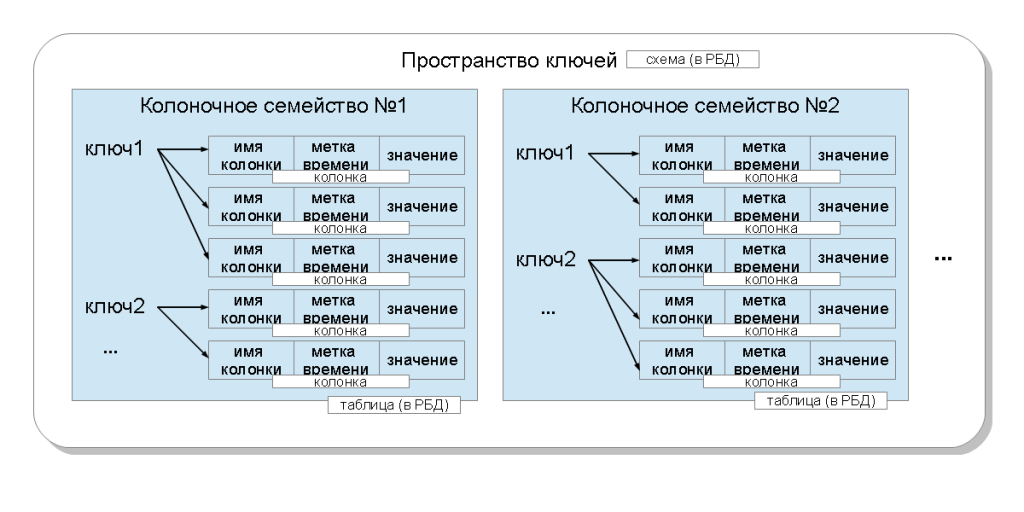
В этом топике я хотел бы рассказать о том, как устроена *кассандра* ([cassandra](http://cassandra.apache.org/)) — децентрализованная, отказоустойчивая и надёжная база данных “ключ-значение”. Хранилище само позаботится о проблемах наличия *единой точки отказа* ([single point of failure](http://en.wikipedia.org/wiki/Single_point_of_failure)), отказа серверов и о распределении данных между *узлами кластера* ([cluster node](http://www.datastax.com/docs/1.1/initialize/index)). При чем, как в случае размещения серверов в одном *центре обработки данных* ([data center](http://en.wikipedia.org/wiki/Data_center)), так и в конфигурации со многими центрами обработки данных, разделенных расстояниями и, соответственно, сетевыми задержками. Под надёжностью понимается *итоговая согласованность* ([eventual consistency](http://en.wikipedia.org/wiki/Eventual_consistency)) данных с возможностью установки *уровня согласования данных* ([tune consistency](http://www.datastax.com/docs/1.1/dml/data_consistency)) каждого запроса.  
  
[*NoSQL*](http://en.wikipedia.org/wiki/NoSQL) базы данных требуют в целом большего понимания их внутреннего устройства чем [*SQL*](http://en.wikipedia.org/wiki/SQL_database). Эта статья будет описывать базовое строение, а в следующих статьях можно будет рассмотреть: CQL и интерфейс программирования; техники проектирования и оптимизации; особенности кластеров размещённых в многих центрах обработки данных.

Модель данных

В терминологии кассандры приложение работает с *пространством ключей* ([keyspace](http://www.datastax.com/docs/1.1/ddl/keyspaces)), что соответствует понятию *схемы базы данных* ([database schema](http://en.wikipedia.org/wiki/Database_schema)) в реляционной модели. В этом пространстве ключей могут находиться несколько *колоночных семейств* ([column family](http://www.datastax.com/docs/1.1/ddl/column_family)), что соответствует понятию реляционной таблицы. В свою очередь, колоночные семейства содержат *колонки* ([column](http://www.datastax.com/docs/1.1/ddl/column_family" \l "standard-columns)), которые объединяются при помощи *ключа* ([row key](http://www.datastax.com/docs/1.1/ddl/about-data-model)) в *записи* ([row](http://www.datastax.com/docs/1.1/ddl/about-data-model)). Колонка состоит из трех частей: *имени* ([column name](http://www.datastax.com/docs/1.1/ddl/column_family" \l "standard-columns)), *метки времени* ([timestamp](http://www.datastax.com/docs/1.1/ddl/column_family" \l "standard-columns)) и *значения* ([value](http://www.datastax.com/docs/1.1/ddl/column_family" \l "standard-columns)). Колонки в пределах записи упорядочены. В отличие от реляционной БД, никаких ограничений на то, чтобы записи (а в терминах БД это строки) содержали колонки с такими же именами как и в других записях — нет. Колоночные семейства могут быть нескольких видов, но в этой статье мы будем опускать эту детализацию. Также в последних версиях кассандры появилась возможность выполнять *запросы определения и изменения данных* ([DDL](http://www.datastax.com/docs/1.1/ddl/index), [DML](http://www.datastax.com/docs/1.1/dml/index)) при помощи языка [*CQL*](http://www.datastax.com/docs/1.1/dml/using_cql), а также создавать *вторичные индексы* ([secondary indices](http://www.datastax.com/docs/1.1/ddl/indexes" \l "about-secondary-indexes)).  
  
  
Конкретное значение, хранимое в кассандре идентифицируется:

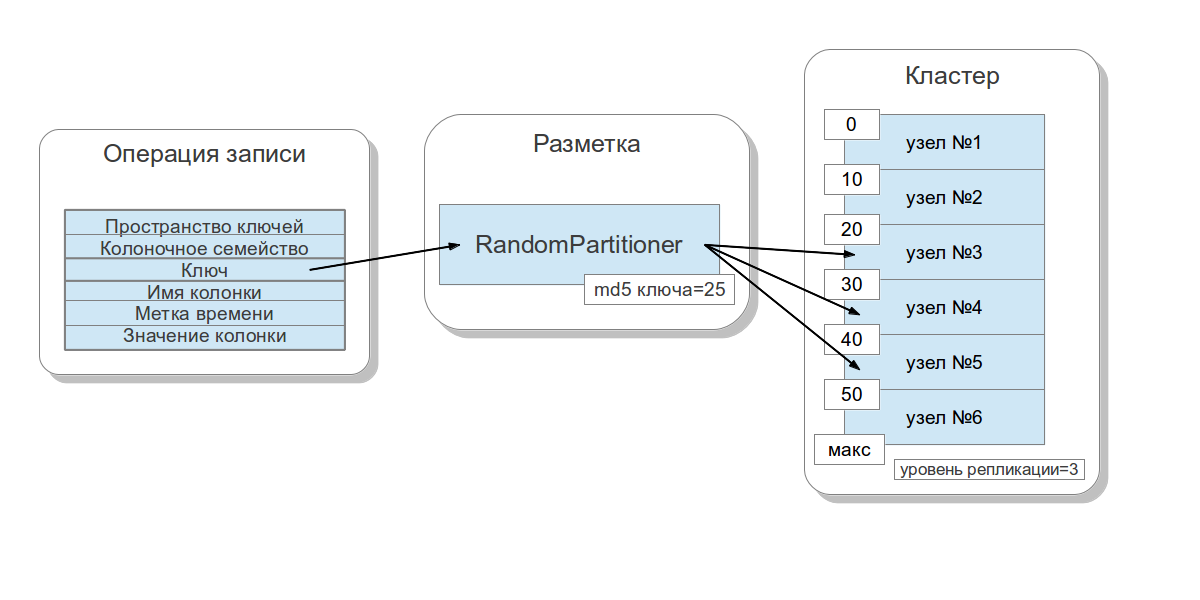
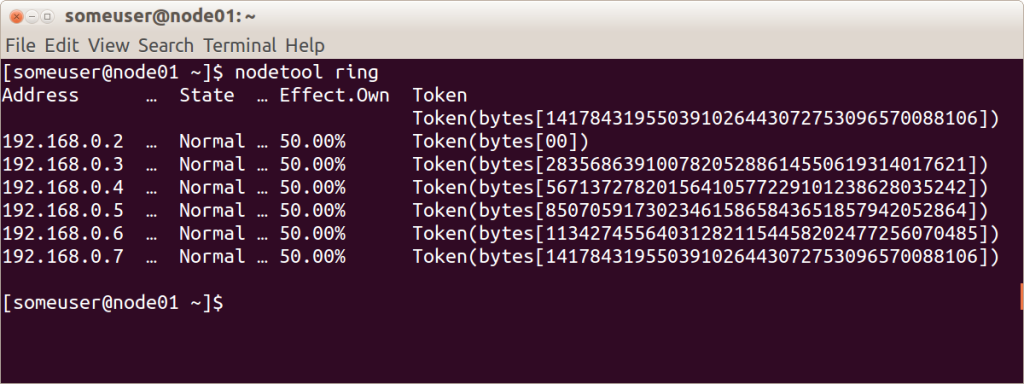
* пространством ключей — это привязка к приложению (предметной области). Позволяет на одном кластере размещать данные разных приложений;
* колоночным семейством — это привязка к запросу;
* ключом — это привязка к узлу кластера. От ключа зависит на какие узлы попадут сохранённые колонки;
* именем колонки — это привязка к атрибуту в записи. Позволяет в одной записи хранить несколько значений.

С каждым значением связана метка времени — задаваемое пользователем число, которое используется для разрешения конфликтов во время записи: чем больше число, тем колонка считается новее, а при сравнении перетирает старые колонки.  
  
По типам данных: пространство ключей и колоночное семейство — это строки (имена); метка времени — это 64-битное число; а ключ, имя колонки и значение колонки — это массив байтов. Также кассандра имеет понятие *типов данных* ([data type](http://www.datastax.com/docs/1.1/ddl/column_family" \l "about-data-types-comparators-and-validators)). Эти типы могут по желанию разработчика (опционально) задаваться при создании колоночного семейства. Для имён колонок это называется *сравнителем* ([comparator](http://www.datastax.com/docs/1.1/ddl/column_family" \l "about-data-types-comparators-and-validators)), для значений и ключей — *валидатором* ([validator](http://www.datastax.com/docs/1.1/ddl/column_family" \l "about-data-types-comparators-and-validators)). Первый определяет какие байтовые значения допустимы для имён колонок и как их упорядочить. Второй — какие байтовые значение допустимы для значений колонок и ключей. Если эти типы данных не заданы, то кассандра хранит значения и сравнивает их как *байтовые строки* ([BytesType](http://www.datastax.com/docs/1.1/ddl/column_family" \l "about-data-types-comparators-and-validators)) так как, по сути, они сохраняются внутри.  
  
Типы данных бывают такими:

* BytesType: любые байтовые строки (без валидации)
* AsciiType: ASCII строка
* UTF8Type: UTF-8 строка
* IntegerType: число с произвольным размером
* Int32Type: 4-байтовое число
* LongType: 8-байтовое число
* UUIDType: [*UUID*](http://en.wikipedia.org/wiki/UUID) 1-ого или 4-ого типа
* TimeUUIDType: UUID 1-ого типа
* DateType: 8-байтовое значение метки времени
* BooleanType: два значения: true = 1 или false = 0
* FloatType: 4-байтовое число с плавающей запятой
* DoubleType: 8-байтовое число с плавающей запятой
* DecimalType: число с произвольным размером и плавающей запятой
* CounterColumnType: 8-байтовый счётчик

В кассандре все операции записи данных это всегда операции перезаписи, то есть, если в колоночную семью приходит колонка с таким же ключом и именем, которые уже существуют, и метка времени больше, чем та которая сохранена, то значение перезаписывается. Записанные значения никогда не меняются, просто приходят более новые колонки с новыми значениями.  
  
Запись в кассандру работает с большей скоростью, чем чтение. Это меняет подход, который применяется при проектировании. Если рассматривать кассандру с точки зрения проектирования модели данных, то проще представить колоночное семейство не как таблицу, а как *материализованное представление* ([materialized view](http://en.wikipedia.org/wiki/Materialized_view)) — структуру, которая представляет данные некоторого сложного запроса, но хранит их на диске. Вместо того, чтобы пытаться как-либо скомпоновать данные при помощи запросов, лучше постараться сохранить в коночное семейство все, что может понадобиться для этого запроса. То есть, подходить необходимо не со стороны отношений между сущностями или связями между объектами, а со стороны запросов: какие поля требуются выбрать; в каком порядке должны идти записи; какие данные, связанные с основными, должны запрашиваться совместно — всё это должно уже быть сохранено в колоночное семейство. Количество колонок в записи ограничено теоретически 2 миллиардами. Это краткое отступление, а подробней — в статье о техниках проектирования и оптимизации. А теперь давайте углубимся в процесс сохранения данных в кассандру и их чтения.

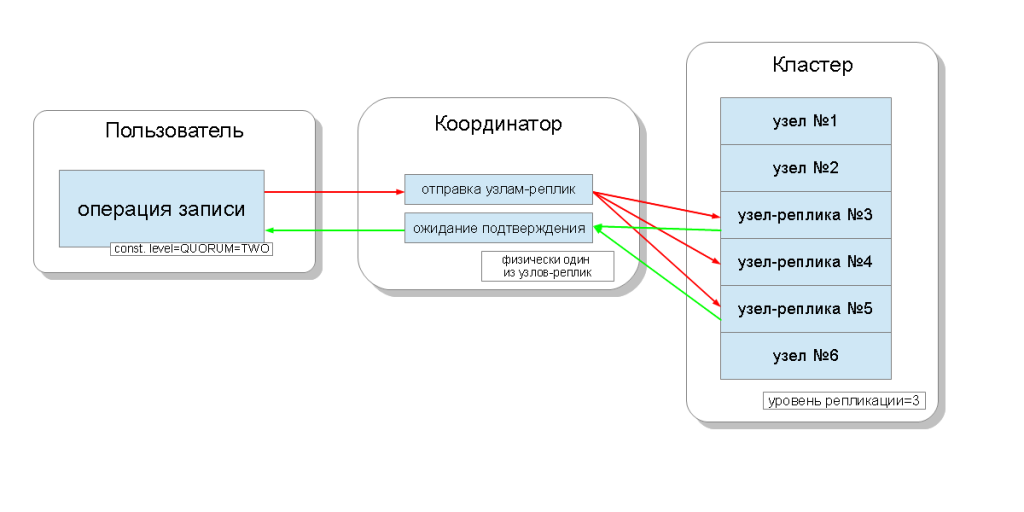
Распределение данных

Рассмотрим каким образом данные распределяются в зависимости от ключа по *узлам кластера* ([cluster nodes](http://www.datastax.com/docs/1.1/initialize/index)). Кассандра позволяет задавать стратегию распределения данных. Первая такая стратегия распределяет данные в зависимости от md5 значения ключа — *случайный разметчик* ([random partitioner](http://www.datastax.com/docs/1.1/cluster_architecture/partitioning" \l "about-the-randompartitioner)). Вторая учитывает само битовое представление ключа —*порядковый разметчик* ([byte-ordered partitioner](http://www.datastax.com/docs/1.1/cluster_architecture/partitioning" \l "about-the-byteorderedpartitioner)). Первая стратегия, в большинстве своем, даёт больше преимуществ, так как вам не нужно заботиться о равномерном распределение данных между серверами и подобных проблемах. Вторую стратегию используют в редких случаях, например если необходимы *интервальные запросы* ([range scan](http://www.datastax.com/docs/1.1/cluster_architecture/partitioning" \l "byteorderedpartitioner)). Важно заметить, что выбор этой стратегии производится перед созданием кластера и фактически не может быть изменён без полной перезагрузки данных.  
  
Для распределения данных кассандра использует технику, известную как *согласованное хеширование* ([consistent hashing](http://en.wikipedia.org/wiki/Consistent_hashing)). Этот подход позволяет распределить данные между узлами и сделать так, что при добавлении и удалении нового узла количество пересылаемых данных было небольшим. Для этого каждому узлу ставится в соответствие *метка* ([token](http://wiki.apache.org/cassandra/Operations" \l "Token_selection)), которая разбивает на части множество всех md5 значений ключей. Так как в большинстве случаев используется RandomPartitioner, рассмотрим его. Как я уже говорил, RandomPartitioner вычисляет 128-битный md5 для каждого ключа. Для определения в каких узлах будут храниться данные, просто перебираются все метки узлов от меньшего к большему, и, когда значение метки становится больше, чем значение md5 ключа, то этот узел вместе с некоторым количеством последующих узлов (в порядке меток) выбирается для сохранения. Общее число выбранных узлов должно быть равным *уровню репликации*([replication factor](http://www.datastax.com/docs/1.1/cluster_architecture/replication" \l "about-replication-in-cassandra)). Уровень репликации задаётся для каждого пространства ключей и позволяет регулировать*избыточность данных* ([data redundancy](http://en.wikipedia.org/wiki/Data_redundancy)).  
  
  
  
Перед тем, как добавить узел в кластер, необходимо задать ему метку. От того, какой процент ключей покрывает промежуток между этой меткой и следующей, зависит сколько данных будет храниться на узле. Весь набор меток для кластера называется *кольцом* ([ring](http://wiki.apache.org/cassandra/Operations" \l "Ring_management)).  
  
Вот иллюстрация отображающая при помощи встроенной утилиты *[nodetool](http://www.datastax.com/docs/1.1/references/nodetool)* кольцо кластера из 6 узлов с равномерно распределенными метками.  
  


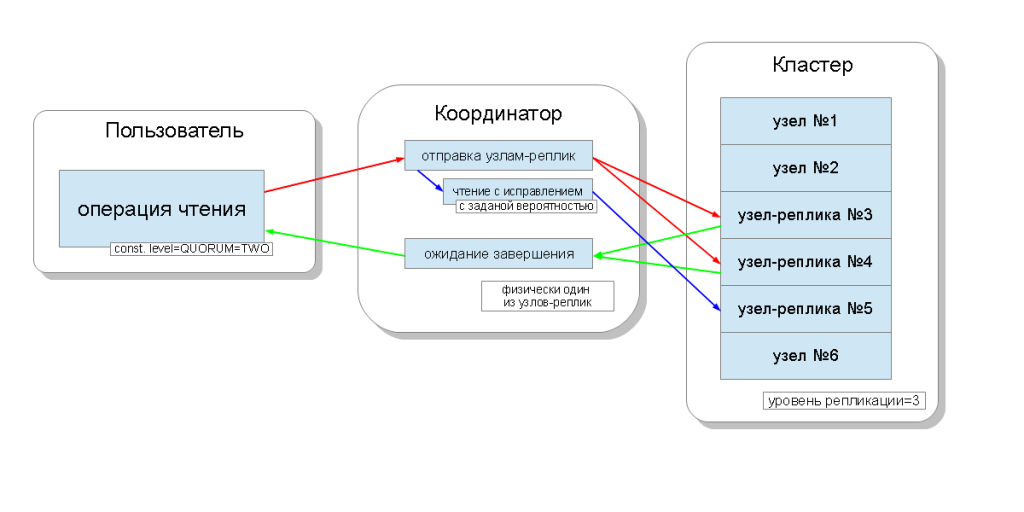
Согласованность данных

Узлы кластера кассандры равноценны, и клиенты могут соединятся с любым из них, как для записи, так и для чтения. Запросы проходят стадию координации, во время которой, выяснив при помощи ключа и разметчика на каких узлах должны располагаться данные, сервер посылает запросы к этим узлам. Будем называть узел, который выполняет координацию — *координатором* ([coordinator](http://www.datastax.com/docs/1.1/cluster_architecture/about_client_requests)), а узлы, которые выбраны для сохранения записи с данным ключом —*узлами-реплик* ([replica nodes](http://www.datastax.com/docs/1.1/cluster_architecture/about_client_requests" \l "about-write-requests)). Физически координатором может быть один из узлов-реплик — это зависит только от ключа, разметчика и меток.  
  
Для каждого запроса, как на чтение, так и на запись, есть возможность задать уровень согласованности данных.  
  
Для записи этот уровень будет влиять на количество узлов-реплик, с которых будет ожидаться подтверждение удачного окончания операции (данные записались) перед тем, как вернуть пользователю управление. Для записи существуют такие уровни согласованности:

* ONE — координатор шлёт запросы всем узлам-реплик, но, дождавшись подтверждения от первого же узла, возвращает управление пользователю;
* TWO — то же самое, но координатор дожидается подтверждения от двух первых узлов, прежде чем вернуть управление;
* THREE — аналогично, но координатор ждет подтверждения от трех первых узлов, прежде чем вернуть управление;
* QUORUM — собирается кворум: координатор дожидается подтверждения записи от более чем половины узлов-реплик, а именно round(N / 2) + 1, где N — уровень репликации;
* LOCAL\_QUORUM — координатор дожидается подтверждения от более чем половины узлов-реплик в том же центре обработки данных, где расположен координатор (для каждого запроса потенциально свой). Позволяет избавиться от задержек, связанных с пересылкой данных в другие центры обработки данных. Вопросы работы с многими центрами обработки данных рассматриваются в этой статье вскользь;
* EACH\_QUORUM — кооринатор дожидается подтверждения от более чем половины узлов-реплик в каждом центре обработки данных независимо;
* ALL — координатор дожидается подтверждения от всех узлов-реплик;
* ANY — даёт возможность записать данные, даже если все узлы-реплики не отвечают. Координатор дожидается или первого ответа от одного из узлов-реплик, или когда данные сохранятся при помощи *направленной отправки* ([hinted handoff](http://www.datastax.com/docs/1.1/dml/about_writes" \l "about-hinted-handoff-writes)) на координаторе.

  
  
Для чтения уровень согласованности будет влиять на количество узлов-реплик, с которых будет производиться чтение. Для чтения существуют такие уровни согласованности:

* ONE — координатор шлёт запросы к ближайшему узлу-реплике. Остальные реплики также читаются в целях *чтения с исправлением* ([read repair](http://www.datastax.com/docs/1.1/dml/data_consistency" \l "about-cassandra-s-built-in-consistency-repair-features)) с заданной в конфигурации кассандры вероятностью;
* TWO — то же самое, но координатор шлёт запросы к двум ближайшим узлам. Выбирается то значение, которое имеет большую метку времени;
* THREE — аналогично предыдущему варианту, но с тремя узлами;
* QUORUM — собирается кворум, то есть координатор шлёт запросы к более чем половине узлов-реплик, а именно round(N / 2) + 1, где N — уровень репликации;
* LOCAL\_QUORUM — собирается кворум в том центре обработки данных, в котором происходит координация, и возвращаются данные с последней меткой времени;
* EACH\_QUORUM — координатор возвращает данные после собрания кворума в каждом из центров обработки данных;
* ALL — координатор возвращает данные после прочтения со всех узлов-реплик.

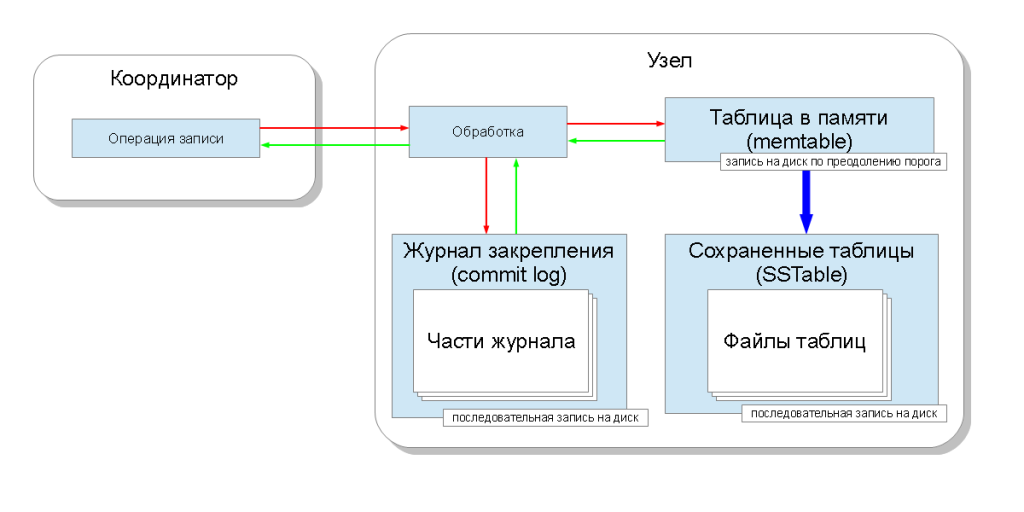
  
  
Таким образом, можно регулировать временные задержки операций чтения, записи и *настраивать согласованность* ([tune consistency](http://www.datastax.com/dev/blog/your-ideal-performance-consistency-tradeoff)), а также *доступность* ([availability](http://whatis.techtarget.com/definition/Reliability-Availability-and-Serviceability-RAS)) каждой из видов операций. По сути, доступность напрямую зависит от уровня согласованности операций чтения и записи, так как он определяет, сколько узлов-реплик может выйти из строя, и при этом эти операции все ещё будут подтверждены.  
  
Если число узлов, с которых приходит подтверждения о записи, в сумме с числом узлов, с которых происходит чтение, больше, чем уровень репликации, то у нас есть гарантия, что после записи новое значение всегда будет прочитано, и это называется *строгой согласованностью* ([strong consistency](http://www.datastax.com/docs/1.1/dml/data_consistency" \l "tunable-consistency-for-client-requests)). При отсутствии строгой согласованности существует возможность того, что операция чтения возвратит устаревшие данные.  
  
В любом случае, значение в конце концов распространится между репликами, но уже после того, как закончится координационное ожидание. Такое распространение называется *итоговой согласованностью* ([eventual consistency](http://en.wikipedia.org/wiki/Eventual_consistency)). Если не все узлы-реплики будут доступны во время записи, то рано или поздно будут задействованы средства восстановления, такие как чтение с исправлением и анти-энтропийное восстановление узла (anti-entropy node repair). Об этом чуть позже.  
  
Таким образом, при уровне согласованности QUORUM на чтение и на запись всегда будет поддерживаться строгая согласованность, и это будет некий баланс между задержкой операции чтения и записи. При записи ALL, а чтении ONE будет строгая согласованность, и операции чтения будут выполняться быстрее и будут иметь большую доступность, то есть количество вышедших из строя узлов, при котором чтение все еще будет выполнено, может быть большим, чем при QUORUM. Для операций записи же потребуются все рабочие узлы-реплик. При записи ONE, чтении ALL тоже будет строгая согласованность, и операции записи будут выполняться быстрее и доступность записи будет большой, ведь будет достаточно подтвердить лишь, что операция записи прошла хотя бы на одном из серверов, а чтение — медленней и требовать всех узлов-реплик. Если же к приложению нету требования о строгой согласованности, то появляется возможность ускорить и операции чтения и операции записи, а также улучшить доступность за счет выставления меньших уровней согласованности.

Восстановление данных

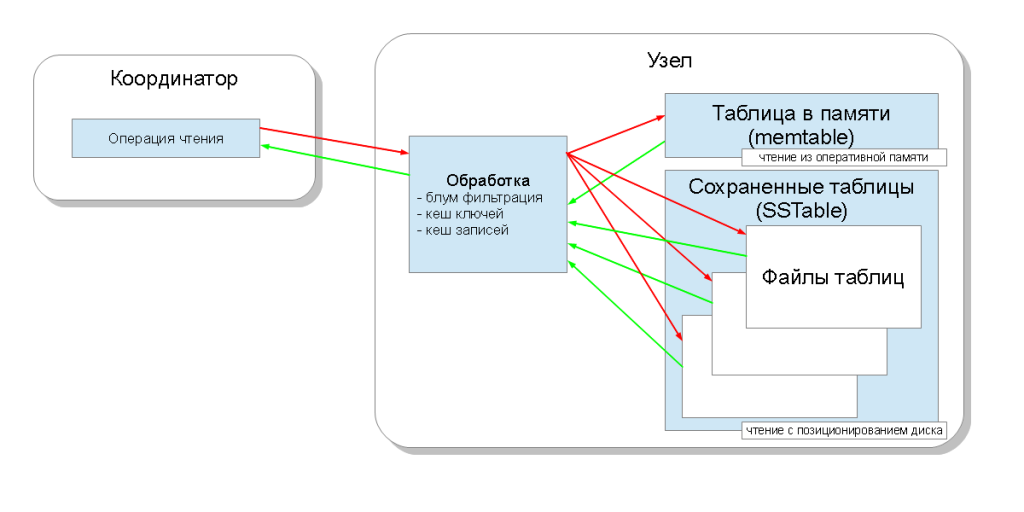
Кассандра поддерживает три механизма восстановления данных:

* *чтение с восстановлением* ([read repair](http://www.datastax.com/docs/1.1/dml/data_consistency" \l "about-cassandra-s-built-in-consistency-repair-features)) — во время чтения данные запрашиваются со всех реплик и сравниваются уже после завершения координации. Та колонка, которая имеет последнюю метку времени, распространится на узлы, где метки устаревшие.
* *направленной отправки* ([hinted handoff](http://www.datastax.com/docs/1.1/dml/data_consistency" \l "about-cassandra-s-built-in-consistency-repair-features)) — позволяет сохранить информацию об операции записи на координаторе в том случае, если запись на какой-либо из узлов не удалась. Позже, когда это будет возможно, запись повторится. Позволяет быстро производить восстановление данных в случае краткосрочного отсутствия узла в кластере. Кроме того, при уровне согласованности ANY позволяет добиться *полной доступности для записи* ([absolute write availability](http://www.datastax.com/docs/1.1/dml/about_writes" \l "about-hinted-handoff-writes)), когда даже все узлы-реплик недоступны, операция записи подтверждается, а данные сохранятся на узле-координаторе.
* *анти-энтропийное восстановление узла* ([anti-entropy node repair](http://www.datastax.com/docs/1.1/dml/data_consistency" \l "about-cassandra-s-built-in-consistency-repair-features)) — это некий процесс восстановления всех реплик, который должен запускаться регулярно вручную при помощи команды “nodetool repair” и позволяет поддержать количество реплик всех данных, которые возможно были не восстановлены первыми двумя способами, на требуемом уровне репликации.

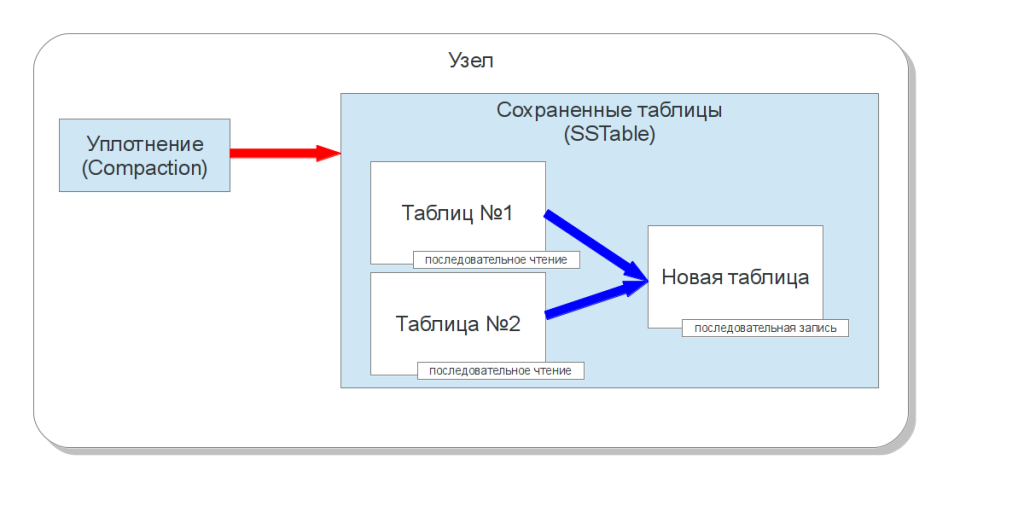
Запись на диск

Когда данные приходят после координации на узел непосредственно для записи, то они попадают в две структуры данных: в *таблицу в памяти* ([memtable](https://git-wip-us.apache.org/repos/asf?p=cassandra.git;a=blob_plain;f=src/java/org/apache/cassandra/db/Memtable.java;hb=refs/heads/cassandra-1.1)) и в *журнал закрепления* ([commit log](https://git-wip-us.apache.org/repos/asf?p=cassandra.git;a=blob_plain;f=src/java/org/apache/cassandra/db/commitlog/CommitLog.java;hb=refs/heads/cassandra-1.1)). Таблица в памяти существует для каждого колоночного семейства и позволяет запомнить значение моментально. Технически это *хеш-таблица* ([hashmap](http://en.wikipedia.org/wiki/Hashmap)) с возможностью *одновременного доступа* ([concurrent access](http://en.wikipedia.org/wiki/Concurrent_access)) на основе структуры данных, называемой *“списками с пропусками”* ([skip list](http://en.wikipedia.org/wiki/Skip_list)). Журнал закрепления один на всё пространство ключей и сохраняется на диске. Журнал представляет собой последовательность операций модификации. Так же он разбивается на части при достижении определённого размера.  
  
Такая организация позволяет сделать скорость записи ограниченной скоростью последовательной записи на жесткий диск и при этом гарантировать *долговечность данных* ([data durability](http://en.wikipedia.org/wiki/ACID" \l "Durability)). Журнал закрепления в случае аварийного останова узла читается при старте сервиса кассандры и восстанавливает все таблицы в памяти. Получается, что скорость упирается во время последовательной записи на диск, а у современных жёстких дисков это порядка 100МБ/с. По этой причине журнал закрепления советуют вынести на отдельный дисковый носитель.  
  
Понятно, что рано или поздно память может заполниться. Поэтому таблицу в памяти также необходимо сохранить на диск. Для определения момента сохранения существует *ограничение объёма занимаемыми таблицами в памяти*([memtable\_total\_spacein\_mb](http://www.datastax.com/docs/1.1/configuration/node_configuration" \l "memtable-total-space-in-mb)), по умолчанию это ⅓ максимального размера *кучи Java* ([Java heapspace](http://en.wikipedia.org/wiki/Java_Virtual_Machine_heap#Heap)). При заполнении таблицами в памяти объёма больше чем это ограничение, кассандра создает новую таблицу и записывает старую таблицу в памяти на диск в виде *сохраненной таблицы* ([SSTable](https://git-wip-us.apache.org/repos/asf?p=cassandra.git;a=blob_plain;f=src/java/org/apache/cassandra/io/sstable/SSTable.java;h=9a29066b8af7334be8fa9325fa480cee632abca0;hb=refs/heads/cassandra-1.1)). Сохранённая таблица после создания больше никогда *не модифицируется* ([is immutable](http://en.wikipedia.org/wiki/Immutable)). Когда происходит сохранение на диск, то части журнала закрепления помечаются как свободные, таким образом освобождая занятое журналом место на диске. Нужно учесть, что журнал имеет переплетённую структуру из данных разных колоночных семейств в пространстве ключей, и какие-то части могут быть не освобождены, так как некоторым областям будут соответствовать другие данные, все ещё находящиеся в таблицах в памяти.  
  
  
  
В итоге, каждому колоночному семейству соответствует одна таблица в памяти и некоторое число сохранённых таблиц. Теперь, когда узел обрабатывает запрос чтения, ему необходимо запросить все эти структуры и выбрать самое последнее по метке времени значение. Для ускорения этого процесса существует три механизма: *блум-фильтрация* ([bloom filter](http://en.wikipedia.org/wiki/Bloom_filter)), *кэш ключей* ([key cache](http://www.datastax.com/docs/1.1/operations/tuning" \l "tuning-data-caches)) и *кэш записей* ([record cache](http://www.datastax.com/docs/1.1/operations/tuning" \l "tuning-data-caches)):

* блум-фильтр — это структура данных, которая занимает немного места и позволяет ответить на вопрос: содержится ли элемент, а в нашем случае это ключ, в множестве или нет. При чем, если ответ — “нет”, то это 100%, а если ответ “да”, то это, возможно, ложно-положительный ответ. Это позволяет уменьшить количество чтений из сохранённых таблиц;
* кэш ключей сохраняет позицию на диске записи для каждого ключа, таким образом уменьшая количество *операций позиционирования* ([seek operations](http://en.wikipedia.org/wiki/Seek_time" \l "Seek_time)) во время поиска по сохранённой таблице;
* кэш записей сохраняет запись целиком, позволяя совсем избавиться от операций чтения с диска.



Уплотнение

В определенный момент времени данные в колоночном семействе перезапишутся — придут колонки, которые будут иметь те же имя и ключ. То есть, возникнет ситуация, когда в более старой сохранённой таблице и более новой будут содержаться старые и новые данные. Для того, чтобы гарантировать целостность, кассандра обязана читать все эти сохранённые таблицы и выбирать данные с последней меткой времени. Получается, что количество операций позиционирования жёсткого диска при чтении пропорционально количеству сохранённых таблиц. Поэтому для того, чтобы освободить перезаписанные данные и уменьшить количество сохранённых таблиц, существует *процесс уплотнения*([compaction](http://www.datastax.com/docs/1.1/operations/tuning" \l "tuning-compaction)). Он читает последовательно несколько сохранённых таблиц и записывает новую сохранённую таблицу, в которой объединены данные по меткам времени. Когда таблица полностью записана и введена в использование, кассандра может освободить таблицы-источники(таблицами, которые её образовали). Таким образом, если таблицы содержали перезаписанные данные, то эта избыточность устраняется. Понятно, что во время такой операции объем избыточности увеличивается — новая сохранённая таблица существует на диске вместе с таблицами-источниками, а это значит, что объем места на диске всегда должен быть такой, чтобы можно было произвести уплотнение.  
  
  
  
Кассандра позволяет выбрать одну из двух стратегий проведения уплотнения:

* стратегия *уплотнения сохраненных таблиц связанных размером* ([size-tiered compaction](http://www.datastax.com/docs/1.1/operations/tuning" \l "tuning-compaction-for-size-tiered-compaction)) — эта стратегия уплотняет определенным образом выбранные две таблицы. Применяется автоматически в виде *фонового уплотнения* ([minor compaction](http://www.datastax.com/docs/1.1/operations/tuning" \l "tuning-compaction-for-size-tiered-compaction)) и в ручном режиме, для *полного уплотнения* ([major compaction](http://www.datastax.com/docs/1.1/operations/tuning" \l "tuning-compaction-for-size-tiered-compaction)). Допускает ситуацию нахождения ключа во многих таблицах и, соответственно, требует выполнять операцию поиска для каждой такой таблицы.
* стратегия *уплотнение сохраненных таблиц уровнями* ([leveled compaction](http://www.datastax.com/dev/blog/when-to-use-leveled-compaction)) — уплотняет сохраненные таблицы, которые изначально создаются небольшими — 5 МБ, группируя их в уровни. Каждый уровень в 10 раз больший чем предыдущий. Причем, существуют такие гарантии: 90% запросов чтения будут происходить к одной сохраненной таблице, и только 10% пространства на диске будет использоваться под устаревшие данные. В этом случае для выполнения уплотнения под временную таблицу достаточно только 10-кратного размера таблицы, то есть 50 Мб. Подробнее в этой [статье](http://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra)

Операции удаления

С точки зрения внутреннего устройства, операции удаление колонок — это операции записи специального значения —*затирающего значения* ([tombstone](http://www.datastax.com/docs/1.1/dml/about_writes" \l "about-deletes)). Когда такое значение получается в результате чтения, то оно пропускается, словно такого значения никогда и не существовало. В результате же уплотнения, такие значения постепенно вытесняют устаревшие реальные значения и, возможно, исчезают вовсе. Если же появятся колонки с реальными данными с еще более новыми метками времени, то они перетрут, в конце концов, и эти затирающие значения.

Транзакционность

Кассандра поддерживает транзакционность на уровне одной записи, то есть для набора колонок с одним ключом. Вот как выполняются четыре требования ACID:

* *атомарность* ([atomicity](http://en.wikipedia.org/wiki/ACID" \l "Atomicity)) — все колонки в одной записи за одну операцию будут или записаны, или нет;
* *согласованность* ([consistency](http://en.wikipedia.org/wiki/ACID" \l "Consistency)) — как уже было сказано выше, есть возможность использовать запросы с строгой согласованностью взамен доступности, и тем самым выполнить это требование;
* *изолированность* ([isolation](http://en.wikipedia.org/wiki/ACID" \l "Isolation)) — начиная с кассандры версии 1.1, появилась поддержка изолированности, когда во время записи колонок одной записи другой пользователь, который читает эту же запись, увидит или полностью старую версию записи или, уже после окончания операции, новую версию, а не часть колонок из одной и часть из второй;
* *долговечность* ([durability](http://en.wikipedia.org/wiki/ACID" \l "Durability)) обеспечивается наличием журнала закрепления, который будет воспроизведён и восстановит узел до нужного состояния в случае какого-либо отказа.

Послесловие

Итак, мы рассмотрели как устроены основные операции — чтение и запись значений в кассандру.  
  
В дополнение, важные, по моему мнению, ссылки:

* Страница на apache.org [cassandra.apache.org](http://cassandra.apache.org/)
* Документация на datastax.com [www.datastax.com/docs/1.1/index#apache-cassandra-1-1-documentation](http://www.datastax.com/docs/1.1/index#apache-cassandra-1-1-documentation)
* Про BigTable [research.google.com/archive/bigtable.html](http://research.google.com/archive/bigtable.html)
* Про Amazon Dynamo [www.allthingsdistributed.com/2007/10/amazons\_dynamo.html](http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html)
* Просто один из блогов, про модель данных [maxgrinev.com/2010/07/09/a-quick-introduction-to-the-cassandra-data-model](http://maxgrinev.com/2010/07/09/a-quick-introduction-to-the-cassandra-data-model/)

Просьба: замечания по орфографии и идеи по улучшению статьи высказывайте в личном сообщении.

# Моделирование данных в Cassandra 2.0 на CQL3 tutorial

[Big Data](https://habrahabr.ru/hub/bigdata/)\*, [NoSQL](https://habrahabr.ru/hub/nosql/" \o "Вы не подписаны на этот хаб)\*

*Статья предназначена для людей пытающихся создать свою первую «таблицу» в БД Cassandra.*  
  
За посление несколько релизов Кассандры разработчики взяли правильный вектор направленный на простоту использования этой базы данных. Учитывая её достоинства, такие как скорость работы и отказоустойчиваость, её было сложно как администрировать, так и писать под неё. Сейчас же количество танцев с бубном, которые надо провести прежде чем запустить и начать разрабатывать, свели к минимуму — несколько комманд в bash или один .msi в Windows.  
Более того, сильно облегчил жизнь разработчикам недавно обновлённый CQL (язык запросов), вытеснив бинарный и довольно сложный язык Thrift.  
Лично я столкнулся с проблемой наличия отсуствия русскоязычных руководств по Кассандре. Самую, на мой взгляд, сложную тему мне бы хотелось поднять в этой статье. *Как же дизайнить базу данных то?*

##### **Disclaimer**

* Статья НЕ предназначена для людей, которые впервые видят слово Cassandra.
* Статья НЕ служит как рекламный материал той или иной технологии.
* Статья НЕ стремится доказать что-либо кому-либо.
* Если скорость записи/чтения не так важна, и если «100% uptime» не сильно нужен, и если у вас всего лишь несколько миллионов записей, то, вероятно, эта статья, да и вся Cassandra в целом, — не то, что вам нужно.

#### **Ликбез**

* [**Cassandra**](http://cassandra.apache.org/) (далее **C\***) — распределённая NoSQL БД, поэтому все решения «почему так, а не вот так» всегда принимаются с оглядкой на кластеризацию.
* [**CQL**](http://cassandra.apache.org/doc/cql3/CQL.html) — это SQL-подобный язык. Аббревиатура от **C**assandra **Q**uery **L**anguage.
* **Node** (нода) — инстанс C\*, или java процесс в терминах операционных систем. На одной машине можно запустить несколько нод, например.
* Основная единица хранения — **строка**. Строка целиком хранится на нодах, т.е. нет ситуаций когда полстроки — на одной ноде, полстроки — на другой. Строка может динамически раширяться до 2 миллиардов колонок. Это важно.
* **cqlsh** — коммандная строка для CQL. Все примеры ниже выполняются именно в ней. Является частью дистрибутива C\*.

#### **Основное правило моделирования данных в C\***

Кассандра создавалась как распределённая БД с упором на максимальную скорость записи и чтения. Моделировать «таблицы» нужно в зависимости от SELECT запросов вашего приложения.  
В SQL мы привыкли накидать таблиц, связей между ними, и потом уже **SELECT ... JOIN ...** чего хотим и как хотим. Именно JOIN-ы основная проблема с произвоидтельностью в RDBMS. Их нет в CQL.

#### **Первый пример.**

У нас есть сотрудники какой-то компании. Создадим таблицу (которые на самом деле называются Column Family, но для простоты перехода с SQL на CQL используют слово table) на CQL и заполним данными:

**CREATE** **TABLE** employees (

**name** text, *-- уникальное имя*

age int, *-- какие-то данные про человека*

**role** text, *-- ещё какие-то данные*

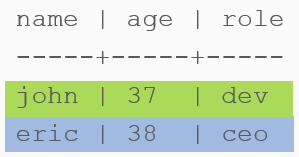
PRIMARY **KEY** (**name**)); *-- обязательная часть любой таблицы*

**INSERT** **INTO** employees (**name**, age, **role**) **VALUES** ('john', 37, 'dev');

**INSERT** **INTO** employees (**name**, age, **role**) **VALUES** ('eric', 38, 'ceo');

Таблицы в C\* обязаны иметь PRIMARY KEY. Он используется для поиска ноды, в которой хранится искомая строка.  
  
Прочитаем данные:

**SELECT** \* **FROM** employees;

Эта картинка — руками разукрашенный вывод cqlsh.  
  
  
Выглядит как обычная таблица из реляционной БД. C\* создаст две строки.  
  
Внимание! Это две **внутренние структуры строк**, а не таблицы. Если чуть слукавить, то можно сказать, что каждая строка — это как маленькая таблица. Далее понятней.

#### **Второй пример.**

Усложняем. Добавим название компании.

**CREATE** **TABLE** employees (

company text,

**name** text,

age int,

**role** text,

PRIMARY **KEY** (company,**name**) *-- две части главного ключа: распределительный ключ company и кластерный ключ name*

);

**INSERT** **INTO** employees (company, **name**, age, **role**) **VALUES** ('OSC', 'eric', 38, 'ceo');

**INSERT** **INTO** employees (company, **name**, age, **role**) **VALUES** ('OSC', 'john', 37, 'dev');

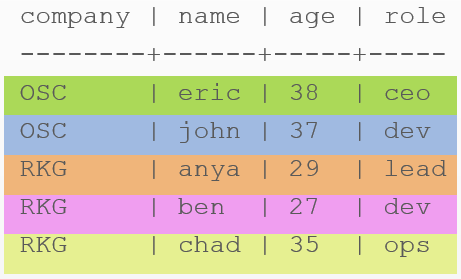
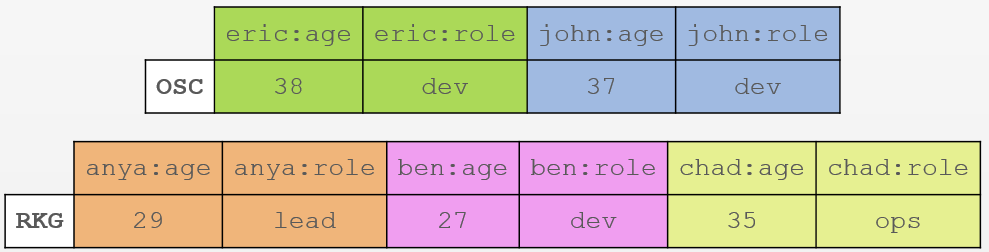
**INSERT** **INTO** employees (company, **name**, age, **role**) **VALUES** ('RKG', 'anya', 29, 'lead');

**INSERT** **INTO** employees (company, **name**, age, **role**) **VALUES** ('RKG', 'ben', 27, 'dev');

**INSERT** **INTO** employees (company, **name**, age, **role**) **VALUES** ('RKG', 'chan', 35, 'ops');

Прочитаем данные:

**SELECT** \* **FROM** employees;

  
Внимание на PRIMARY KEY. **Первый** из параметров — company — это распределительный ключ, именно он будет использоваться для поиска ноды с этих пор. Второй ключ name — это кластерный ключ (clustering key). Он превращается в колонку. Т.е. мы данные превращаем в название колонки. Был 'eric' обычными четырмя байтами, а стал частью названия колонки.  
  
Вот так теперь выглядит **внутреняя структура**.  
  
Как видите у нас:

* Две компании — OSC и RKG. Здесь создалось всего две строки.
* Зелёный eric хранит свой возраст и роль в двух ячейках. Аналогично все остальные.
* Получается с такой структурой мы можем хранить 1 млрд сотрудников в каждой компании (строке). Помним же, что лимит количества колонок — 2 млрд?
* Может показаться, что мы лишний раз храним одни и те же данные. Это так, но в C\* такой дизайн — правильный паттерн моделирования.
* **Расширять строки — это основная фича при моделировании в С\*.**

#### **Третий пример.**

Ещё сложнее. Заглавная буква — название колонки. Строчная — данные.

**CREATE** **TABLE** example (

A text,

B text,

**C** text,

**D** text,

**E** text,

**F** text,

PRIMARY **KEY** ((A,B), **C**, **D**)); *-- составной распределительный ключ (A,B) и кластерные ключи (C,D)*

**INSERT** **INTO** example (A, B, **C**, **D**, **E**, **F**) **VALUES** ('a', 'b', 'c', 'd', 'e', 'f');

**INSERT** **INTO** example (A, B, **C**, **D**, **E**, **F**) **VALUES** ('a', 'b', 'c', 'g', 'h', 'i');

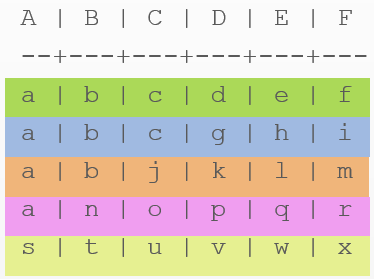
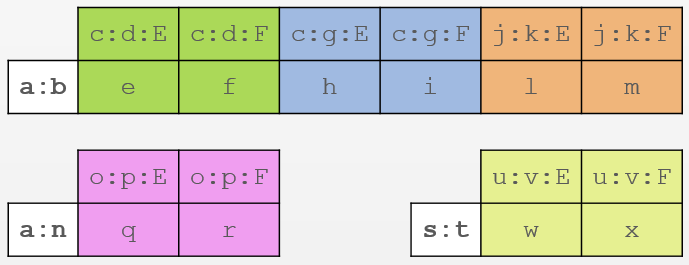
**INSERT** **INTO** example (A, B, **C**, **D**, **E**, **F**) **VALUES** ('a', 'b', 'j', 'k', 'l', 'm');

**INSERT** **INTO** example (A, B, **C**, **D**, **E**, **F**) **VALUES** ('a', 'n', 'o', 'p', 'q', 'r');

**INSERT** **INTO** example (A, B, **C**, **D**, **E**, **F**) **VALUES** ('s', 't', 'u', 'v', 'w', 'x');

Прочитаем данные:

**SELECT** \* **FROM** example;

  
  
Теперь наш распределительный ключ составной — (A,B). Кластерный ключ тоже составной — C, D.  
  
Внутрення структура усложнилась. Такие данные как c, d, g, k, o, p, u, v участвуют в названии колонок наравне с E и F:  


* Как видите, теперь каждая уникальная комбинация A и B — это ключ к строке.
* У нас всего три уникальных распределительных ключа — a:b, a:n и s:t.
* Колонки же размножились благодаря кластерным ключам. В строке a:b у нас три уникальных комбинации — c:d,c:g, j:k — которые хранят в колонках E и F собственно данные — e и f, h и i, l и m.
* Аналогично две другие строки.

#### **Почему так сложно?**

Это самый быстрый способ записи и хранения бесконечного количества данных в распределённой БД. C\* как раз была разработана с упором на скорость записи/чтения. Вот, например, [сравнение скоростей MongoDB, HBase и С\*](http://www.datastax.com/wp-content/uploads/2013/02/WP-Benchmarking-Top-NoSQL-Databases.pdf).

#### **Пример из реальной жизни**

У нас есть некие события, которые происходят 1000 раз в секунду. Например с датчиков уровня шума снимаются показатели. 10 датчиков. Каждый из них присылает данные 100 раз в секунду. У нас 3 задачи:

1. Продолжать записывать, если сервер БД (нода) остановит свою работу.
2. Успевать записывать 1000 новых записей в секуду несмотря ни на что.
3. Предоставлять график любого датчика за любой день за пару-тройку миллисекунд.
4. Предоставлять график любого датчика за любой промежуток времени как можно быстрее.

##### **Первый и второй пункты — легко.**

Нам нужно установить несколько нод, сделать каждую автономной. Может даже вынести одну из них в облако.

##### **Третий пункт — основная хитрость.**

Мы будем хранить данные одного дня в одной строке.

**CREATE** **TABLE** temperature\_events\_by\_day (

**day** text, *-- Text of the following format: 'YYYY-MM-DD'*

sensor\_id **uuid**,

event\_time **timestamp**,

temperature **double**,

PRIMARY **KEY** ((**day**,sensor\_id), event\_time) *-- составной распред. ключ (day,sensor\_id) и кластерный ключ (event\_time)*

)

**WITH** **CLUSTERING** **ORDER** **BY** event\_time **DESC**; *-- обратная сортировка записываемых данных*

Так как распределительным ключом является уникальная комбинация день+датчик, то данные за один день будут храниться для каждого датчика в отдельной строке. Благодаря обратной сортировке внутри строки мы получаем самые важные для нас данные (последние) «на кончике пальцев».  
Так как поиск распределительного ключа (дня) — очень быстрая операция в С\*, то третий пункт можно считать выполненным.

##### **Четвертый пункт**

Конечно, мы можем сделать поиск дня/дней, а внутри дня уже сравнивать timestamp. Но дней может быть очень много.  
У нас ведь всего 10 датчиков. Нельзя ли этим воспользоваться? Можно, если представить, что один датчик — одна строка. В этом случае С\* закеширует в памяти местоположение всех десяти строк на диске.  
  
Создадим вторую таблицу, где будем хранить те же самые данные, но без учета дней.

**CREATE** **TABLE** temperature\_events (

sensor\_id **uuid**,

event\_time **timestamp**,

temperature **double**,

PRIMARY **KEY** (sensor\_id, event\_time) *-- распределительный ключ (sensor\_id) и кластерный ключ (event\_time)*

)

**WITH** **CLUSTERING** **ORDER** **BY** event\_time **DESC**; *-- обратная сортировка записываемых данных*

И когда будем вставляеть данные, то ограничим время жизни каждой ячейки чтобы не привысить 2 млрд колонок. У нас каждый датчик даёт не более 100 показаний в секунду. Отсюда:  
2\*\*31 / (24 часа \* 60 мин \* 60 сек \* 100 событий/сек) = 2147483648 / (24 \* 60 \* 60 \* 100) = 248.55 дней  
Надо сделать, чтобы через 248 дней самые старые данные тихо и незаметно самоудалялись.

**INSERT** **INTO** temperature\_events (sensor\_id, event\_time, temperature)

**VALUES** ('12341234-1234-1234-123412', 2535726623061, 36.6)

TTL 21427200; *-- 248 days in seconds*

В коде приложения нужно будет поставить условие, что если запрашиваемые данные выходят за границы последних 248 дней, то используем таблицу temperature\_events\_by\_day, если нет — temperature\_events. Поиск по последней будет на несколько миллисекунд быстрее.  
  
«Что за бред! Зачем вторую таблицу?» — подумаете вы. Повторюсь: в БД С\* хранить одно и то же значение по нескольку раз — это норма, правильная модель. Выигрыши следующие:

* Запись данных во вторую таблицу быстрее, чем в первую. Кассандре не придётся искать ноду(-ы) в которую бы сложить новое значение. Она будет знать заранее.
* Чтение данных тоже очень быстрое. Например в разы превосходит обычную индексированную, нормированную SQL БД.