[1 ACID - properties of database transactions. 1](#_Toc503334762)

[2 Transaction Isolation Levels 2](#_Toc503334763)

[3 Pessimistic vs optimistic locking(оптимистичная и пессимистичная блокировка) 4](#_Toc503334764)

[4 RDBMS (Relational Database Management System) 5](#_Toc503334765)

[6 Types of NoSQL databases 7](#_Toc503334766)

[8 Cassandra data model: primary key, partition and clustering keys 8](#_Toc503334767)

[9 Cassandra: Bloom filter. Queries 10](#_Toc503334768)

[10 Distribution Models. Consistency problem 12](#_Toc503334769)

[12 Distribution Models. Replication 13](#_Toc503334770)

[14. Consistency guarantee 15](#_Toc503334771)

[18 CAP theorem with SQL and NoSQL DBs 17](#_Toc503334772)

[19 Polyglot Persistence 18](#_Toc503334773)

[22 MapReduce. Map/Reduce/Combine functions requirements 20](#_Toc503334774)

[25 Hadoop architecture and ecosystem 21](#_Toc503334775)

[31 Apache Spark RDD 23](#_Toc503334776)

[32 Apache Spark RDD transformations and actions: filter, map, flatMap, distinct, reduce, groupBy. collect 25](#_Toc503334777)

# 1 ACID - properties of database transactions.

ACID описывает требования к транзакционной системе (например СУБД), обеспечивающие ее надежную работу. Впервые были сформулированы в конце 70-х годов Джимом Греем.

АCID набор свойств, гарантирующие надежность выполнения транзакций в БД.

A - Atomicy (Атомарность)

Либо commit(выполнено), либо rollback(откат). Фиксации промежуточного состояния быть не должно.

Атомарность гарантирует, что никакая транзакция не будет выполнена частично. Будут либо выполнены все её подоперации, либо не выполнено ни одной. Поскольку на практике невозможно одновременно и атомарно выполнить всю последовательность операций внутри транзакции, вводится понятие «отката» (rollback).

С - Consistency — Согласованность

Согласованность не нарушается при выполнении транзакций. Допустимые значения ключей, fk, constraints, cascades, triggers.

Проведенная транзакция (достигшая end of transaction), фиксирующая свои результаты, сохраняет согласованность базы данных. Каждая успешная транзакция фиксирует только допустимые результаты.

Например, если деньги с одного счета списаны, но на другой не зачислены, то свойство согласованности будет нарушено.

Согласованность во время выполнения транзакции не требуется.

I - Isolation - Изолированность

Транзакции, выполняющиеся параллельно, не влияют друг на друга. Результат таких транзакций должен быть таким же, если бы транзакции были выполнены последовательно.

Во время выполнения транзакции параллельные транзакции не должны оказывать влияние на её результат. Изолированность — требование дорогое, поэтому в реальных БД существуют режимы, не полностью изолирующие транзакцию.

Проблемы:

* потерянное обновление: когда одна транзакция перезаписывает другую и результаты одной из них теряются.
* грязное чтение: чтение данных, которые добавлены или изменены транзакцией, а в последующем rollback.
* неповторяющееся чтение: при повторном чтении в рамках одной транзакции данные изменяются.
* фантомное чтение: одна транзакция делает выборку строк несколько раз, другая транзакция изменяет строки из выборки, либо добавляет/удаляет строки, попадающие в выборку. В итоге первая транзакция дает разные выборки по одному условию. От неповторяющегося чтения оно отличается тем, что результат повторного обращения к данным изменился не из-за изменения/удаления самих этих данных, а из-за появления новых (фантомных) данных.
* D - Durability - Долговечность
* Если транзакция завершена успешно, никакое внешнее событие не должно повлиять на сохранность проведенных изменений.
* Если пользователь получил подтверждение от системы, что транзакция выполнена, он может быть уверен, что сделанные им изменения не будут отменены из-за какого-либо сбоя.

На этих принципах построены все реляционные базы данных. Если говорить о не реляционных, то высокой производительности и отказоустойчивости они могут жертвовать изолированностью и полной согласованностью (Теорема CAP), меняя ее на мягкую. В не реляционных бдесть два подхода к обеспечению согласованности: пессимистический (предотвращение возникших конфликтов) и оптимистический (допускает возникновение конфликтов, но подразумевает их идентификацию и меры по их устранению). Также атомарность в не реляционных бд действует только в пределах одного агрегата.

Не реляционные бд поддерживают набор свойств BASE:

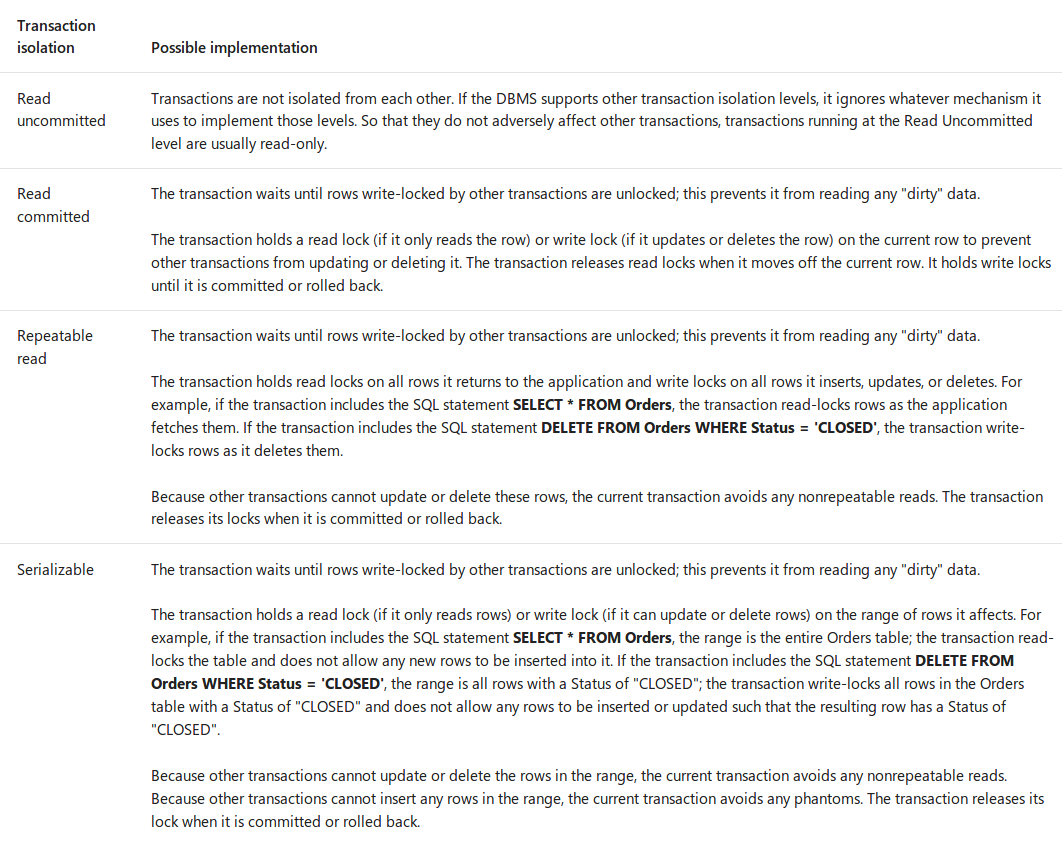
1. базовая доступность — каждый запрос гарантированно завершается (успешно или безуспешно), если на рабочий узел пришел запрос, то и должен быть ответ;
2. гибкое состояние - состояние системы может изменяться со временем, даже без ввода новых данных, для достижения согласования данных.
3. согласованность в конечном счёте — данные могут быть некоторое время не согласованы, но приходят к согласованию через некоторое время.

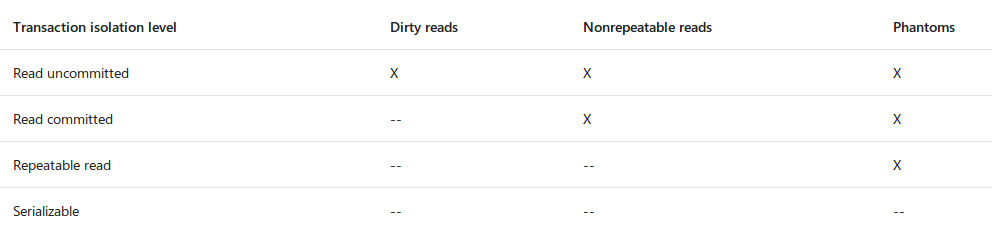
# 2 Transaction Isolation Levels

Transaction isolation levels are a measure of the extent to which transaction isolation succeeds. In particular, transaction isolation levels are defined by the presence or absence of the following phenomena:

* Dirty Reads A dirty read occurs when a transaction reads data that has not yet been committed. For example, suppose transaction 1 updates a row. Transaction 2 reads the updated row before transaction 1 commits the update. If transaction 1 rolls back the change, transaction 2 will have read data that is considered never to have existed.
* Nonrepeatable Reads A nonrepeatable read occurs when a transaction reads the same row twice but gets different data each time. For example, suppose transaction 1 reads a row. Transaction 2 updates or deletes that row and commits the update or delete. If transaction 1 rereads the row, it retrieves different row values or discovers that the row has been deleted.
* Phantoms A phantom is a row that matches the search criteria but is not initially seen. For example, suppose transaction 1 reads a set of rows that satisfy some search criteria. Transaction 2 generates a new row (through either an update or an insert) that matches the search criteria for transaction 1. If transaction 1 reexecutes the statement that reads the rows, it gets a different set of rows.

The four transaction isolation levels (as defined by SQL-92) are defined in terms of these phenomena. In the following table, an "X" marks each phenomenon that can occur.



It is important to note that the transaction isolation level does not affect a transaction's ability to see its own changes; transactions can always see any changes they make. For example, a transaction might consist of two UPDATE statements, the first of which raises the pay of all employees by 10 percent and the second of which sets the pay of any employees over some maximum amount to that amount. This succeeds as a single transaction only because the second UPDATE statement can see the results of the first.

**Examples:**

## *Isolation Levels in MySQL*

InnoDB defaults to Repeatable Read, but offers all four ANSI SQL isolation levels. Read Committed uses Snapshot isolation semantics.

For more information on InnoDB, see [15.3.2.1 Transaction Isolation Levels](https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html).

When using the MyISAM storage engine, transactions are not supported at all. Instead it uses a single reader-writer lock at the table level. (Though in some cases, insert operations can bypass the lock.)

## Isolation Levels in MongoDB

As mentioned before, MongoDB doesn’t support transactions. From the manual,

Because only single-document operations are atomic with MongoDB, two-phase commits can only offer transaction-like semantics. It is possible for applications to return intermediate data at intermediate points during the two-phase commit or rollback.

In real terms this means MongoDB uses dirty read semantics, which includes the possibility for doubled or missing records.

## Isolation Levels in Cassandra

In Cassandra 1.0, not even writes to a single row are isolated. Fields were updated one-by-one, so you could end up reading a record with a mixture of old and new values.

Starting with version 1.1, Cassandra offers “Row Level Isolation”. This brings it up to the same level of isolation that other databases refer to as Read Uncommitted. Higher levels of isolation are not possible.

# 3 Pessimistic vs optimistic locking(оптимистичная и пессимистичная блокировка)

Изоляция транзакций реализуется путём блокировки любого доступа к транзакции. Существует два подхода к блокировке транзакций: пессимистическая блокировка и оптимистическая блокировка.

При **оптимистической блокировке (optimistic locking)** делается

предположение, что конфликта не произойдет. Данные считываются,

транзакция обрабатывается, производятся обновления, а после этого

проверяется, не возник ли конфликт. Если нет, транзакция завершается.

Если да, транзакция повторяется до тех пор, пока не сможет успешно

завершиться. При **пессимистической блокировке (pessimistic locking)**

предполагается, что конфликт обязательно произойдет. Сначала

налагаются блокировки, затем обрабатывается транзакция, а после этого

блокировки снимаются.

**Первая стратегия** — это ввести блокировку вида «данные кто-то держит».

Все сталкивались с ситуацией, когда при открытии Word-документа,

лежащего в сети, показывается сообщение, что мол файл занят и будет

открыт только для чтения. Это называется pessimistic locking

(пессимистическое блокирование). Называется так, потому что

предполагает худшее: данные кому-то понадобились, и он их обязательно

поменяет. Подход надежный, но не лишен недостатков. Может случиться

так, что кто-то начал редактирование, думал-думал минут 20, да и нажал

Cancel. В результате блокировка прошла впустую.

**Другой (менее популярный) сценарий** — это когда во время редактирования человекапохищает летающая тарелка, или у него накрывается комп. Надопредусматривать, как снимать блокировку в таких случаях (тайм-аут иличто-то еще).

Вторая стратегия - **оптимистическая блокировка (optimistic locking)**. Название объясняется тем фактом, что даже если несколько пользователей начали правку данных, то система надеется на лучшее — на то, что они не станут одновременно свои изменения сохранять. Или если правки не пересекаются (и система это допускает), то все также закончится благополучно.

**Пример** — это поведение систем управления ревизиями (version control systems). В них бывает эксклюзивный режим: пользователь «забирает на себя» файл (в SourceSafe это называется checkout), и все его коллеги томятся в ожидании. Бывает и другой режим edit/merge, который мы видим в Subversion и Git. Он как раз использует optimistic locking.

**Недостатком пессимистической блокировки** является то, что ресурс, использующийся одной транзакцией (с момента первого доступа), заблокирован, что делает его недоступным для других транзакций на время выполнение первой транзакции. Если большинство транзакций просто читают данные и никогда не меняют их, то блокировка может быть лишней(может вызвать конфликт блокировок).Тогда оптимистическая блокировка может быть лучшим выходом.

При пессимистической блокировке, блокировки применяются ошибкоустойчивым способом. На примере банковского приложения, учетная запись будет заблокирована, как только она будет доступна в транзакции. Попытки использовать эту учетную запись в других транзакциях во время ее блокировки либо приведут к задержке другого процесса до тех пор, пока блокировка учетной записи не будет снята, или транзакция процесса будет отменена. Блокировка существует до тех пор, пока транзакция не будет совершена (committed) или не будет отменена (rolled back).

**При оптимистической блокировке**, данные фактически не блокируются, когда к ним впервые обращается транзакция. Вместо этого сохраняется состояние данных в момент блокировки пессимистической блокировкой. Другие транзакции могут одновременно получать доступ к этим данным, и возникает возможность возникновения противоречивых изменений. Во время фиксации (commit time), когда данные собираются обновляться в постоянном хранилище, состояние ресурса снова считывается из хранилища и сравнивается с состоянием, которое было сохранено, когда данные были впервые доступны транзакции. Если два состояния отличаются друг от друга, то возник конфликт обновлений, и транзакция будет отменена.

В примере с банковским приложением, сумма в учетной записи сохраняется при первом обращении к учетной записи в транзакции. Если транзакция изменит сумму счета, сумма будет считана из магазина снова, прежде чем сумма будет обновлена. Если сумма была изменена с момента начала транзакции, транзакция сама себя отменит, иначе новая сумма будет записана в постоянное хранилище.

# 4 RDBMS (Relational Database Management System)

Реляционная система управления базами данных (RDBMS) - программная система, обеспечивающая доступ к реляционной базе данных. Представляет собой набор программных приложений, которые могут использоваться для создания, обслуживания, управления и использования базы данных. В реляционной бд данные хранятся и представлены в табличном формате.

(Родионов очень кратко упомянул на лекции. К примеру если есть интернет магазин у которого Табличная БД и нужно добавить товар с каким то новым полем, то это очень много времени займёт, а если в той же монге, то это не вызовет особых трудностей)

Самые популярные: MySQL, MS SQL Server, ORACLE, MS ACCESS.

[**Преимущества RDBMS**](https://ru.bmstu.wiki/RDBMS_(Relational_Database_Management_System)#.D0.A2.D0.B8.D0.BF.D1.8B_RDBMS)

**Недостатки RDBMS**

Стоимость

* дорогостоящая настройка и поддержка системы баз данных (по, долго заполнять\вносить изменения в бд)

Изобилие информации

* РБД предназначены для организации данных по общим характеристикам(все, что не поддается простой классификации ведет к созданию нового типа базы данных, называемого объектно-реляционными системами управления базами данных ORDBMS. Эти системы предназначены для работы с более сложными приложениями и могут быть масштабируемыми)

Ограниченные пределы

* Некоторые рбд имеют ограничения на длину поля (задается при создании бд и при неправильной эксплуатации может привести к потере данных)

Изолированные базы данных

* Приведение различных баз данных к единой структуре может быть сложным и дорогостоящим

**ORM** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Object-Relational Mapping, объектно-реляционное отображение) — технология программирования, которая связывает [базы данных](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B7%D1%8B_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) с концепциями [объектно-ориентированных языков программирования](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), создавая «виртуальную [объектную базу](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D0%B1%D0%B0%D0%B7%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) данных».

<https://ru.wikipedia.org/wiki/ORM>

**NoSQL (SQL vs NoSQL or NotOnlySQL):**

* Not using the relational model
* Don’t use SQL
* Some of them do have query languages
* Running well on clusters
* Relational databases use ACID transactions to handle consistency across the whole database
* NoSQL databases offer a range of options for consistency and distribution
* Graph databases are one style of NoSQL databases that uses a distribution model similar to relational databases but offers a different data model that makes it better at handling data with complex relationships
* Структура и тип хранящихся данных: SQL/реляционные базы данных требуют наличия однозначно определённой структуры хранения данных, а NoSQL базы данных таких ограничений не ставят.
* Запросы: вне зависимости от лицензии, РСУБД реализуют SQL-стандарты, поэтому из них можно получать данные при помощи языка SQL. Каждая NoSQL база данных реализует свой способ работы с данными.
* Масштабируемость: оба решения легко растягиваются вертикально (например, путём увеличения системных ресурсов). Тем не менее, из-за своей современности, решения NoSQL обычно предоставляют более простые способы горизонтального масштабирования (например, создания кластера из нескольких машин).

# 6 Types of NoSQL databases

В зависимости от модели данных и подходов к распределённости и репликации можно выделить четыре типа хранилищ: «ключ-значение» (key-value store), документно-ориентированные (document store), хранилища семейств колонок (column database), графовые базы данных (graph database).

1. Хранилище «ключ-значение»

Хранилище «ключ-значение» является простейшим хранилищем данных, использующим ключ для доступа к значению. Такие хранилища используются для хранения изображений, создания специализированных файловых систем, в качестве кэшей для объектов, а также в системах, спроектированных с прицелом на масштабируемость. Примеры таких хранилищ — Berkeley DB, MemcacheDB, Redis, Riak, Amazon DynamoDB.

1. Хранилище семейств колонок

Таблица в column family databases представляет собой отображение ключа ряда (row key), ключа столбца (column key) и временной метки (timestamp) в значение в виде строки. Ключ ряда и ключ столбца также являются обычными строками. Ключи рядов упорядочены в лексикографическом порядке, а столбцы объединены в семейства столбцов (column family), которые должны быть определены до использования, после чего в каждое семейство столбцы могут быть добавлены динамически. Семейства столбцов обычно хранят однотипные данные, и их число невелико (не более сотни), в то время как столбцов в семействе может быть неограниченное количество. Каждая ячейка таблицы может содержать несколько версий данных, помеченных временными метками и упорядоченными по ним, текущее значение имеет наибольшую временную метку, поддерживается автоматическое удаление старых версий; ячейки также могут вообще не содержать данных. Таким образом, каждая строка таблицы (ряд) может содержать произвольное число атрибутов (столбцов), входящих в заранее определённые семейства.

Типичным применением этого вида СУБД является веб-индексирование, а также задачи, связанные с большими данными, с пониженными требованиями к согласованности данных. Примерами СУБД данного типа являются: Apache HBase, Apache Cassandra, Apache Accumulo, Hypertable, SimpleDB (Amazon.com).

1. Документо-ориентированная СУБД

В основе документоориентированных СУБД лежат документные хранилища (англ. document store), имеющие структуру дерева (иногда леса). Структура дерева начинается с корневого узла и может содержать несколько внутренних и листовых узлов. Листовые узлы содержат данные, которые при добавлении документа заносятся в индексы, что позволяет даже при достаточно сложной структуре находить место (путь) искомых данных. API для поиска позволяет находить по запросу документы и части документов. В отличие от хранилищ типа ключ-значение, выборка по запросу к документному хранилищу может содержать части большого количества документов без полной загрузки этих документов в оперативную память.

Документы могут быть организованы (сгруппированы) в коллекции. Их можно считать отдалённым аналогом таблиц реляционных СУБД, но коллекции могут содержать другие коллекции. Хотя документы коллекции могут быть произвольными, для более эффективного индексирования лучше объединять в коллекцию документы с похожей структурой.

Примеры СУБД данного типа — CouchDB, Couchbase, MarkLogic, MongoDB, eXist, Berkeley DB XML.

1. Базы данных на основе графов

Основными элементами модели являются узлы и связи. Графовые базы данных применяются для задач, в которых данные имеют большое количество связей. Графовые базы данных применяются для моделирования социальных графов (социальных сетей), в биоинформатике, а также для семантической паутины. Примеры: Neo4j, OrientDB, AllegroGraph, Blazegraph (RDF-хранилище, ранее называлось Bigdata), InfiniteGraph, FlockDB, Titan.

Так как рёбра графа материализованы (англ. materialized), то есть, являются хранимыми, обход графа не требует дополнительных вычислений (как JOIN в SQL), но для нахождения начальной вершины обхода требуется наличие индексов. Графовые базы данных как правило поддерживают ACID, а также имеют различные языки запросов, вроде Gremlin и Cypher (Neo4j).

7. Column-family store

* Представляет собой отображение ключа ряда (row key), ключа

столбца (column key) и временной метки (timestamp) в значение в

виде строки.

• Ключи рядов упорядочены в лексикографическом порядке, а

столбцы объединены в семейства столбцов (column family)

• Семейства столбцов обычно хранят однотипные данные, и их

число невелико (не более сотни)

• Столбцов в семействе столбцов может быть неограниченное

количество

• Каждая ячейка таблицы может содержать несколько версий

данных, помеченных временными метками и упорядоченными по

ним

# 8 Cassandra data model: primary key, partition and clustering keys

Система Cassandra была разработана и использовалась в Facebook. В её основе лежат идеи Google BigTable и Amazon Dynamo. В настоящее время Cassandra является проектом с открытым исходным кодом (на Java), поддерживаемым Apache Software Foundation.

База данных в Cassandra называется «пространством ключей» (keyspace) и содержит семейства столбцов (column family), которые являются аналогом таблиц и служат контейнерами для строк (рядов, rows), идентифицируемых уникальными ключами (row key). Строки состоят из столбцов (column) или супер-столбцов (super column). Столбец является минимальной единицей данных в Cassandra и состоит из имени, значения и временной метки (все эти поля предоставляются клиентом), хранится только последняя версия данных. Супер-столбцы, в свою очередь, содержат внутри себя столбцы, добавляя тем самым еще один уровень вложенности. Кроме того, поддерживаются специальные столбцы, такие как счётчики или столбцы с указанным временем жизни (TTL). Разным строкам необязательно должен соответствовать один и тот же набор столбцов или супер-столбцов. Семейства столбцов хранятся в отдельных файлах с сортировкой по ключам строк и должны содержать столбцы, доступ к которым в запросах предполагается осуществлять вместе.

Для работы с данными Cassandra поддерживает SQL-подобный язык CQL (Cassandra Query Language), кроме того, есть поддержка Hadoop MapReduce.

Для ускорения запросов поддерживается создание вторичных индексов. Операции модификации данных являются атомарными на уровне одной строки таблицы, постоянство хранения обеспечивается с помощью записи в журнал, поддерживается сжатие данных.

Cassandra позволяет гибко варьировать уровень согласованности данных на уровне операций. Конфликты разрешаются на основании временных меток (выигрывает последняя версия).

Cassandra проектировалась так, чтобы обеспечить хорошую масштабируемость и надежность на большом количестве недорогих (и ненадежным) машин. В кластере Cassandra нет выделенных узлов, все они равноправны и выполняют одни и те же функции. Для распределения данных по узлам применяется консистентное хэширование (consistent hashing) и направленная передача (hinted handoff), новые узлы могут быть легко добавлены в кластер, а обнаружение сбоев и восстановление происходят автоматически. Разделение строк по узлам может осуществляться как случайным образом, так и с сохранением порядка. Репликация поддерживается как в пределах кластера, так и между географически распределёнными кластерами.

* Primary Key эквивалентен Partition Key в таблице с одним полем.
* Первичный ключ (primary key) является основой эффективного моделирования данных в Apache Cassandra.
* Первичный ключ определяется на этапе инструкции создания колоночного семейства
  + Первичный ключ может состоять из нескольких элементов, первый из которых является ключом раздела (partition key). Composite/Compound Key - составной первичный ключ состоит из одного или нескольких столбцов, на которые ссылаются первичный ключ. Один компонент составного первичного ключа - Partition Key, а другой компонент Clustering Key. Ниже приведены различные варианты первичных ключей.C1, C2, C3, … и тд представляют столбцы в таблице:
    - C1: первичный ключ имеет только один Partition Key и Clustering Key.
    - (C1, C2): столбец C1 является Partition Key, а столбец C2 является Clustering Key.
    - (C1, C2, C3, ...): столбец C1 является Partition Key, а столбцы C2, C3 и тд образуют Clustering Key.
    - (C1, (C2, C3, ...)): столбец C1 является Partition Key, а столбцы C2, C3, ... образуют Clustering Key.
    - ((C1, C2, ...), (C3, C4, ...)): столбцы C1, C2 составляют Partition Key и столбцы C3, C4, ... образуют Clustering Key.
  + Важно отметить, что когда составным ключом является C1, C2, C3, тогда первый C1 становится Partition Key, а остальные ключи становятся частью Clustering Key. Для создания составных ключей разделов нужно указывать ключи в скобках, например: ((C1, C2), C3, C4). В этом случае C1 и C2 являются частью Partition Key, а C3, C4 являются частью Clustering Key.
* Partition Key (ключ раздела) отвечает за распределение данных по вашим узлам.
* Ключ раздела не только определяет уникальность записи в БД, но также имеет специальное назначение в Cassandra: определяет местоположение данных, что крайне важно в распределенных системах.
* Цель ключа раздела - идентифицировать раздел или узел в кластере, который хранит эту строку. Когда данные считываются или записываются из кластера, для вычисления хэш-значения ключа раздела используется функция Partitioner. Это значение хеша используется для определения узла / раздела, который содержит эту строку.
* Ключ раздела всегда принадлежит одному узлу, и данные этого раздела всегда можно найти на этом узле.
* Если бы не было задано абсолютное местоположение данных, тогда их пришлось бы искать на всех узлах кластера. В небольшом кластере такую операцию можно выполнить достаточно быстро, однако в более крупных кластерах это потребует слишком много времени.
* Clustering Key отвечает за сортировку данных в разделе.
  + Цель Clustering Key - хранить данные строки в отсортированном порядке.
  + Сортировка данных основана на столбцах, которые включены в Clustering Key. Такая компоновка позволяет эффективно извлекать данные с помощью Clustering Key.
  + Кластеризация - процесс механизма хранения, который создает индекс и сохраняет данные в порядке, основанном на индексе.

# 9 Cassandra: Bloom filter. Queries

Changing

The bloom filter false positive chance is visible in the DESCRIBE TABLE output as the field bloom\_filter\_fp\_chance. Operators can change the value with an ALTER TABLE statement:

ALTER TABLE keyspace.table WITH bloom\_filter\_fp\_chance=0.01

Bloom-фильтр

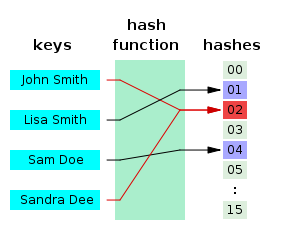
Имея сотни, а может быть и тысячи SSTable как быстро ответить на вопрос в какой из них искать нужную запись? Можно последовательно перебирать файлы индексов раздела, но это очень сильно нагрузит дисковую подсистему, пусть даже это будет проходить через сводки разделов, располагающихся в оперативной памяти. Есть более гибкое решение, которое может однозначно предупреждать о 100% отсутствии нужной записи в той или иной SSTable. Решение заключается в использовании вероятностной структуры данных под названием bloom-фильтр (по фамилии создателя). «Вероятностная» означает, что полученный ответ не может быть на 100% достоверным, в нашем случае мы задаем вопрос «Имеется ли запись А в SSTable 1» и получаем один из возможных ответов — «Точно нет» или «Скорее всего да». То есть во втором случае мы имеем некоторую вероятность нахождения записи в определенной SSTable. В случае с Cassandra значение вероятности можно гибко регулировать и итоговая вероятность отсутствия записи при положительном ответе составит не больше нескольких процентов или ещё меньше.

Для понимания работы компонента более подробного объяснения в принципе не нужно, к тому же в официальной документации все ограничивается достаточно скупыми объяснениями и рекомендаціями. Однако я все же постараюсь более подробно объяснить как работает bloom-фильтр.

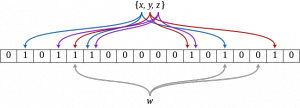
Допустим мы имеем массив битов некоторой длины и 2 разные хэш-функции. Значение хэш-функций для имени Lena (в нашем случае имя — это ключ записи) будет равно 45 для первой хэш-функции и 92 для второй соответственно. В массиве битов, который изначально состоит только из 0, переводим соответствующие по счету биты в значение 1. То же самое выполняем для имени Masha, но только получаем 73 для первой хэш-функции и 23 для второй. Теперь имеем массив битов, в котором позиции 45, 92, 73 и 23 установлены в 1. Далее сбрасываем данные в SSTable, в которой в итоге будем иметь всего две записи — Lena и Masha, назовем её «SSTable 1».

Сценарий работы: получаем на входе ключ записи со значением Egor (для наглядности я все также использую имена как ключи записей). Имея SSTable 1, нам  надо ответить на вопрос «В SSTable 1 есть запись Egor?». Для этого вычисляем для ключа записи Egor хэш-функции и на выходе получаем, например, значения 48 для первой и 76 для второй хэш-функции. Теперь просматриваем наш массив битов «bloom-фильтра» и проверяем в какое значение установлены соответствующие биты. В итоге видим, что биты под номерами 48 и 76 установлены в 0 и таким образом получаем ответ: «В SSTable 1 точно нет записи Egor». В итоге не обращаясь ни к индексам раздела, ни к сводке индекса мы получили нужный ответ. При проверке ключа записи Lena мы получим значения 45 и 92 для первой и второй хэш-функции и проверив соответствующие биты массива увидим, что они установлены в 1, таким образом получаем ответ, что скорее всего запись с ключом Lena присутствует в SSTable 1.

Почему «скорее всего»? Дело в том, что присутствует некоторая вероятность того, что у полностью отличающегося от присутствующих значения ключа записи, мы можем получить значения всех хэш-функций, которые будут равны значениям существующих ключей записей. В итоге будет ложное срабатывание. Наглядный пример из Википедии [15](http://blog.bissquit.com/dbms/apache-cassandra/apache-cassandra-zapis-dannyh-chast-3-sstable/#note-910-15):

[](http://blog.bissquit.com/wp-content/uploads/2015/03/300px-Hash_table_4_1_1_0_0_1_0_LL.svg_.png)

Иллюстрация, подробно показывающая пример работы bloom-фильтра, все также из Википедии [16](http://blog.bissquit.com/dbms/apache-cassandra/apache-cassandra-zapis-dannyh-chast-3-sstable/#note-910-16):

[](http://blog.bissquit.com/wp-content/uploads/2015/03/649px-Bloom_filter.svg_.png)

An example of a Bloom filter, representing the set { x, y, z }. The colored arrows show the positions in the bit array that each set element is mapped to. The element w is not in the set { x, y, z }, because it hashes to one bit-array position containing 0. For this figure, m = 18 and k = 3.

С bloom-фильтром на этом все, переходим к компоненту SSTable, ради которого собственно и нужны все механизмы, рассмотренные выше.

Фильтр Блума ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Bloom filter) — это вероятностная [структура данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), придуманная Бёртоном Блумом в 1970 году[[1]](https://ru.wikipedia.org/wiki/%D0%A4%D0%B8%D0%BB%D1%8C%D1%82%D1%80_%D0%91%D0%BB%D1%83%D0%BC%D0%B0#cite_note-1), позволяющая проверять принадлежность элемента к [множеству](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE_%28%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85%29). При этом существует возможность получить [ложноположительное срабатывание](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B6%D0%BD%D0%BE%D0%BF%D0%BE%D0%BB%D0%BE%D0%B6%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D1%81%D1%80%D0%B0%D0%B1%D0%B0%D1%82%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) (элемента в множестве нет, но структура данных сообщает, что он есть), но не [ложноотрицательное](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B6%D0%BD%D0%BE%D0%BE%D1%82%D1%80%D0%B8%D1%86%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D1%81%D1%80%D0%B0%D0%B1%D0%B0%D1%82%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5).

Фильтр Блума может использовать любой объём [памяти](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C), заранее заданный пользователем, причём чем он больше, тем меньше [вероятность](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D1%80%D0%BE%D1%8F%D1%82%D0%BD%D0%BE%D1%81%D1%82%D1%8C) ложного срабатывания. Поддерживается операция добавления новых элементов в множество, но не удаления существующих (если только не используется модификация со счётчиками).

Свойства

* В отличие от многих других структур данных (например, [хеш-таблиц](https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0)), также хранящих множество элементов, фильтр Блума может представлять [универсальное множество](https://ru.wikipedia.org/wiki/%D0%A3%D0%BD%D0%B8%D0%B2%D0%B5%D1%80%D1%81%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE) всех возможных элементов. В этом случае все биты в его битовом массиве равны единице.
* [Объединение](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%B4%D0%B8%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2) и [пересечение](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D1%81%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2) двух фильтров Блума одинакового размера и c одинаковым множеством хеш-функций может быть реализовано [побитовыми операциями](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82%D0%BE%D0%B2%D1%8B%D0%B5_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8) [OR](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%B7%D1%8A%D1%8E%D0%BD%D0%BA%D1%86%D0%B8%D1%8F) и [AND](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D1%8A%D1%8E%D0%BD%D0%BA%D1%86%D0%B8%D1%8F) над их битовыми массивами.

Применение

По сравнению с [хеш-таблицами](https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0), фильтр Блума может обходиться на несколько порядков меньшими объёмами памяти, жертвуя детерминизмом. Обычно он используется для уменьшения числа запросов к несуществующим данным в структуре данных с более дорогостоящим доступом (например, расположенной на жестком диске или в сетевой базе данных), то есть для «фильтрации» запросов к ней.

Примеры практических применений:

* Прокси-сервер [Squid](https://ru.wikipedia.org/wiki/Squid) использует фильтры Блума для опции [cache digests](http://wiki.squid-cache.org/SquidFaq/CacheDigests).
* [Google](https://ru.wikipedia.org/wiki/Google_%28%D0%BA%D0%BE%D0%BC%D0%BF%D0%B0%D0%BD%D0%B8%D1%8F%29) [BigTable](https://ru.wikipedia.org/wiki/BigTable) использует фильтры Блума для уменьшения числа обращений к жесткому диску при проверке на существование заданной строки или столбца в таблице базы данных.[[2]](https://ru.wikipedia.org/wiki/%D0%A4%D0%B8%D0%BB%D1%8C%D1%82%D1%80_%D0%91%D0%BB%D1%83%D0%BC%D0%B0#cite_note-2)
* Компьютерные программы для проверки орфографии.

Запросы

Querying data

Retrieve data sets from a table using SELECT statements.

About Select statement

Use a SELECT statement to retrieve a results set from a table as standard output or in JSON format.

Functions allow you to manipulate the column values or aggregate the entire results set of specific columns.

SELECT column\_list FROM [keyspace\_name.]table\_name

[WHERE prinmary\_key\_conditions[ AND clustering\_columns\_conditions]] | PRIMARY KEY LIMIT

UPDATE [keyspace\_name.] table\_name

[USING TTL time\_value | USING TIMESTAMP timestamp\_value]

SET assignment [, assignment, ...]

WHERE row\_specification

[IF EXISTS | IF condition [AND condition]]

DELETE [column\_name (term)][, ...]

FROM [keyspace\_name.] table\_name

[USING TIMESTAMP timestamp\_value]

WHERE PK\_column\_conditions

[IF EXISTS | IF static\_column\_conditions]

# 10 Distribution Models. Consistency problem

Проблема неузгодженості даних

1. При реплікації даних в кластері типу master-slave можливе читання недостовірних даних. Наприклад, один користувач записав дані (на master node). Далі вони реплікуються в асинхронному режимі. В той же час інший користувач читає дані з якоїсь вторинної ноди (slave) та отримує попередні дані так як нові ще не реплікувались на неї
2. Читання своїх записів (read your writes) - підсилює узгодженість «в кінцевому рахунку», даючи гарантію того, що клієнт завжди побачить дані, які він до цього записав. В цьому також є своя проблема. В найгіршому випадку клієнт не може читати дані запису, який тільки що відбувся.

Inconsistency window -  якщо період між оновленням даних та моментом поки база гарантує, що будь-який користувач бачитиме оновлені дані більше ніж час за який користувач перезавантажить сторінку

# 12 Distribution Models. Replication

Одни и те же данные хранятся на нескольких узнал в сети.

Типы репликации: синхронная и асинхронная.

Репликация помогает повысить надежность системы и справляться со сбоями как отдельных узлов, так и с потерей целого кластера.

Позволяет получить прирост операции чтения:

* при синхронной репликации увеличивается время ответа системы. По сути в таком случае нам нужно дождаться записи на требуемые ноды
* при асинхронном этой проблемы нет, но появляется другая. Возникает промежуток времени, когда реплики находятся в несогласованном состоянии

Репликация позволяет использовать два или больше одинаковых серверов вместо одного. Операций чтения (SELECT) данных часто намного больше, чем операций изменения данных (INSERT/UPDATE). Поэтому, репликация позволяет разгрузить основной сервер за счет переноса операций чтения на слейв.

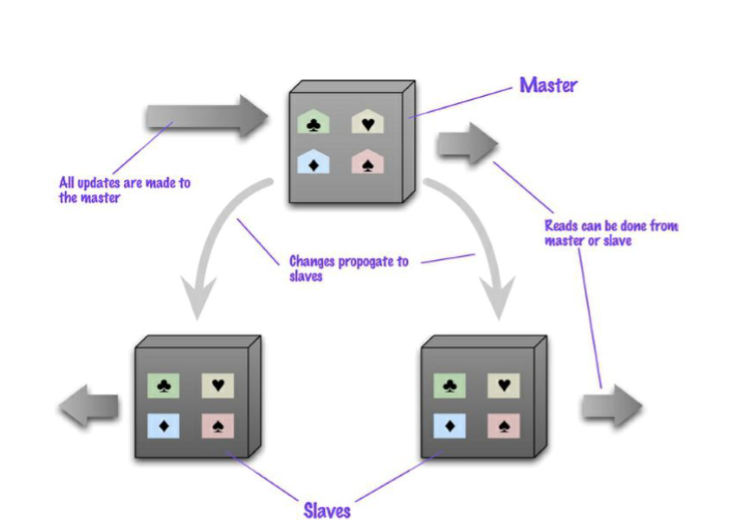
**Master-slave репликация**

Из названия понятно, что есть master нода, которая отвечает за запись данных. Таким образом запись проходит через нее.

Чтение же может осуществляться с любой ноды, в том числе и с master, на которой гарантированно находится самая актуальная версия реплики. Данные на slave нодах могут «отставать» от ноды master и быть устаревшими при асинхронной репликации.

Хорошо подходит для случаев, когда преобладает операция чтения, ведь операции чтения могут происходить с slave нод, уменьшая нагрузку на master.

Master нода может задаваться как вручную, так и выбираться автоматически в случае краха master.



**Несогласованность - Inconsistency**

**Replication inconsistency**

Разные клиенты, читающие с разных slave нод, могут получить разные данные.

**Read-your-writes consistency**

В худшем случае прочитать то, что только что записал.

**Inconsistency window**

Понятие, определяющее промежуток времени, когда система находится в несогласованном состоянии.

**Replication in mongoDB**

Реплика сет имеет только одну primary ноду, она отвечает за все операции записи от клиентов. Реплика сет предоставляет строгую согласованность для всех операций чтения с праймери ноды.

В случае, если праймери нода не связывается с другими нодами реплика сета, то происходит голосование за выбор новой ноды.

**Write Concern Levels**

* **Unacknowledged**

MongoDB не получает подтверждения об успешности операции записи.

* **Acknowledged**

MongoDB подтверждает операцию записи и вносит изменения в in-memory-view of data.

* **Journaled**

MongoDB дает ответ о записи только после занесения данных о записи в журнал.

* **Replica acknowledged**

MongoDB гарантирует, что запись была произведена на дополнительные ноды (кроме как только на праймери). Тут мы можем задавать количество нод, на которые будем гарантировать запись и timeout понятно для чего.

**Read Preference**

primary - по дефолту. Всё чтение происходит с праймери ноды.

primaryPrefered - в большинстве ситуаций чтение происходит с праймери, но в ситуациях, когда она недоступна, чтение происходит с secondary ноды.

secondary - все операции чтения происходят с secondary нод.

secondaryPrefered - в большинстве ситуаций с secondary. Но если secondary недоступна, то чтение происходит с master node.

nearest - операции чтения будут выполняться с той ноды, с которой скорость ответа по сети будет наименьшей независимо от типа этой ноды.

**Пару слов об Apache Cassandra**

В кластере несколько узлов. При операции записи происходит вычисление md5 ключа и записывается на подходящую ноду, а также на следующие за ней. (количество задается - consistency level).

В Cassandra нет master node, так как она проектировалась для высокой доступности. Каждая нода - peer (равная) в кластере.

Итоги:

Master-slave дает прирост в чтении, но в операции записи профита не получаем.

Все узлы могут обрабатывать операции чтения и записи и передавать обновления другим.

Реализовать синхронную репликацию достаточно сложно. (Подразумевается адекватное время ответа системы)

При асинхронном обновлении могут появиться конфликтующие версии данных, которые требуют наличия механизма для определения и разрешения конфликтов.

Peer-to-peer репликация подразумевает отсутствие master ноды.

# 14. Consistency guarantee

(Загальні формули для гарантії узгодженості)

У деяких системах рівень підтримки узгодженості можна варіювати, використовуючи різні конфігурації, налаштування і / або операції. Розглянемо, як цього можна досягти на практиці за допомогою кворуму.

N - число вузлів на які реплікуються дані

W - число вузлів, від яких очікується підтвердження успішного запису

R - число вузлів, до яких звертається клієнт при читанні даних

Змінюючи ці параметри, можна домогтися необхідної поведінки у розподіленій системі:

* R + W > N - множини реплік для запису і читання перетинаються, а значить, читання завжди поверне успішний результат операції запису, що завершилася - strong consistency.
* R + W ≤ N - неможливо гарантувати повернення останньої версії даних, гарантується лише eventual consistency.

При R = N, W = 1 і W = N, R = 1

* підтримується миттєва узгодженість (strong consistency)
* R = N, W = 1 - операції запису (write) працюють швидко за рахунок більш повільних і дорогих читань (read) з N реплік.
* W = N, R = 1 - операції читання (read) працюють швидко за рахунок більш повільних і дорогих записів (write) на N реплік

Чим ближче значення R і W до загальної кількості реплік N, тим більша ймовірність, що операція може завершитися помилкою через вихід будь-яких вузлів з ладу.

(Детальне пояснення щодо забезпечення сonsistency guarantee для різних видів реплікації)

Чем больше узлов задействовано в вашем запросе, тем выше вероятность, что согласованность не возникнет. Естественным образом возникает вопрос: сколько узлов должно быть вовлечено в запрос, чтобы обеспечить строгую согласованность данных?

Представьте себе ситуацию, в которой данные реплицированы на трех узлах. Совершенно не нужно, чтобы все узлы подтверждали запись для обеспечения строгой согласованности; нужно, чтобы это сделали два из них, т.е. большинство. Если возникнут конфликтующие записи, то большинство получит только одна из них. Это явление называется кворумом записи (write quorum) и выражается несколько претенциозным неравенством W > N / 2, означающим, что количество узлов, участвующих в записи (W), должно превышать половину количества узлов, задействованных в чтении (N). Количество реплик часто называется коэффициентом репликации.

Аналогично кворуму записи, существует кворум чтения: со сколькими узлами следует установить контакт, чтобы гарантировать, что вы получаете самое последнее изменение. Кворум чтения представляет собой немного более сложное понятие, потому что он зависит от того, сколько узлов должны подтвердить запись. Рассмотрим ситуацию, в которой коэффициент репликации равен 3. Если все записи должны подтвердить два узла (W == 2), то мы должны установить контакт по крайней мере с двумя узлами, чтобы гарантировать получение последних данных. Если же записи подтверждаются только одним узлом (W == 1), то мы должны связаться со всеми тремя узлами, чтобы гарантировать получение последних обновлений. В последнем случае у нас нет кворума записи, поэтому возникает конфликт обновлений, но, контактируя с достаточно большим количеством читателей, мы можем гарантированно обнаружить конфликт. Таким образом, мы можем получить строго согласованные результаты чтения, даже если у нас нет строгой согласованности записей.

Это отношение между количеством узлов, с которыми необходимо связаться при чтении (R), узлов, подтверждающих запись (W), и коэффициентом репликации (N) можно выразить в виде неравенства: строгую согласованность можно получить, если R + W > N.

Эти неравенства выведены для одноранговой модели распределения (Peer-to-peer). В случае распределения "master-slave", чтобы избежать конфликтов "запись-запись'', достаточно записать данные на master узла. Аналогично, для того чтобы избежать конфликтов "чтение-запись", достаточно выполнять чтение только с slave узла. В использованной системе обозначений легко перепутать количество узлов в кластере с коэффициентом репликации, но часто это совершенно разные числа. При шардинге баз данных вы можете иметь 100 узлов в кластере при коэффициенте репликации, равном 3.

В большинстве случаев руководство считает, что для обеспечения хорошей отказоустойчивости достаточно коэффициента репликации, равного 3. Это позволяет сохранить кворумы чтения и записи при сбое одного узла. При автоматической балансировке процесс создания третьей реплики может потребовать недопустимо много времени, поэтому существует небольшая вероятность потери второй реплики до замены узла.

Количество узлов, задействованных в операции, может зависеть от вида операции. При записи одни типы обновлений могут требовать наличия кворума, а другие нет. Это зависит от того, насколько высокую степень согласованности и доступности мы требуем от системы. Аналогично, если необходимо обеспечить быстрое чтение и допускается определенное устаревание данных, количество узлов может быть небольшим.

Часто требуется учесть оба фактора. Если требуется быстрое, строго согласованное чтение, то можно потребовать подтверждение записей от всех узлов, разрешая чтение только с одного узла (N = 3, W = 3, R = 1). Это может замедлить процесс записи, поскольку при этом необходимо установить контакт со всеми тремя узлами, а вы не можете допустить потери ни одного узла. Однако в некоторых ситуациях возможен компромисс.

# 18 CAP theorem with SQL and NoSQL DBs

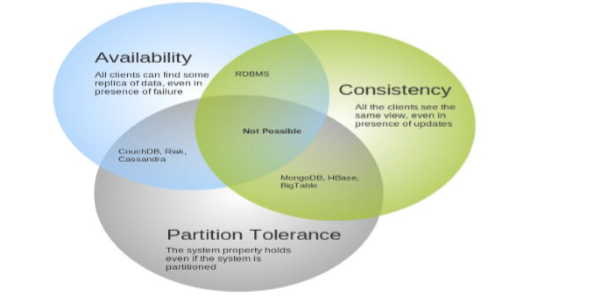
The CAP Theorem

Published by Eric Brewer in 2000, the theorem is a set of basic requirements that describe any distributed system. If you imagine a distributed database system with multiple servers, here's how the CAP theorem applies:

* Consistency - All the servers in the system will have the same data so users will get the same copy regardless of which server answers their request.
* Availability - The system will always respond to a request (even if it's not the latest data or consistent across the system or just a message saying the system isn't working).
* Partition Tolerance - The system continues to operate as a whole even if individual servers fail or can't be reached.

It's theoretically impossible to have all 3 requirements met, so a combination of 2 must be chosen and this is usually the deciding factor in what technology is used.

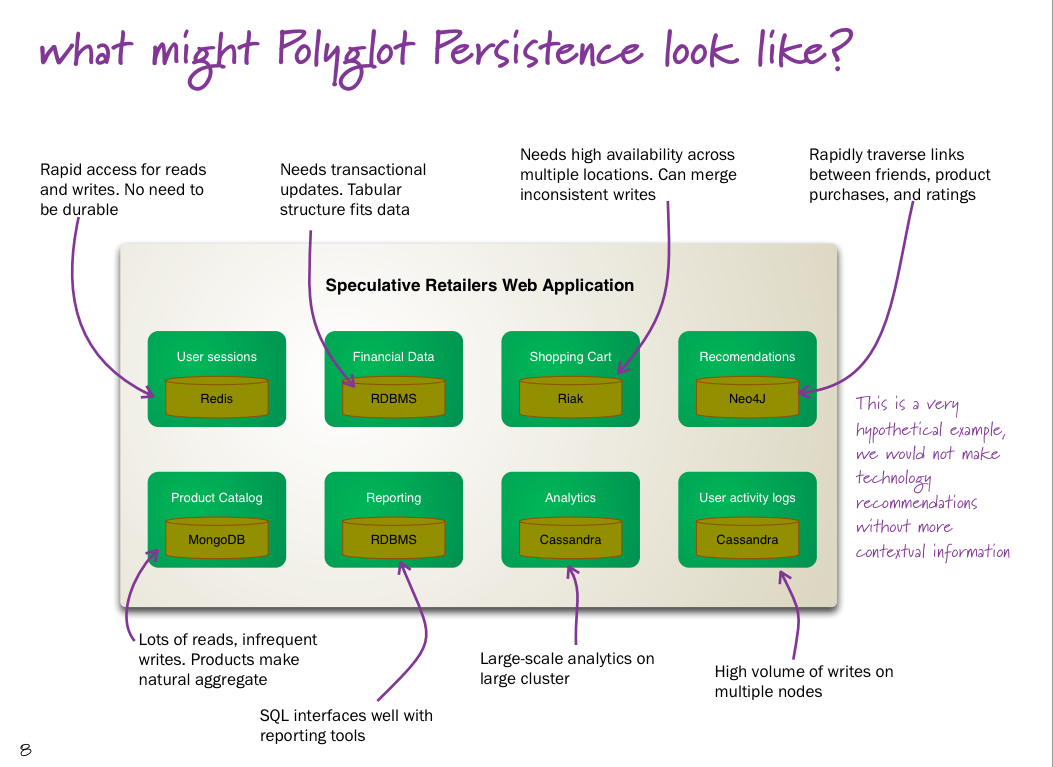
When it comes to distributed databases, the two choices are only AP or CP because if it's not partition tolerant, it's not really a reliable distributed database. So the choice is simpler: if a network split happens, do you want the database to keep answering but with possibly old/bad data (AP)? Or should it just stop responding unless you can get the absolute latest copy (CP)?



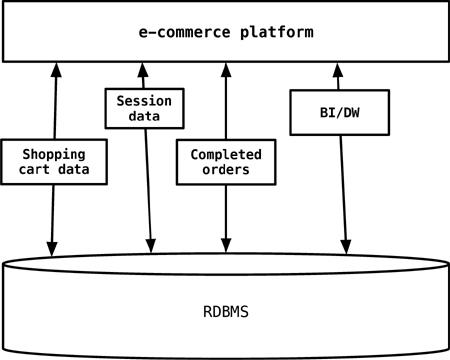
# 19 Polyglot Persistence

Polyglot Persistence - состоит в том, что приложение будет взаимодействовать с разными базами данных и использовать каждую в зависимости от поставленной задачи, для решения которой эта база данных будет подходить лучше всего. Например, RDBMS хороши при наличие множества связей (relationships) между сущностями: для поиска данных из разных таблиц, принадлежащих одному и тому же объекту, мы можем использовать SQL join. Этот вариант хорошо подходит для данных небольшого размера, но когда данных становится больше, могут возникать проблемы. Базы данных на основе графов может решить проблему отношений в случае BigData, но она может не решить проблему транзакций, предоставляемую RDBMS. Документоориентированная NoSQL может хранить неструктурированные данные для решения этой проблемы. Таким образом, у нас есть разные проблемы, которые решаются разными базами данных из одного приложения. Проще говоря, Polyglot Persistence это использование разных технологий хранения данных в зависимости от поставленных задач и типов хранимых данных.

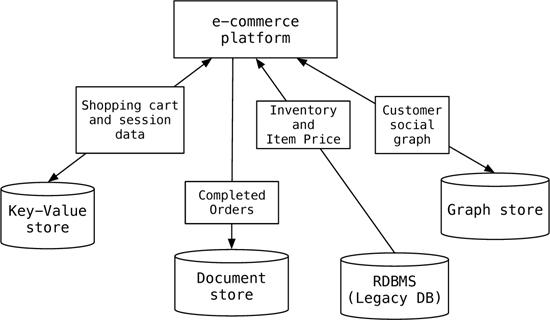
Преимущества различных баз данных для решения разных задач:



Пример стандартного подхода(Monolithic relational DB):



Пример Polyglot Persistence(Specialized NoSQL):



# 22 MapReduce. Map/Reduce/Combine functions requirements

In MongoDB

**Map**

* In the map function, reference the current document as this within the function.
* The map function should not access the database for any reason.
* The map function should be pure, or have no impact outside of the function (i.e. side effects.)
* A single emit can only hold half of MongoDB’s maximum BSON document size.
* The map function may optionally call emit(key,value) any number of times to create an output document associating key with value.

**Reduce**

* The reduce function should *not* access the database, even to perform read operations.
* The reduce function should *not* affect the outside system.
* MongoDB will not call the reduce function for a key that has only a single value. The values argument is an array whose elements are the value objects that are “mapped” to the key.
* MongoDB can invoke the reduce function more than once for the same key. In this case, the previous output from the reduce function for that key will become one of the input values to the next reducefunction invocation for that key.
* The reduce function can access the variables defined in the scope parameter.
* The inputs to reduce must not be larger than half of MongoDB’s maximum BSON document size. This requirement may be violated when large documents are returned and then joined together in subsequent reduce steps.
* Because it is possible to invoke the combine function zero or more times for the same key the type of the return object must be identical to the type of the value emitted by the map function
* the reduce function must be **associative**

reduce(key, [ C, reduce (key, [ A, B ]) ] ) == reduce ( key, [ C, A, B ] )

* the reduce function must be **idempotent**

reduce ( key, [reduce (key, valuesArray) ] ) == reduce ( key, valuesArray )

* the reduce function should be **commutative**. that is, the order of the elements in the valuesArray should not affect the output of the reduce function

reduce ( key, [ A, B ] ) == reduce ( key, [ B, A ] )

In Hadoop

**Map**

* When an individual map task starts it will open a new output writer per configured reduce task. It will then proceed to read its FileSplit using the RecordReader it gets from the specified InputFormat. InputFormat parses the input and generates key-value pairs. InputFormat must also handle records that may be split on the FileSplit boundary.
* As the Map operation is parallelized the input file set is first split to several pieces called [FileSplits](http://hadoop.apache.org/core/docs/current/api/org/apache/hadoop/mapreduce/lib/input/FileSplit.html). If an individual file is so large that it will affect seek time it will be split to several Splits
* As key-value pairs are read from the RecordReader they are passed to the configured [Mapper](http://hadoop.apache.org/core/docs/current/api/org/apache/hadoop/mapreduce/Mapper.html). The user supplied Mapper does whatever it wants with the input pair and calls [OutputCollector.collect](http://hadoop.apache.org/core/docs/current/api/org/apache/hadoop/mapred/OutputCollector.html#collect(org.apache.hadoop.io.WritableComparable,%20org.apache.hadoop.io.Writable)) with key-value pairs of its own choosing. The output it generates must use one key class and one value class.
* The Map input and output key-value pairs are not necessarily related typewise or in cardinality.
* When Mapper output is collected it is partitioned, which means that it will be written to the output specified by the [Partitioner](http://hadoop.apache.org/core/docs/current/api/org/apache/hadoop/mapreduce/Partitioner.html).

**Reduce**

* When a reduce task starts, its input is scattered in many files across all the nodes where map tasks ran.
* When the copy phase completes, executes the sort phase to merge the copied results into a single sorted set of (key, value-list) pairs.
* When the sort phase completes, executes the reduce phase, invoking the job-supplied reduce function on each (key, value-list) pair.
* Saves the final results to the output destination, such as HDFS.
* The output will consist of one output file per executed reduce task

**Combine**

* When the map operation outputs its pairs they are already available in memory.
* For efficiency reasons, sometimes it makes sense to take advantage of this fact by supplying a combiner class to perform a reduce-type function.
* If a combiner is used then the map key-value pairs are not immediately written to the output. Instead they will be collected in lists, one list per each key value.
* When a certain number of key-value pairs have been written, this buffer is flushed by passing all the values of each key to the combiner's reduce method and outputting the key-value pairs of the combine operation as if they were created by the original map operation.

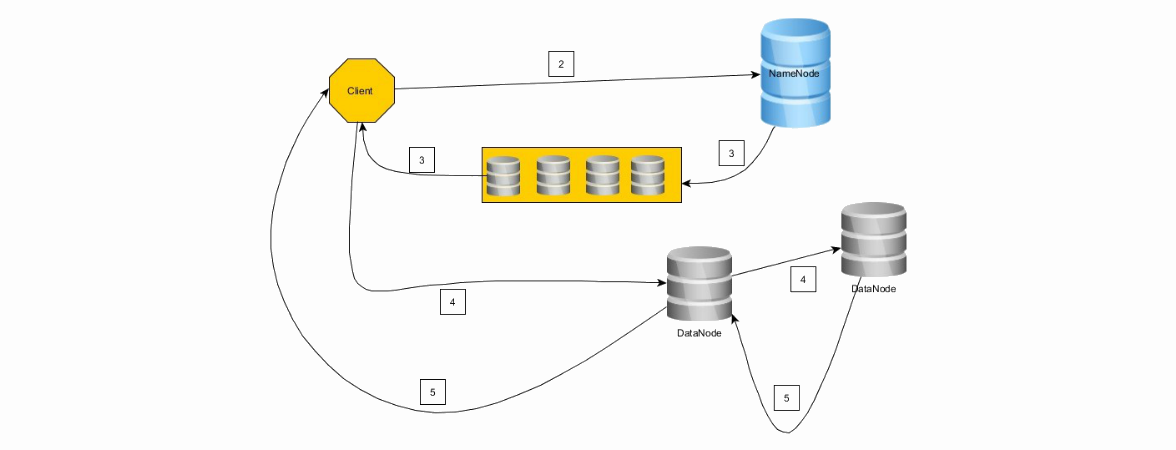
# 25 Hadoop architecture and ecosystem

Экосистема Hadoop есть HDFS.

Отличия HDFS от других ФС:

* Предназначена для хранения большого количества огромных (>10GB) файлов. Одним Следствие — большой размер блока по сравнению с другими файловыми системами (>64MB)
* Оптимизирована для поддержки потокового доступа к данным (high-streaming read), соответственно производительность операций произвольного чтения данных начинает хромать.
* Ориентирована на использование большого количество недорогих серверов. В частности, серверы используют JBOB структуру (Just a bunch of disk) вместо RAID — зеркалирование и репликация осуществляются на уровне кластера, а не на уровне отдельной машины.
* Многие традиционные проблемы распределенных систем заложены в дизайн — уже по дефолту все выход отдельных нод из строя является совершенно нормальной и естественной операцией, а не чем-то из ряда вон.

Hadoop-кластер состоит из нод трех типов: NameNode, Secondary NameNode, Datanode.

**Namenode** — мозг системы. Как правило, одна нода на кластер. Хранит в себе все метаданные системы — непосредственно маппинг между файлами и блоками. Если нода 1 то она же и является Single Point of Failure. Эта проблема решена во второй версии Hadoop с помощью [Namenode Federation](http://www.slideshare.net/EdurekaIN/hadoop-20-architecture-hdfs-federation-namenode-high-availability).  
**Secondary NameNode** — 1 нода на кластер. Принято говорить, что «Secondary NameNode» — это одно из самых неудачных названий за всю историю программ. Действительно, Secondary NameNode не является репликой NameNode. Состояние файловой системы хранится непосредственно в файле fsimage и в лог файле edits, содержащим последние изменения файловой системы (похоже на лог транзакций в мире РСУБД). Работа Secondary NameNode заключается в периодическом мерже fsimage и edits — Secondary NameNode поддерживает размер edits в разумных пределах. Secondary NameNode необходима для быстрого ручного восстанавления NameNode в случае выхода NameNode из строя.   
В реальном кластере NameNode и Secondary NameNode — отдельные сервера, требовательные к памяти и к жесткому диску. А заявленное “commodity hardware” — уже случай DataNode.   
**DataNode** — Таких нод в кластере очень много. Они хранят непосредственно блоки файлов. Нода регулярно отправляет NameNode свой статус (показывает, что еще жива) и ежечасно — репорт, информацию обо всех хранимых на этой ноде блоках. Это необходимо для поддержания нужного уровня репликации. 

# 31 Apache Spark RDD

Устойчивые распределенные наборы данных (Resilient Distributed Datasets (RDD)) - это фундаментальная структура данных Spark. Это неизменная распределенная коллекция объектов. Каждый набор данных(dataset) в RDD разделен на логические разделы(partitions), которые могут быть вычислены на разных узлах кластера. RDD могут содержать любые типы объектов Python, Java или Scala, включая пользовательские классы.

Формально RDD является доступным для чтения разделенным набором записей. RDD - это отказоустойчивый набор элементов, которые могут работать параллельно.

Существует два способа создания RDD - параллелизация существующей коллекции или ссылка на набор данных во внешней системе хранения, такой как файловая система общего доступа, HDFS или любой источник данных, предлагающий входной формат Hadoop.

Spark использует концепцию RDD для достижения более быстрых и эффективных операций MapReduce.

Обмен данными медленнее в MapReduce

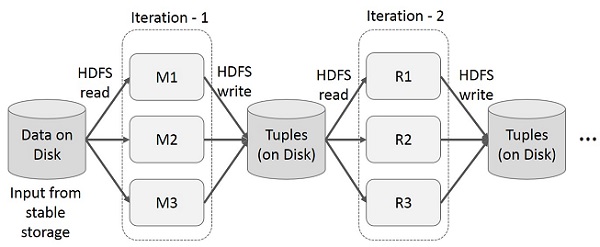
MapReduce широко применяется для обработки и создания больших наборов данных с параллельным распределенным алгоритмом на кластере. Это позволяет пользователям писать параллельные вычисления, используя набор операторов высокого уровня, не беспокоясь о распределении работы и отказоустойчивости.

К сожалению, в большинстве современных рамок единственным способом повторного использования данных между вычислениями (Ex - между двумя заданиями MapReduce) является запись его во внешнюю стабильную систему хранения (Ex - HDFS). Хотя эта структура обеспечивает многочисленные абстракции для доступа к вычислительным ресурсам кластера, пользователям по-прежнему требуется больше.

Оба Итерационных и интерактивные приложения требуют более быстрого обмена данных через параллельные задания. Обмен данными медленнее в MapReduce из-за репликации, сериализации и дискового ввода-вывода . Что касается системы хранения данных, то большинство приложений Hadoop они тратят более 90% времени на операции чтения и записи HDFS.

Итерационные операции на MapReduce

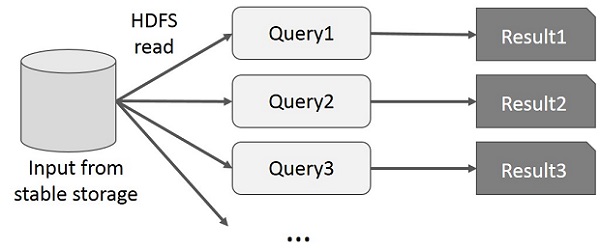
Повторное использование промежуточных результатов для нескольких вычислений в многоступенчатых приложениях. Следующая иллюстрация объясняет, как работает текущая структура, делая итеративные операции с MapReduce. Это приводит к существенным накладным расходам из-за репликации данных, дискового ввода-вывода и сериализации, что делает систему медленной.



Интерактивные операции на MapReduce

Пользователь запускает специальные запросы в одном и том же подмножестве данных. Каждый запрос будет выполнять операции ввода-вывода на жестком диске, что может доминировать над временем выполнения приложения.

На следующем рисунке показано, как работает текущая структура при выполнении интерактивных запросов в MapReduce.



Обмен данными с использованием Spark RDD

Обмен данными медленнее в MapReduce из-за репликации, сериализации и дискового ввода-вывода . В большинстве приложений Hadoop они тратят более 90% времени на операции чтения и записи HDFS.

Признавая эту проблему, исследователи разработали специализированную инфраструктуру Apache Spark. Основная идея R esilient D istributed D atasets (РДД) -  он поддерживает вычисления обработки в памяти. Это означает, что он сохраняет состояние памяти как объект на всех заданиях, и объект может быть разделен между этими заданиями. Обмен данными в памяти в 10-100 раз быстрее, чем в сети и на диске.

Попробуем теперь выяснить, как происходят итеративные и интерактивные операции в Spark RDD.

Итерационные операции на Spark RDD

На приведенной ниже иллюстрации показаны итерационные операции с Spark RDD. Он будет хранить промежуточные результаты в распределенной памяти вместо стабильного хранилища (Диск) и ускорить работу системы.

Примечание. Если распределенной памяти (ОЗУ) недостаточно для хранения промежуточных результатов (состояние задачи), то она сохранит эти результаты на диске.



Интерактивные операции на Spark RDD

На этом рисунке показаны интерактивные операции с Spark RDD. Если разные запросы повторяются в одном наборе данных, эти данные могут храниться в памяти для лучшего времени выполнения.



По умолчанию каждый преобразованный RDD может быть пересчитан каждый раз, когда вы запускаете на нем действие. Тем не менее, вы также можете сохранить RDD в памяти, и в этом случае Spark сохранит элементы в кластере для более быстрого доступа, при следующем запросе. Существует также поддержка постоянных RDD на диске или репликация на нескольких узлах.

# 32 Apache Spark RDD transformations and actions: filter, map, flatMap, distinct, reduce, groupBy. collect

Операции над RDD в Apache Spark подразделяются на два вида: трансформации (transformations) и действия (actions).

Различие: трансформации над RDD всегда возвращают в качестве результата новый RDD, а действия всегда возвращают другой тип данных (например, массив). Фишка спарка состоит в том, что трансформации не будут применены, пока не будет выполнено первое действие. Это называется «ленивые вычисления». Главный девиз – «ничё не будем вычислять, пока нам не понадобится этот результат» (действия как раз и направлены на получения результатов наших трансформаций, например, «Спарк, дай мне 5 элементов» - take(5)).

Примеры трансформаций:

* map(func) – к каждому элементу RDD применяется функция func (каждому элементу сопоставляется ровно один элемент на выходе)
* flatMap – map с последующим применением flatten (дробление результата): к каждому элементу применяется функция, при этом одному элементу на выходе соответствует множество значений (в т.ч. может и пустое множество). Результаты от каждого входа результируются в один RDD.

Пример

tmp = sc.parallelize(List(“Kirill”, “pisya”))

tmp.map(x => x) = (“Kirill”, “pisya”)

tmpt.flatMap(x => x) = ( “K”, “i”, “r”, “i”, “l”, “l”, “p”, “i”, “s”, “y”, “a”)

Помимо того, что применилась функция (каждый элемент в этот же элемент), в случае flatMap произошло ещё и дробление результата в множество.

* filter – возвращает новый RDD, содержащий только элементы, которые соответствуют предикату
* union – объединение двух RDD
* intersection – пересечение двух RDD
* distinct – возвращает новый RDD с уникальными элементами
* groupByKey – из RDD вида (ключ, значение) возвращает новый RDD с группировкой по ключу
* reduceByKey – из RDD вида (ключ, значение) возвращает новый RDD с применением reduce функции к значениям
* sortByKey – из RDD вида (ключ, значение) возвращает новый RDD с сортировкой по ключу

Примеры действий:

* count – возвращает кол-во элементов в RDD
* collect – возвращает весь RDD (в виде массива)
* take(n) – возвращает n элементов RDD
* top(n) – возвращает топ n элементов (если RDD не упорядочен нами, то берётся дефолтный порядок)
* reduce(func) – возвращает результат редуцирующей функции func того же типа, что и данные в RDD