команды `find`.

Язык запросов

Выборка всех данных из коллекции `student`

MongoDB	<code>db.student.find()</code>
SQL	<code>select * from student</code>

Приведем пример простейшего варианта использования команды `find`. Например, чтобы извлечь все документы из коллекции `student`, можно использовать команду
`db.student.find()`

В языке SQL этой команде бы соответствовал запрос

`select * from student`

Никакой фильтр, ни проекция в данном запросе не используется.

В приведенном примере мы рассмотрели выборку всех документов, однако, если нам надо получить не все документы, а только те, которые удовлетворяют

определенному условию, то придется использовать в команде фильтр для отбора требуемых документов.

Язык запросов

Выборка из коллекции *student* 19-летних студентов

MongoDB	<code>db.student.find({age:19})</code>
SQL	<code>select * from student where age = 19</code>

Например, выведем все документы (из коллекции *student*), для которых будет задан фильтр `{age:19}`. То есть то, что в реляционных базах выводилось бы запросом `select * from student where age = 19`. В MongoDB для этого придется написать следующий запрос: `db.student.find({age:19})`. Проекция в запросе, по-прежнему, не используется.

Язык запросов

Выборка 19-летних студентов по имени Semen

MongoDB	<code>db.student.find({age:19, name:"Semen"})</code>
SQL	<code>select * from student where age=19 and name="Semen"</code>

Усложним запрос и выведем только 19-летних студентов по имени Semen. В запросе придется задать более сложный фильтр для данных: `{age:19, name: "Semen"}`. В реляционной базе этому запросу соответствовал бы запрос вида: `select * from student where age = 19 and name = "Semen"`. То есть запятая в фильтре интерпретируется как логическая операция `and`.

Документ может иметь множество полей, но не все эти поля могут быть нужны в результате запроса. В этом случае в операторе выборки необходимо использовать проекцию, позволяющую указать требуемые в конечной выборке поля.

Язык запросов

Выборка имен 19-летних студентов

MongoDB	<code>db.student.find({age:19},{name:1})</code>
SQL	<code>select _id, name from student where age=19</code>

Например, запрос на следующем слайде выводит только имена 19-летних студентов. Единица напротив названия поля в проекции указывает на то, что соответствующее поле должно присутствовать в выборке. При этом одно поле, а именно `_id` по умолчанию будет выведено даже без указания соответствующей единицы, указывающей на необходимость его присутствия.

Поэтому, если мы не хотим видеть поле `_id` в выборке, то надо явным образом упомянуть его в проекции и сопоставить ему значение 0: `{_id:0}`.

✓ Системы регистрации событий

и	е	о
с	г	б
м	и	в
е	с	м
н	н	и
в	р	й
	а	
	ц	
	и	
	и	

Где используются
хранилища типа
ключ-значение?



Где могут использоваться документные хранилища? Существует множество разнообразных приложений, желающих регистрировать разного рода события (**системы регистрации событий - Event registration system**). Документные базы данных могут хранить все эти типы событий и действовать как центральное хранилище. Это особенно важно в ситуациях, когда тип данных, собираемых событиями, постоянно изменяется. События могут группироваться по имени приложения, породившего их, или по типу событий.

- ✓ Системы регистрации событий
- ✓ Системы управления информационным накоплением

и	п	н	а
с	р	ф	к
т	а	о	о
е	в	р	п
м	л	м	л
ы	е	а	е
н	ц	н	
и	и	и	
я	о	е	
	н	м	
	н		
ы			
м			

Где используются
хранилища типа
ключ-значение?



Кроме того, документные системы хорошо работают в системах управления информационным наполнением (**системы управления информационным накоплением – Content management systems**) или в приложениях, связанных с публикацией веб-сайтов, которым требуется управлять комментариями пользователей, их регистрацией, профилями и т.п.

- ✓ Системы регистрации событий
- ✓ Системы управления информационным накоплением
- ✓ Аналитические системы реального времени

н	и	е	р
а	с	а	е
л	м	л	м
и	е	б	е
т	м	н	н
и	bl	о	и
ч		г	
е		о	
с			
к			
и			
е			

Где используются
хранилища типа
ключ-значение?



И, наконец, документные базы данных широко используются для анализа данных в реальном времени (**Аналитика в режиме реального времени - Real-time analytics**); поскольку части документов можно выбирать, обновлять и агрегировать с применением технологии MapReduce.

Введение в Cassandra

В этом и последующих фрагментах данной лекции мы обсудим модель данных в СУБД Cassandra и язык запросов Cassandra Query Language (CQL). Почему именно Cassandra? Да просто потому, что это хранилище данных считают наиболее ярким представителем группы колоночных хранилищ.

Язык запросов



CQL во многих отношениях напоминает SQL, но, тем не менее, имеет ряд важных отличий. Для начала познакомимся с терминологией, используемой в СУБД Cassandra.

Сравнение реляционных баз и Cassandra

Реляционные базы	Cassandra
Данные хранятся в таблицах	Данные хранятся в таблицах
В таблицах хранятся значения	Хранятся значения только столбцов
всех столбцов	с непустыми значениями
Жесткая структура таблиц	Гибкая структура таблиц

На первый взгляд, Cassandra кажется похожей на обычные реляционные СУБД. Например, в Cassandra тоже есть таблицы. Однако, в Cassandra таблицей называется специальная логическая структура, объединяющая похожие данные. Например, могут быть таблицы, представляющие группу пользователей, студентов, отелей и т.п. В этом смысле таблица Cassandra аналогична реляционной таблице. Принципиальное отличие состоит в том, что в Cassandra нет необходимости хранить значения каждого столбца (или поля) при сохранении новой сущности. Возможно, значения некоторых столбцов неизвестны. Например, у одних людей есть второй номер телефона, а у других -нет. Или в форме на странице сайта, реализованного с использованием Cassandra, одни поля обязательны, а другие - нет. Это нормально. Вместо того чтобы хранить значение **null** в качестве признака отсутствия значения и расходовать на это место, Cassandra вообще не хранит соответствующее значение столбца для такой строки. При проектировании таблицы в традиционной реляционной базе данных мы обычно имеем дело с «сущностями», т. е. наборами атрибутов, описывающих некоторый объект. Мы не задумываемся о размере строк, потому что после того, как решение о том, как объект представляется в виде таблицы, размер строки уже не подлежит обсуждению. Но при работе с Cassandra размер строки фиксирован не так жестко: строка может быть широкой или узкой в зависимости от количества присутствующих в ней столбцов.

Первичный ключ

Ключ раздела (*partition key*) +
кластерные столбцы (*clustering columns*)



В Cassandra для представления широких строк, называемых также разделами (*partition*), используется специальный первичный ключ. Его иногда называют составным ключом (*compound key*). Составной ключ состоит из ключа раздела (*partitionkey*) и необязательного набора кластерных столбцов (*clustering column*). Ключ раздела (*partition key*) служит для определения узлов распределенной системы, на которых хранятся строки. Ключ раздела может состоять из нескольких столбцов. Кластерные же столбцы используются для управления сортировкой данных при хранении внутри каждого раздела.

Основные структуры данных в Cassandra

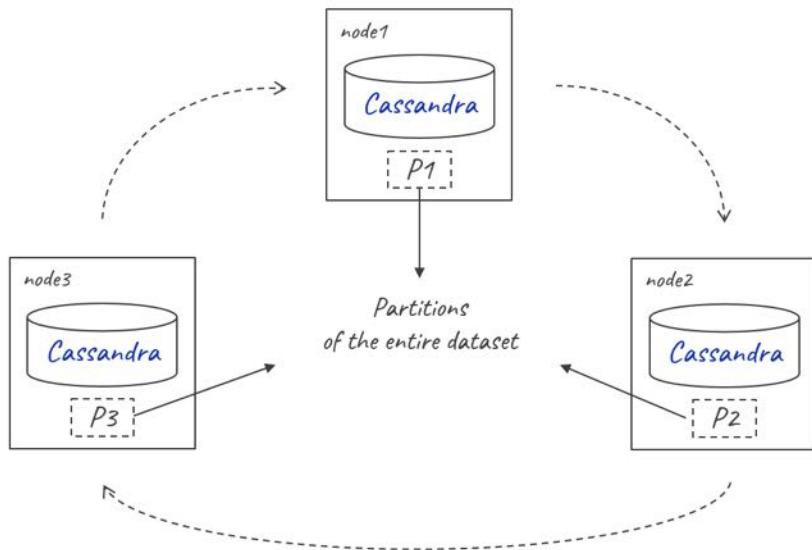
столбец	пара ключ-значение
строка	контейнер столбцов, на который можно сослаться по первичному ключу
таблица	контейнер строк
пространство ключей	контейнер таблиц
кластер	контейнер пространств ключей, расположенных в одном или нескольких узлах

Основными структурами данных в Cassandra являются: столбец, строка, таблица, пространство ключей и кластер. Познакомимся с ними поближе. Итак, столбец в

понимании Cassandra — это пара ключ-значение; строка — это контейнер столбцов, на который можно сослаться по первичному ключу; таблица — контейнер строк; пространство ключей — контейнер таблиц, а кластер — контейнер пространства ключей, расположенных в одном или нескольких узлах сети.

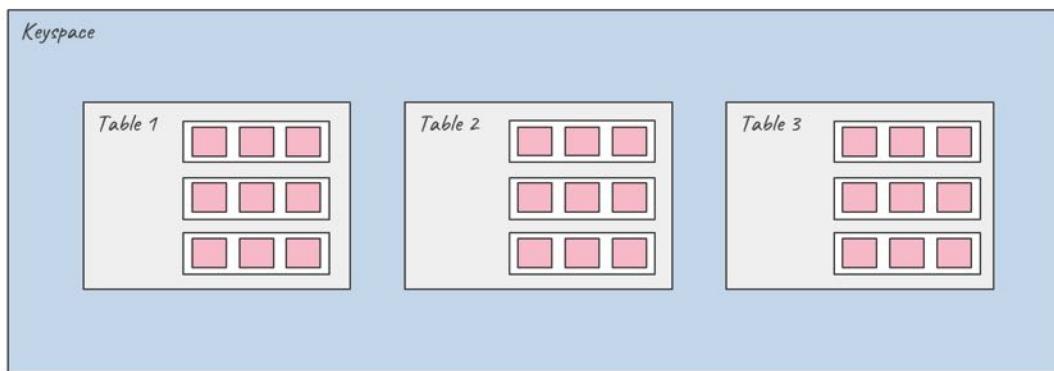
Это взгляд снизу вверх на модель данных Cassandra. Познакомившись с основными терминами, перейдем к детальному изучению каждой структуры.

Кластер



База данных Cassandra изначально спроектирована для распределения данных между несколькими серверами, которые обрабатываются совместно и представляются пользователю единым целым. Поэтому самая внешняя структура в Cassandra называется кластером, или кольцом, потому что Cassandra распределяет данные между узлами кластера, считая их закольцованными.

Пространство ключей



Кластер — это контейнер пространства ключей. Пространство ключей (keyspace) - самый внешний контейнер данных в Cassandra, во многом аналогичный реляционной базе данных. Как база данных является контейнером таблиц в реляционной модели, так пространство ключей - контейнер таблиц в модели данных Cassandra. Как и у реляционной базы данных, у пространства ключей есть имя и набор атрибутов, определяющих его поведение в целом.

Таблицы и строки

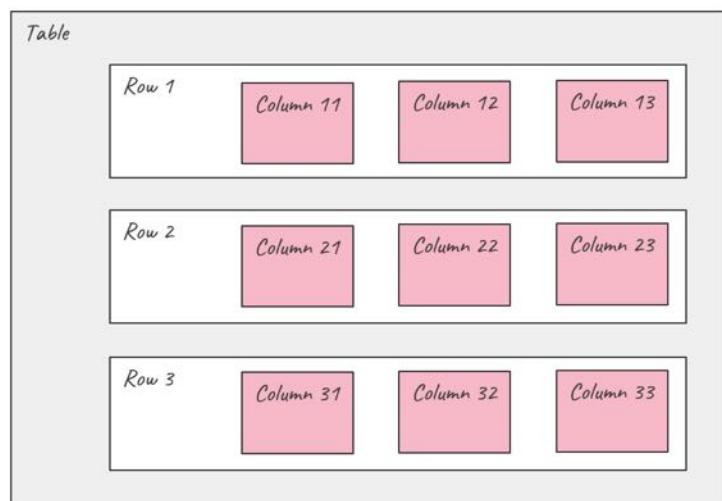


Таблица — это контейнер для упорядоченной коллекции строк, каждая из которых, в свою очередь, представляет собой упорядоченную коллекцию столбцов. Порядок определяется столбцами, назначенными в качестве ключей. ВCassandra могут существовать и дополнительные ключи, помимо первичного. При записи данных в таблицу задаются значения одного или нескольких столбцов. Эта коллекция значений называется строкой. Хотя бы одно из заданных значений должно быть первичным ключом, который играет роль уникального идентификатора строки.



Особенности проектирования модели данных Cassandra



Отсутствуют соединения

Как проектируются модели данных в Cassandra? Прежде, чем начнем проектировать модель данных для Cassandra, обратим внимание на ключевые особенности проектирования. В Cassandra нет операций соединения. Соединение таблиц, если оно необходимо, приходится производить на стороне клиента, либо создавать дополнительную (т.е. денормализованную) таблицу для хранения результатов виртуального соединения. При создании моделей данных для Cassandra предпочтителен второй вариант. Соединение на стороне клиента применяется очень редко, лучше продублировать (т.е. денормализовать) данные.

Особенности проектирования модели данных Cassandra



Отсутствуют соединения



Отсутствует ссылочная целостность

Несмотря на поддержку облегченных транзакций и пакетов совместно выполняемых команд, в Cassandra нет понятия ссылочной целостности таблиц. В реляционной базе данных мы можем определить в таблице внешний ключ для ссылки на первичный ключ в другой таблице. В Cassandra такого механизма нет.

Особенности проектирования модели данных Cassandra



Отсутствуют соединения



Отсутствует ссылочная целостность



Допустима денормализация данных

При проектировании реляционной базы данных чрезвычайно важным фактором считается нормализация таблиц (т.е. чтобы данные в таблицах не повторялись многократно и т.п.). Но при работе с Cassandra это вовсе не считается преимуществом, потому что оптимальная производительность в Cassandra достигается тогда, когда модель данных денормализована. Порою и в реляционной базе приходится идти на денормализацию, и тому есть две причины.

Особенности проектирования модели данных Cassandra



Отсутствуют соединения



Отсутствует ссылочная целостность



Допустима денормализация данных

Причины денормализации данных

1. Производительность
 2. Структура документа, нуждающегося в длительном хранении

Первая - производительность. Разработчику программного обеспечения попросту не удается добиться нужной производительности, если необходимо соединять накопившиеся за много лет данные, поэтому он производит денормализацию, подстраиваясь под нужды известных запросов. Этот подход работает, но идет вразрез с идеологией реляционной модели, так, что, в конце концов, возникает вопрос, а является ли реляционная база оптимальным решением в данных условиях. Вторая причина намеренной денормализации - структура документа, нуждающегося в длительном хранении. То есть имеется основная таблица, которая ссылается на много внешних таблиц, данные в которых со временем изменяются, однако необходимо сохранить для истории документ в том виде, в котором он существовал в момент создания.

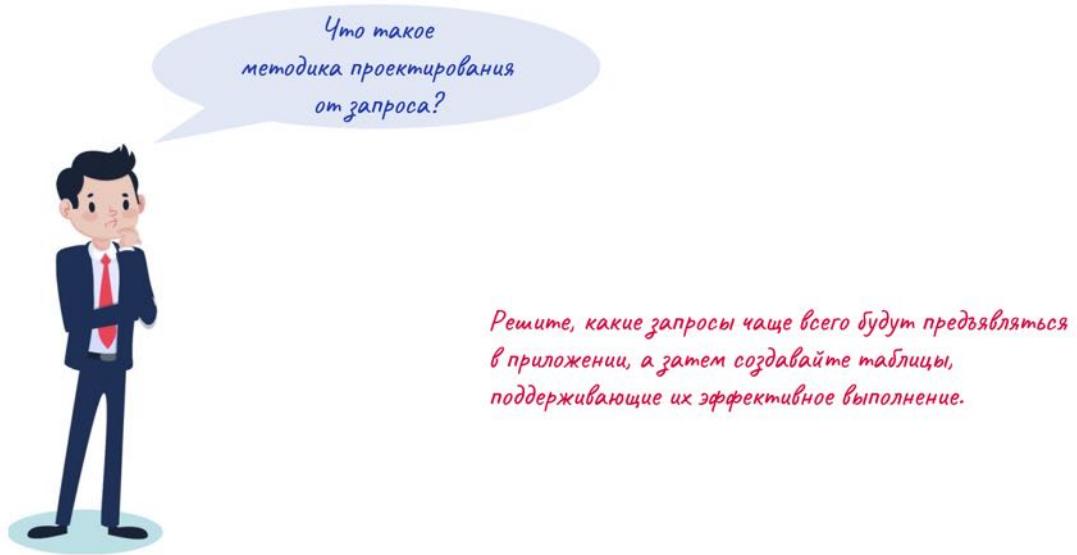
Типичный пример - счета-фактуры. У нас уже есть таблицы заказчиков и продуктов, и, казалось бы, счет-фактура может просто сослаться на эти таблицы. Но на практике так никогда не делается. Информация о заказчике или ценах может измениться, да сама форма и количество полей в счете фактуры может измениться и тогда будет нарушена целостность счета-фактуры в том виде, в котором она существовала на момент создания. А это приведет к ошибкам при аудите, в отчетах, к нарушению законов и прочим проблемам. В реляционном мире денормализация нарушает классические принципы Кодда, и, поэтому, этого стараются избегать. Но в Cassandra денормализация – типичный прием и считается совершенно нормальным делом. Если модель простая, то денормализовывать ее не обязательно. Но и бояться этого не надо.



Особенности проектирования модели данных Cassandra

-  Отсутствуют соединения
-  Отсутствует ссылочная целостность
-  Допустима денормализация данных
-  Используется методика проектирования от запроса

Но, вернемся к особенностям проектирования моделей данных в Cassandra. Еще одно важное отличие состоит в методике проектирования от запроса. Говоря простыми словами, реляционное моделирование означает, что мы начинаем с концептуальной модели предметной области, а затем представляем сущности, оказавшиеся в этой модели, таблицами. После этого назначаются первичные и внешние ключи для моделирования связей. Связи многие-ко-многим представляются связующими таблицами. В реальном мире связующим таблицам ничего не соответствует, это просто необходимый побочный эффект построения реляционных моделей. Определившись с таблицами, мы начинаем писать запросы, в которых данные из разных таблиц связываются с помощью ключей. В реляционном мире запросы - вещь вторичная. Предполагается, что если таблицы спроектированы правильно, то уж данные мы как-нибудь достанем. И обычно это - правда, даже если для решения задачи приходится прибегать к сложным подзапросам или соединениям.



Напротив, в Cassandra мы начинаем не с модели данных, а с модели запросов. Вместо того, чтобы сначала построить модель данных, а затем писать запросы, при работе с Cassandra мы сначала продумываем запросы, а уже вокруг них организуем данные. Решите, какие запросы чаще всего будут предъявляться в приложении, а затем создавайте таблицы, поддерживающие их эффективное выполнение. Критики высказывают мнение, что проектирование от запросов чрезмерно ограничивает область применимости приложения, не говоря уже о моделировании базы данных. Но разве не разумно предположить, что мы обязаны продумать запросы приложения не менее тщательно, чем обдумываем особенности предметной области. Допустив ошибку, мы получим проблемы и в том, и в другом случае. Со временем запросы могут измениться, и тогда придется изменить структуру данных. Но это ничем не отличается от неправильного определения таблиц или необходимости добавления новых таблиц в РСУБД.

Особенности проектирования модели данных Cassandra

-  Отсутствуют соединения
-  Отсутствует ссылочная целостность
-  Допустима денормализация данных
-  Используется методика проектирования от запроса
-  Оптимизация хранения предусматривается на этапе проектирования

В реляционных базах структура хранения таблиц на диске обычно прозрачна для разработчика, и в контексте моделирования данных редко можно встретить рекомендации, касающиеся физического хранения таблиц. Но в Cassandra это важный аспект. Поскольку каждая таблица хранится в отдельном файле, важно, чтобы взаимосвязанные столбцы были определены вместе в одной и той же таблице. При проектировании модели данных в Cassandra важно минимизировать количество разделов, которые нужно просматривать при обработке запроса. Поскольку каждый раздел целиком расположен только на одном узле, быстрее всего выполняются запросы, которым нужно просматривать только один раздел.

Особенности проектирования модели данных Cassandra

-  Отсутствуют соединения
-  Отсутствует ссылочная целостность
-  Допустима денормализация данных
-  Используется методика проектирования от запроса
-  Оптимизация хранения предусматривается на этапе проектирования
-  Решение о сортировке принимается при проектировании

Еще одна важная особенность в проектировании – учет сортировки данных в запросах. В РСУБД порядок, в котором возвращаются записи, легко изменить, включив в запрос фразу ORDER BY. Порядок сортировки по умолчанию не задается, и в отсутствии других указаний записи возвращаются в том порядке, в котором добавлялись. Чтобы изменить порядок сортировки, нужно всего лишь модифицировать запрос, причем сортировать можно по любому набору столбцов. Но в Cassandra к сортировке отношение другое: решение о ней принимается на этапе проектирования. Порядок сортировки в запросе фиксирован и определяется, так называемыми, кластерными столбцами, заданными в команде CREATE TABLE. Команда SELECT (которая также есть в Cassandra) поддерживает семантику ORDER BY, но только в порядке, определяемом кластерными столбцами.



- ✓ Сложная аналитика над большими объемами данных.

л	н	о	г	а
о	а	л	з	н
ж	л	ь	е	н
н	и	и	м	ы
а	т	и	а	х
я	и	м	и	
	к	и		
	а			

Где используются
хранилища типа
ключ-значение?



- ✓ Сложная аналитика над большими объемами данных.
- ✓ Аналитика с непредсказуемыми запросами.

н	е	а
а	п	п
л	р	р
и	е	о
т	д	с
и	с	а
к	к	м
а	а	и
	з	
	у	е
	м	ы
	и	и

Где используются
хранилища типа
ключ-значение?



- ✓ Сложная аналитика над большими объемами данных.
- ✓ Аналитика с непредсказуемыми запросами.
- ✓ Системы с **редким обновлением**.

и	е	б
с	д	н
м	к	о
е	и	в
м	м	л
ы		е
	н	
	и	
	е	
	м	

Где используются
хранилища типа
ключ-значение?



Где используются колоночные хранилища? Колоночные хранилища особенно хорошо подходят для сложной аналитики над большими объемами данных с непредсказуемыми запросами. Например, это может быть статистика о продажах в интернет магазинах по различным категориям, товарам, временным интервалам, покупателям и т.п. Сам характер таких данных подразумевает, что обновляются они крайне редко, а запросы, которые строятся по отношению к этим данным, крайне разнообразны. Колоночные базы данных в силу своей физической организации очень хорошо масштабируются и позволяют проводить практически неограниченное распараллеливание обработки запросов.