CSCI E-88 Principles Of Big Data Processing

Harvard University Extension, Fall 2019

Marina Popova

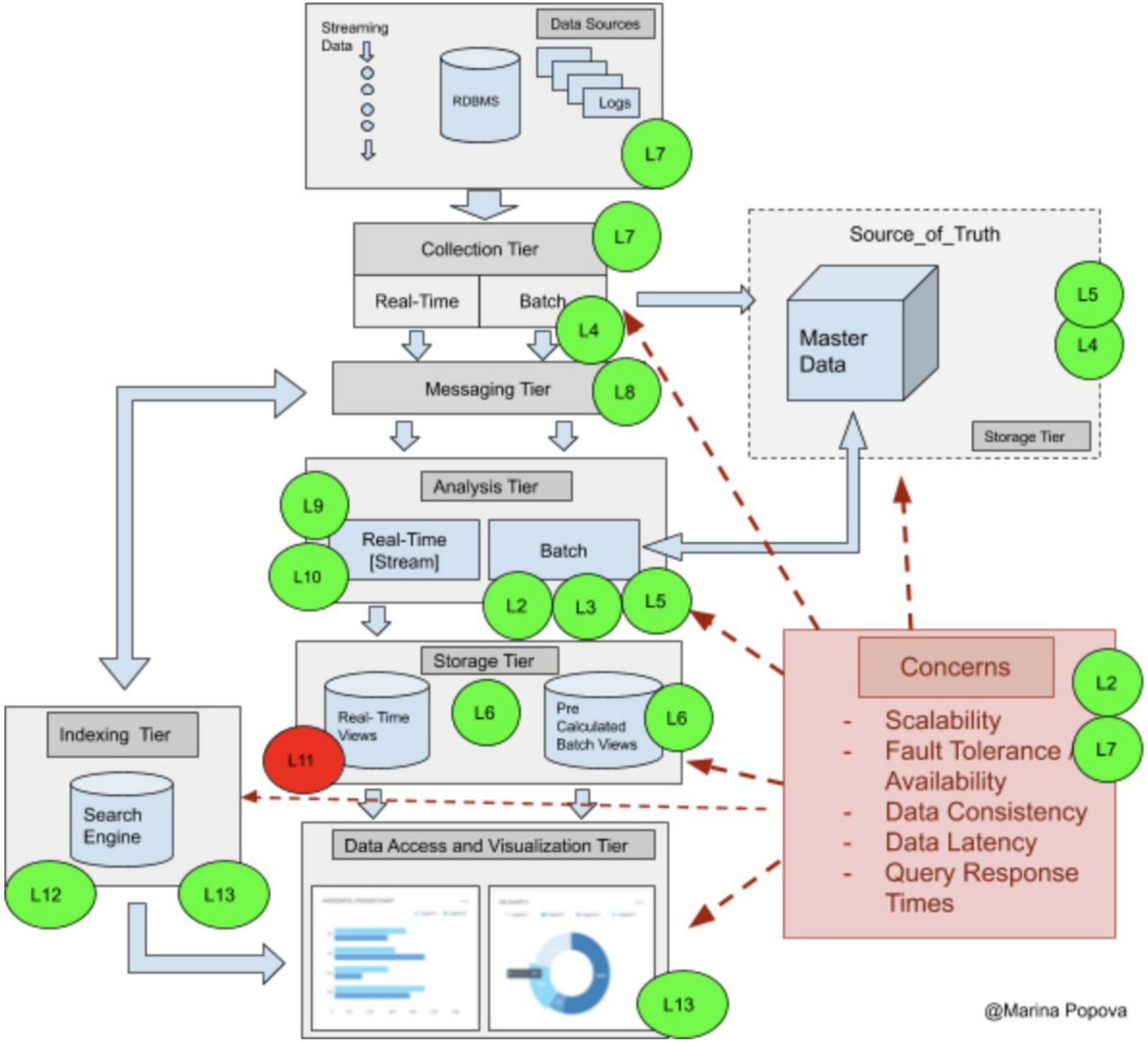
Lecture 11 Time Series Processing and Cassandra

@Marina Popova

Agenda

- Time series processing and DBs
- Real-Time Views
- Cassandra Architecture
- Cassandra Data Modeling

Where Are We?



@Marina Popova

Time-Series Processing and DBs

What is Time-Series Data ?

- any data that includes a time component
- data that arrives in time order
- data that collectively represents how a system/process/behavior changes over time
- thus, time-series datasets are usually **append-only**

Ref: https://blog.timescale.com/blog/what-the-heck-is-time-series-data-and-why-do-i-need-a-time-series-database-dcf3b1b18563/

Time-Series Processing and DBs

What are Time-Series DBs ?

- DBs optimized for the time-series data ingestion, storage and analytics
- Main characteristics:
  - time is the main axis / dimension of the data
  - each incoming data event is stored as an immutable data point in time --> thus:
  - time-series DBs track every change/event as INSERTs , not UPDATEs

"This practice of recording each and every change to the system as a new, different row is what makes time-series data so powerful. It allows us to ***measure change****:*

- *analyze how something changed in the past*
- *monitor how something is changing in the present*
- *predict how it may change in the future."*

Time-Series Processing and DBs

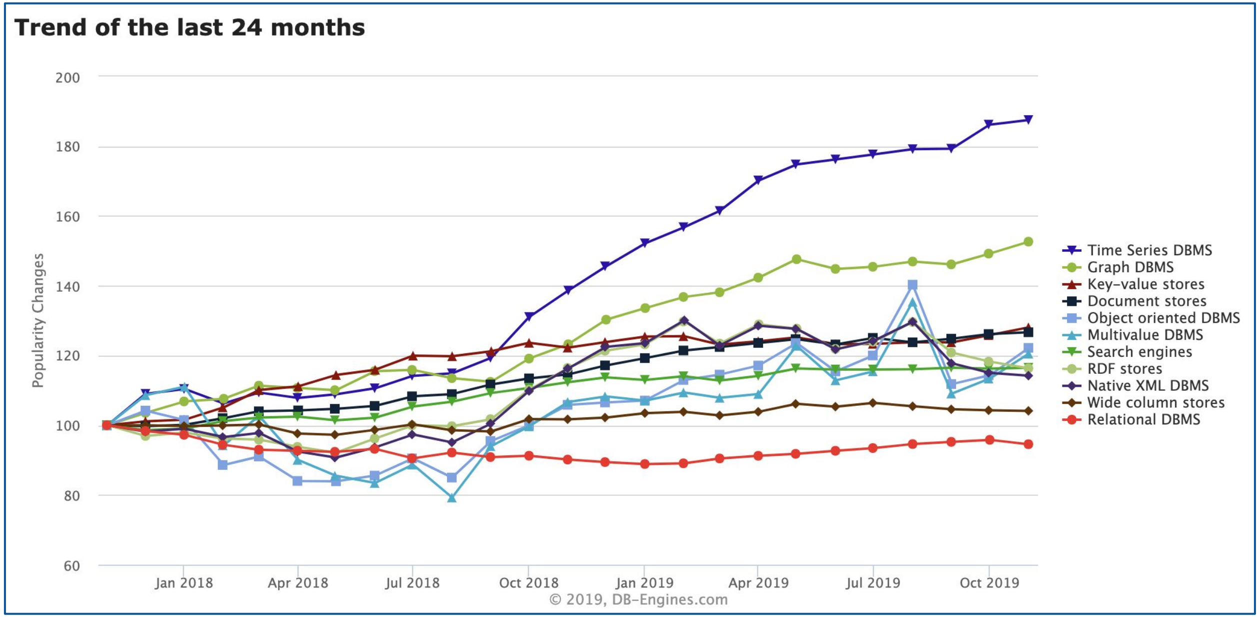
Benefits of Time-Series DBs

Arguably, the benefits are:

- purpose-built DBs to handle huge amounts of data
- faster ingestion and compression
- typically include functions and operations common to time-series data analysis such as:
    - data retention policies
    - continuous queries
    - flexible time aggregations

Time-Series DBs: popularity contest

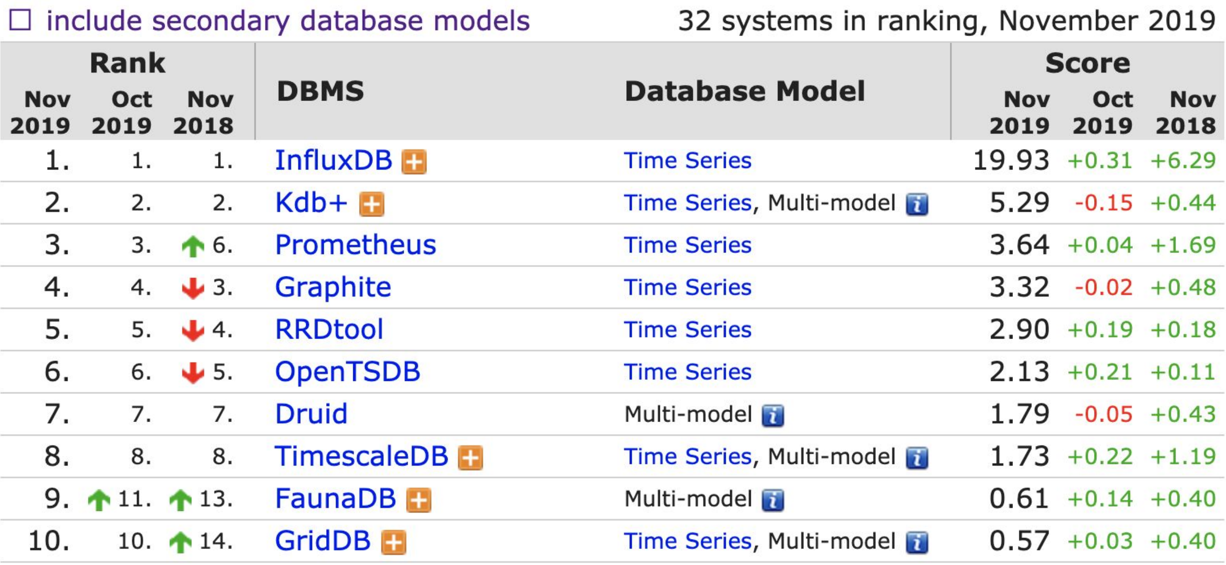
Ref: https://db-engines.com/en/ranking_categories



@Marina Popova

Time-Series DBs: Landscape

Ref:



☐ include secondary database models

32 systems in ranking, November 2019

Rank			DBMS	Database Model	Score		
Nov 2019	Oct 2019	Nov 2018			Nov 2019	Oct 2019	Nov 2018
1.	1.	1.	InfluxDB	Time Series	19.93	+0.31	+6.29
2.	2.	2.	Kdb+	Time Series, Multi-model	5.29	-0.15	+0.44
3.	3.	↑ 6.	Prometheus	Time Series	3.64	+0.04	+1.69
4.	4.	↓ 3.	Graphite	Time Series	3.32	-0.02	+0.48
5.	5.	↓ 4.	RRDtool	Time Series	2.90	+0.19	+0.18
6.	6.	↓ 5.	OpenTSDB	Time Series	2.13	+0.21	+0.11
7.	7.	7.	Druid	Multi-model	1.79	-0.05	+0.43
8.	8.	8.	TimescaleDB	Time Series, Multi-model	1.73	+0.22	+1.19
9.	↑ 11.	↑ 13.	FaunaDB	Multi-model	0.61	+0.14	+0.40
10.	10.	↑ 14.	GridDB	Time Series, Multi-model	0.57	+0.03	+0.40

@Marina Popova

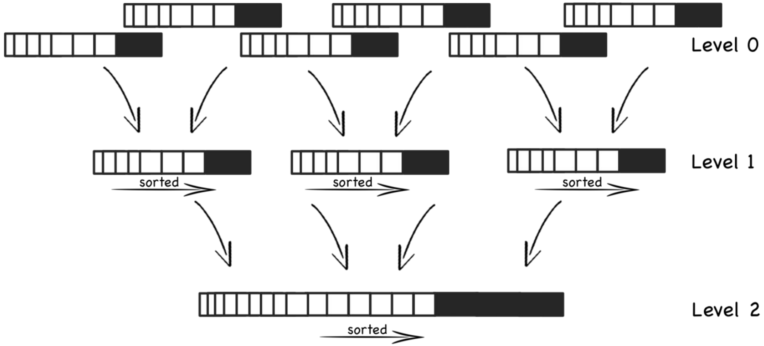
Time-Series DBs: InfluxDB

https://docs.influxdata.com/influxdb/v1.7/concepts/key_concepts/
https://bluefox.io/influxdb-good-bad-ugly

Highlights:

- time-slot aggregates: hourly and daily aggregations using different aggregate functions
- retention policies: automatically delete old data
- sharded (by time) data storage - using LSM Trees - enables ingestion and processing parallelization

Log Structured Merge Trees:
https://en.wikipedia.org/wiki/Log-structured_merge-tree



Compaction continues creating fewer, larger and larger files

TICK Platform:

T=Telegraf a time-series data collector agent for collecting and reporting metrics and data. It's a binary program that runs across all servers.

I= InfluxDB, the time series database piece.

C = Chronograf is the visualization piece for monitoring and dashboarding. It has pieces for working with components of the stack, including monitoring, and alerting rules.

K = Kapacitor is the real-time streaming data processing engine. It offers basic ETL, anomaly detection, monitoring and alerting. It works for both stream and batch mode monitoring and send alerts to about 20 different systems including Slack and PagerDuty.

@Marina Popova

Time-Series DBs: InfluxDB

There are Tradeoffs for everything .....

https://docs.influxdata.com/influxdb/v1.7/concepts/insights_tradeoffs/

- shard size, based on time, is crucial:
  - too small - slow queries
  - too large - slow ingestion
- data UUID is timestamp only
- distributed deployments (clustered) are only supported in Enterprise-level (paid) versions ...
- schema-free - can be advantage or disadvantage
- many more ...

@Marina Popova

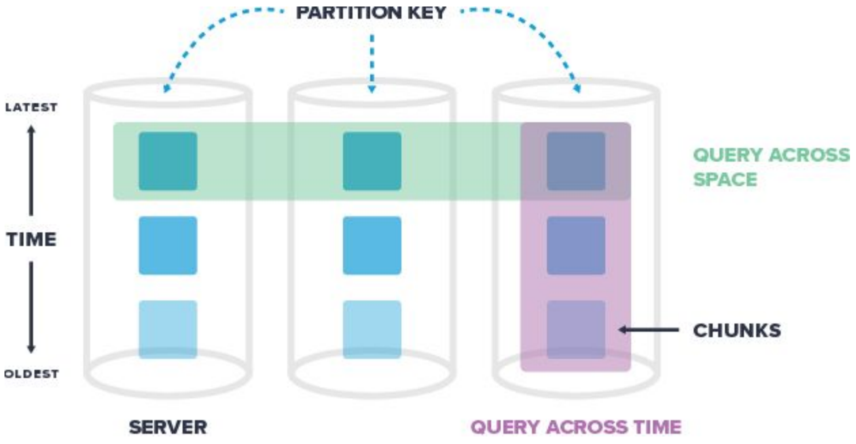
Time-Series DBs: TimescaleDB

https://pdfs.semanticscholar.org/049a/af11fa98525b663da18f39d5dcc5d345eb9a.pdf?_ga=2.202410385.1243494251.1573585903-875000746.1572201200

- Extension to PostgresDB
- full SQL support
- full PostgresDB tools support (backup, CLI, GUI)
- all other Time-series specific functionality like time-bucket aggregations, retention, etc.

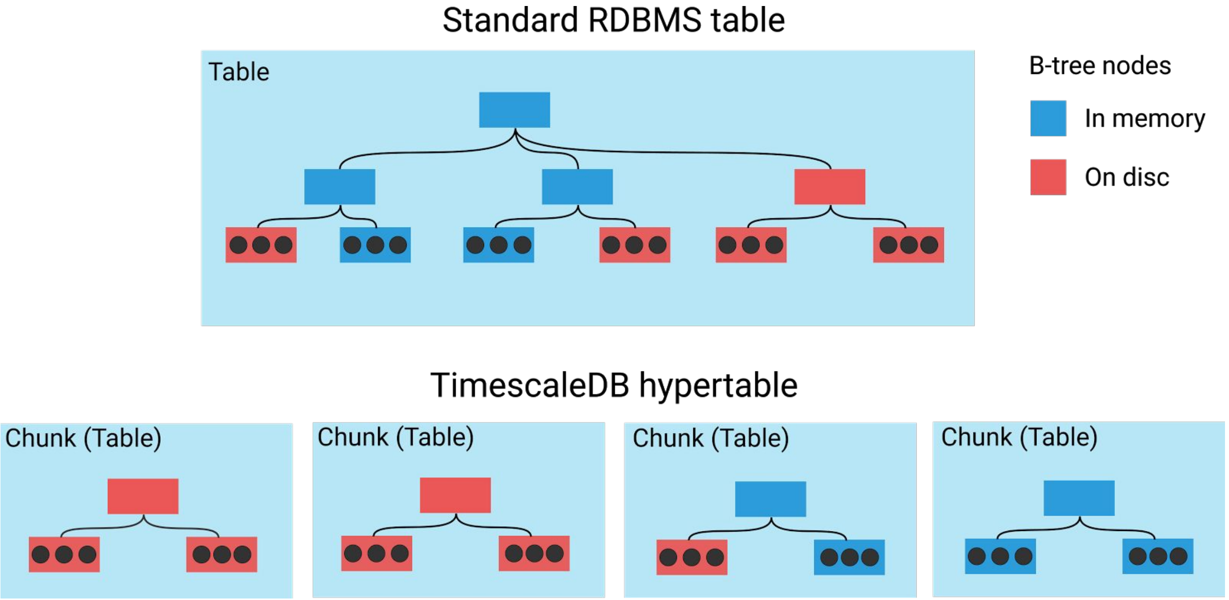
Data Model: adaptive time/space chunking

- Hypertable: a virtual view over multiple distributed tables called "chunks"
- Chunks: physical tables distributed over multiple nodes by time and "space" (other partitioning keys)
- size of "chunks" is adaptable: it changes based on ingestion rate/amount of data



Time-Series DBs: TimescaleDB

https://blog.timescale.com/blog/time-series-data-why-and-how-to-use-a-relational-database-instead-of-nosql-d0cd6975e87c/



"TimescaleDB stores each chunk in an internal database table, so indexes only grow with the size of each chunk, not the entire hypertable. As inserts are largely to the more recent interval, that one remains in memory, avoiding expensive swaps to disk"

@Marina Popova

Time-Series DBs: TimescaleDB

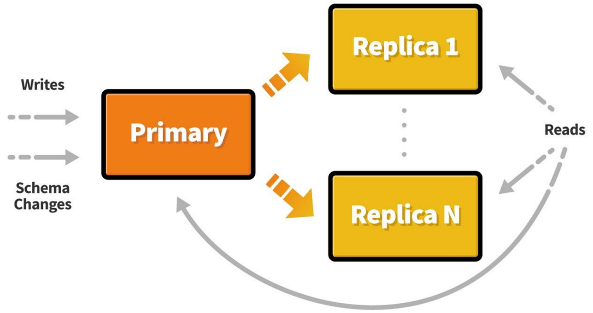
Horizontal Scalability and High Availability
.... is a challenge!

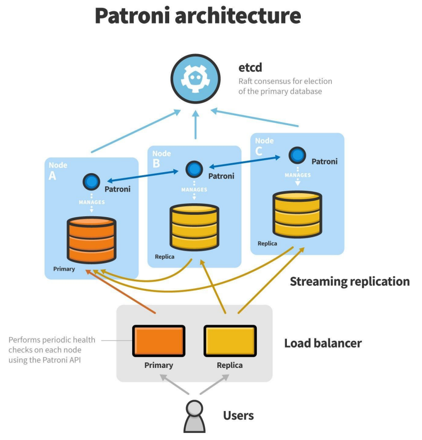
- could use PostgresSQL Streaming Replication
    - ok for < 50K/sec streams
    - If the primary node fails or becomes unreachable, there needs to be a method to promote one of the read replicas to become the new primary
    - there is no automated way to do that in Postgres out-of-the-box

- or other 3-rd party tools like Patroni

https://blog.timescale.com/blog/high-availability-timescaledb-postgresql-patroni-a4572264a831/

PG Streaming Replication

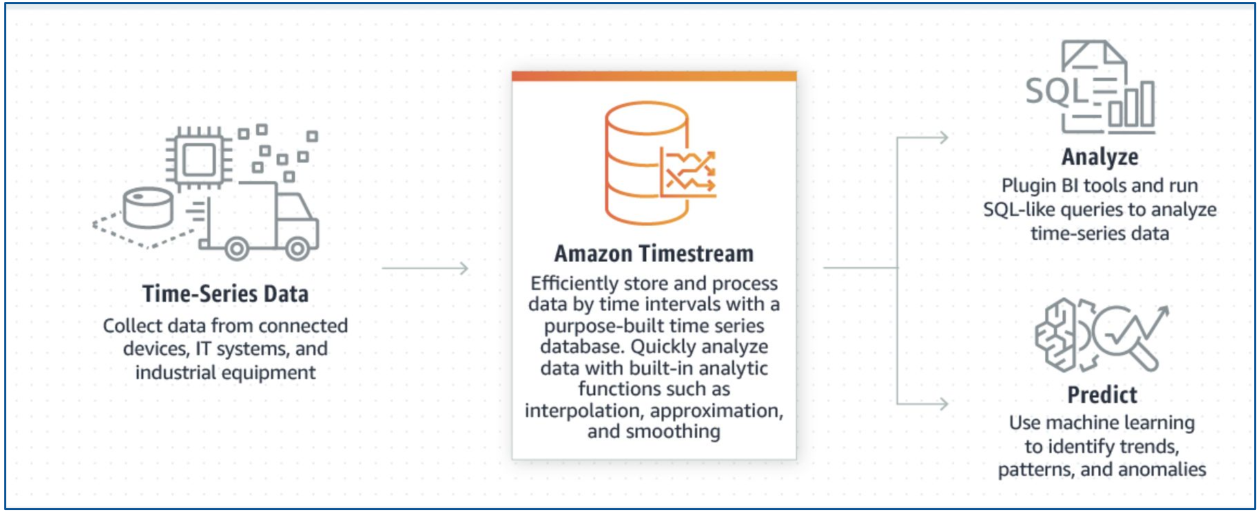




@Marina Popova

Time-Series DBs: Others ...

Amazon Timestream (announced in 2018): available for Preview only as of now



Time-Series Data

Collect data from connected devices, IT systems, and industrial equipment

Amazon Timestream

Efficiently store and process data by time intervals with a purpose-built time series database. Quickly analyze data with built-in analytic functions such as interpolation, approximation, and smoothing

Analyze

Plugin BI tools and run SQL-like queries to analyze time-series data

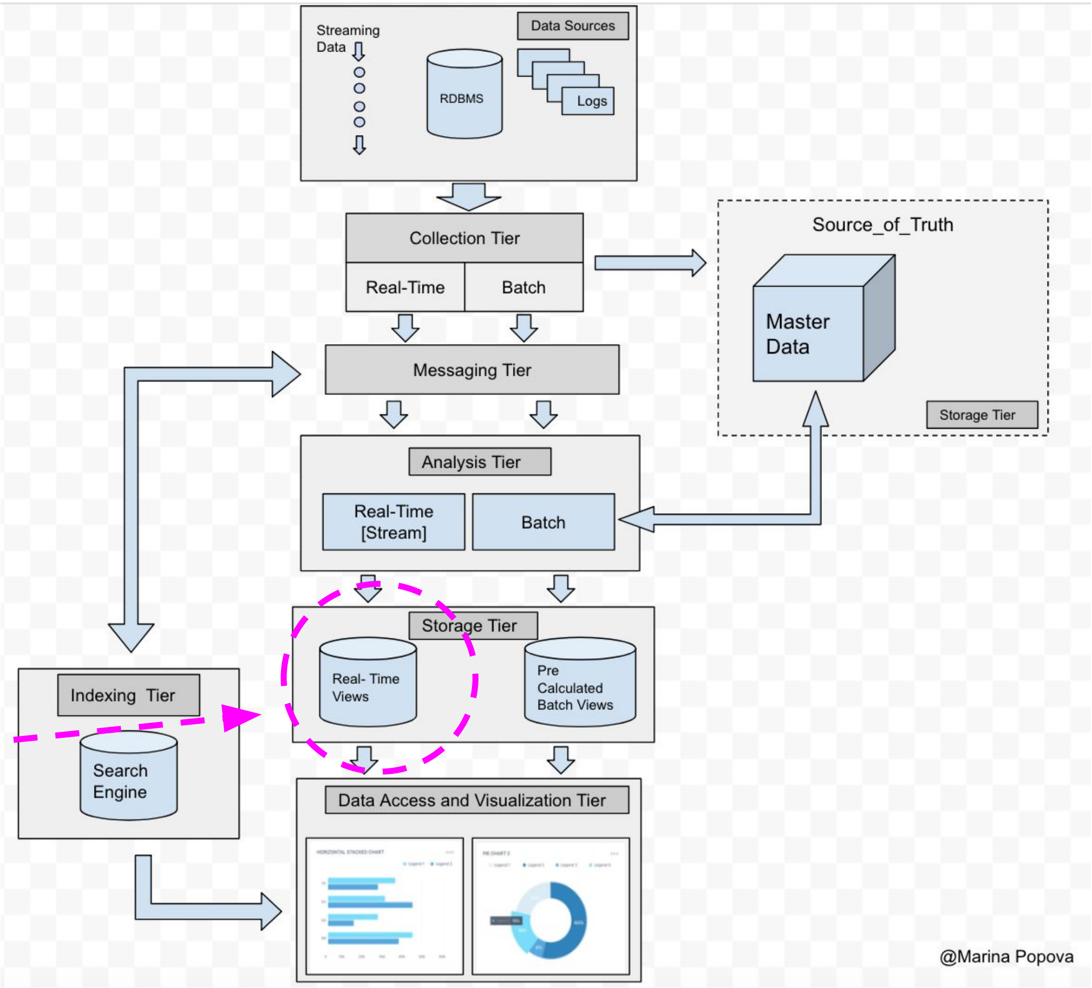
Predict

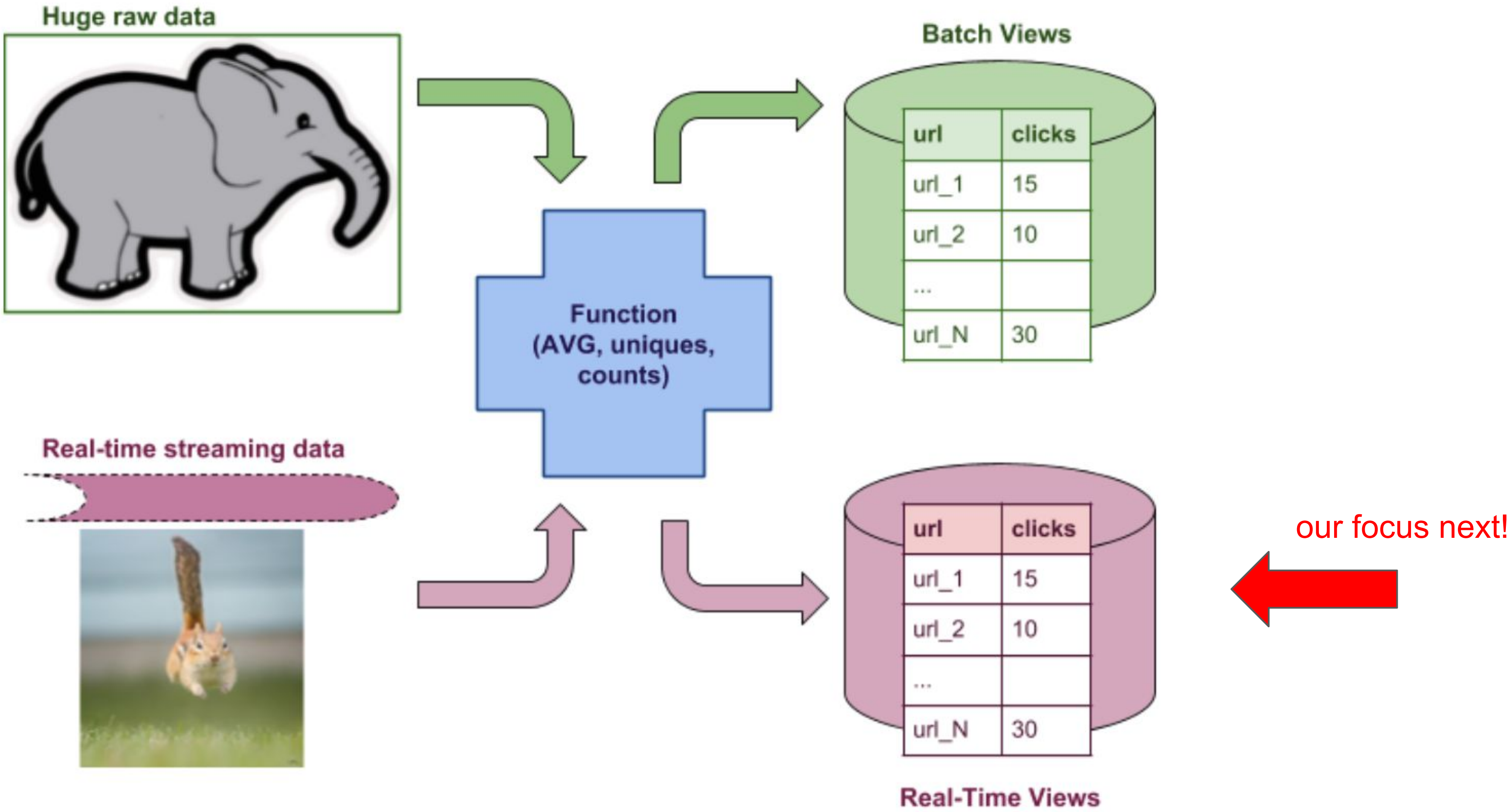
Use machine learning to identify trends, patterns, and anomalies

Prometheus: will discuss later ...

Let's pause ....

Where are we?
And what are we focusing on next?





Huge raw data

Real-time streaming data

Function (AVG, uniques, counts)

Batch Views

url	clicks
url_1	15
url_2	10
...	
url_N	30

Real-Time Views

url	clicks
url_1	15
url_2	10
...	
url_N	30

our focus next!

@Marina Popova

Real-Time Views

What are Real-Time Views and why do we need them?

- results of computations in the Streaming Layer
- provide fast (real-time) and efficient (low latency) access to the calculated/collected analytics results

Requirements:

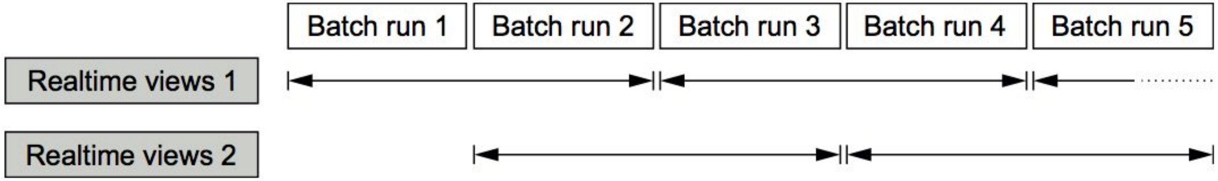
- Random reads - to support fast random reads to answer queries quickly. This means the data it contains must be indexed
- Random writes - to support incremental algorithms - requires low latency updates to the calculated metrics
- Scalability - has to scale with the amount of data and the read/write rates required by the applications; distributed across many machines.
- Fault tolerance—often accomplished by replicating data across machines

You could choose Cassandra to store events in a key/value format, and then use ElasticSearch for indexes that support search queries

Real-Time Views

In the context of Lambda Architecture, you need to worry about **expiring your RT views**, when more correct results are computed by the batch layer
What are the approaches for that?

- TTL/ time-delayed expirations on key-values - not ideal
- maintain two sets of realtime views and alternate clearing them after each batch layer run



@Marina Popova

Cassandra

Apache Cassandra was developed at Facebook

Best source of information:
Apache Cassandra website: http://cassandra.apache.org/doc/latest/

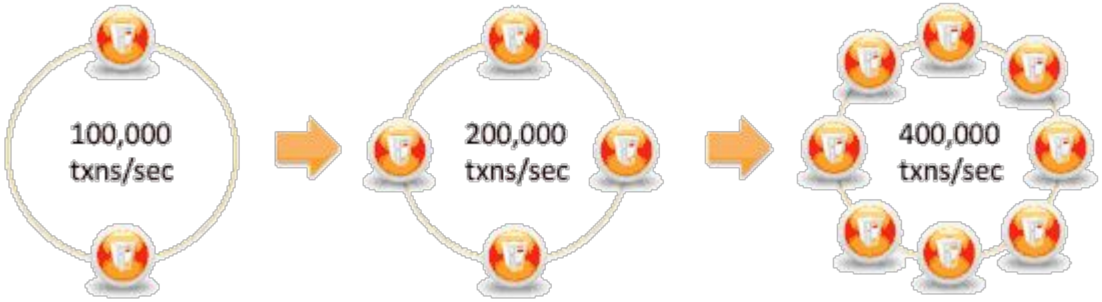
DataStax website:
https://www.datastax.com/dev/blog/basic-rules-of-cassandra-data-modeling
https://docs.datastax.com/en/cql/3.1/cql/ddl/dataModelingApproach.html

Online book "Learn Cassandra" : https://teddyma.gitbooks.io/learncassandra/content/index.html
Covers Cassandra 2.x - but not 3.x

Other:
https://www.jorgeacetozi.com/single-post/cassandra-architecture-and-write-path-anatomy
Cassandra Distributed Architecture:
https://www.guru99.com/cassandra-architecture.html

Cassandra - cluster architecture

- **AP system**: availability and partition tolerance are generally considered to be more important than consistency.
- can be tuned with replication factor and consistency level to also meet C
- supplies linear scalability: capacity may be increased by adding new nodes:
    - For example, if 2 nodes can handle 100,000 transactions per second, 4 nodes will support 200,000 transactions/sec and 8 nodes will tackle 400,000 transactions/sec:

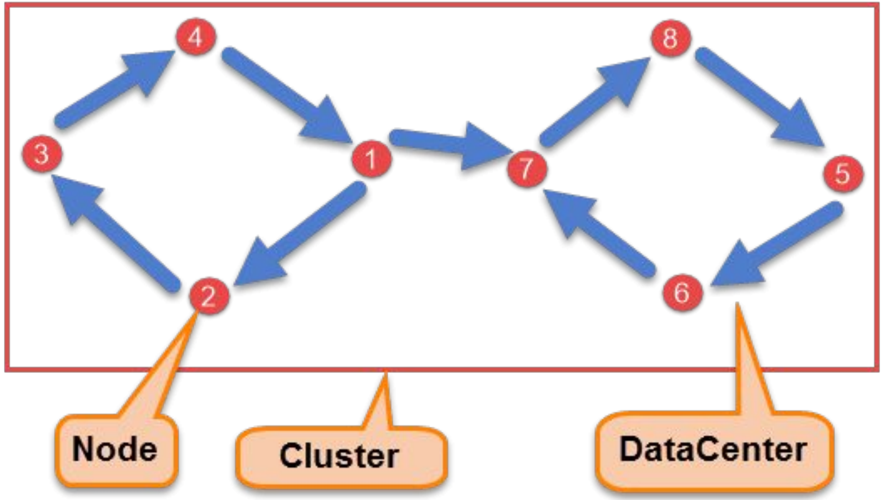


Cassandra - cluster architecture

Main Components of the Cassandra Architecture:

The common topology for a Cassandra installation is a set of instances installed on different server nodes, forming a cluster of nodes also referenced as the **Cassandra ring**.

- a **node** is a cassandra instance (in production: one node per machine)
- a **partition** is one ordered and replicable unit of data on a node
- a **rack** is a logical set of nodes
- a **Data Center** is a logical set or racks
- a **cluster**: full set of nodes



Cassandra - cluster architecture

Cluster Coordination

- The architecture of Cassandra is “masterless”, meaning all nodes are the same.
- Each node can act as a master, known as **coordinator** node.
- A coordinator is the node chosen by the client to receive a particular read or write request to its cluster. It will dispatch the read/write request to specific node[s] and receive an ack
- Each client request may be coordinated by a different node
- **No single point of failure !**

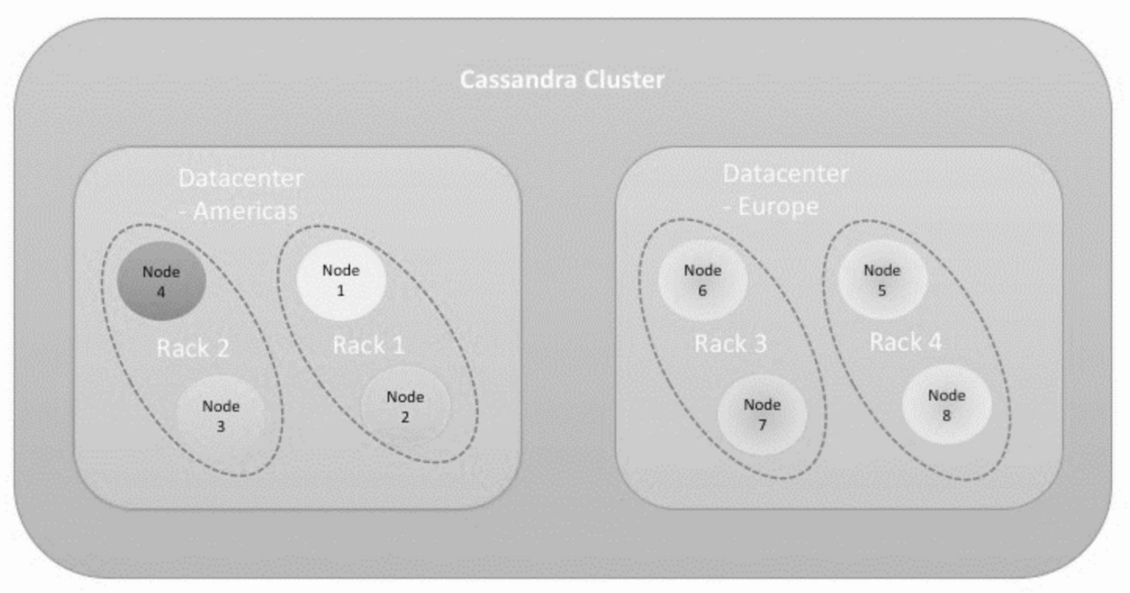
When client makes a request (read or write):

- Cassandra **Driver** will choose a node to act as a coordinator
- By default: Round-Robin pattern

Cassandra - cluster architecture

Datacenter deployment support:

- can specify which nodes will be located in the same datacenter and even the rack position
- Goals:
    - increase the level of high-availability
    - reduce the read latency, so that clients can read data from the nearest node.
- Done via a specific configuration called “Network Topology Strategy” defined on keyspace definition

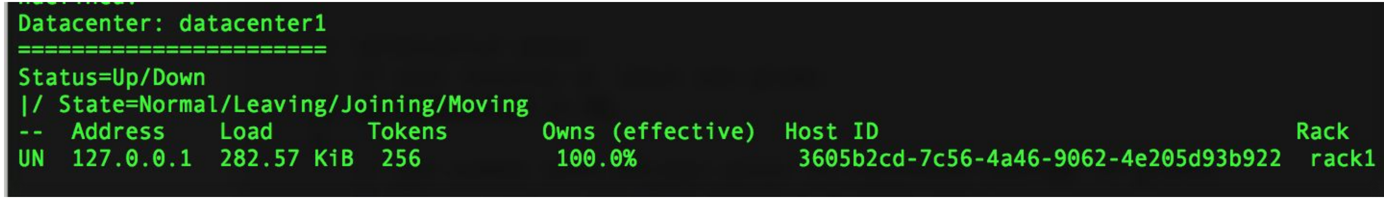


@Marina Popova

Cassandra - cluster architecture

Main configuration file - on each cluster node:
conf/cassandra.yaml

./bin/nodetool status
-- show status of the cluster



```
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load       Tokens       Owns (effective)  Host ID                               Rack
UN  127.0.0.1  282.57 KiB  256          100.0%            3605b2cd-7c56-4a46-9062-4e205d93b922  rack1
```

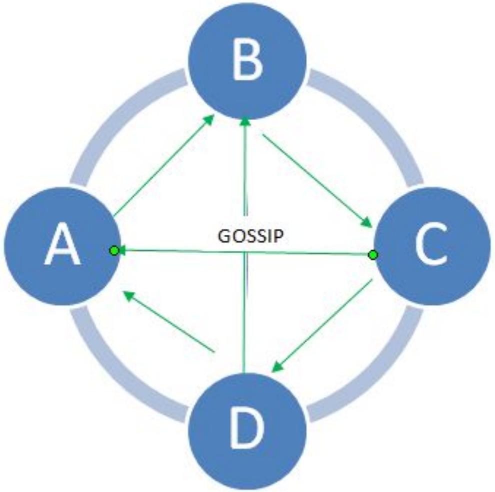
How do nodes recognize their cluster?
-- based on settings in conf/cassandra.yaml :

- **seed:** IP addresses of initial nodes for a new node to contact and discover the cluster topology
- **cluster_name**: shared name to logically distinguish a set of nodes
- **listen_address**: IP address through which this particular node communicates

Cassandra - node communication

Node communication is done through a peer-to-peer communication protocol called **Gossip Protocol**

- broadcasts information about data and nodes health
- One node talks to another node, and passes **info about itself and about other nodes it heard about**
- Used to detect node failures - by **requiring an ACK** from the node it communicates with
- Helps the client to choose a better available Node to connect to in order to load balance connections and find the nearest and fastest path to read required data
- A gossip message has a version associated with it, so that during a gossip exchange, older information is overwritten with the most current state for a particular node.



PEER TO PEER DISTRIBUTION
MODEL OF CASSANDRA

@Marina Popova

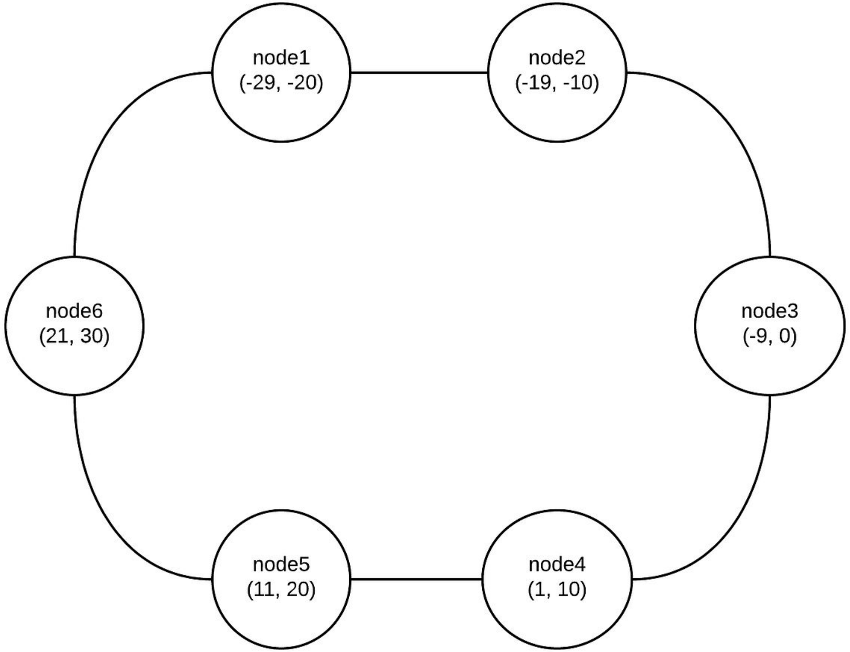
Cassandra - Data Distribution

Cassandra provides automatic data distribution across all nodes of the cluster. How?

Consistent Hashing !!

This work is done by a **Partitioner:**

- determines how data is distributed across the cluster
- each node is assigned a range of tokens
  - OR virtual nodes
- partitioner: to identify which node a request for read/write of a data row should go:
  - **applies a hash function to a partition key to compute the token**
  - each row of data is uniquely identified by a **partition key** and distributed across the cluster by the value of the token
  - example: (next slide)

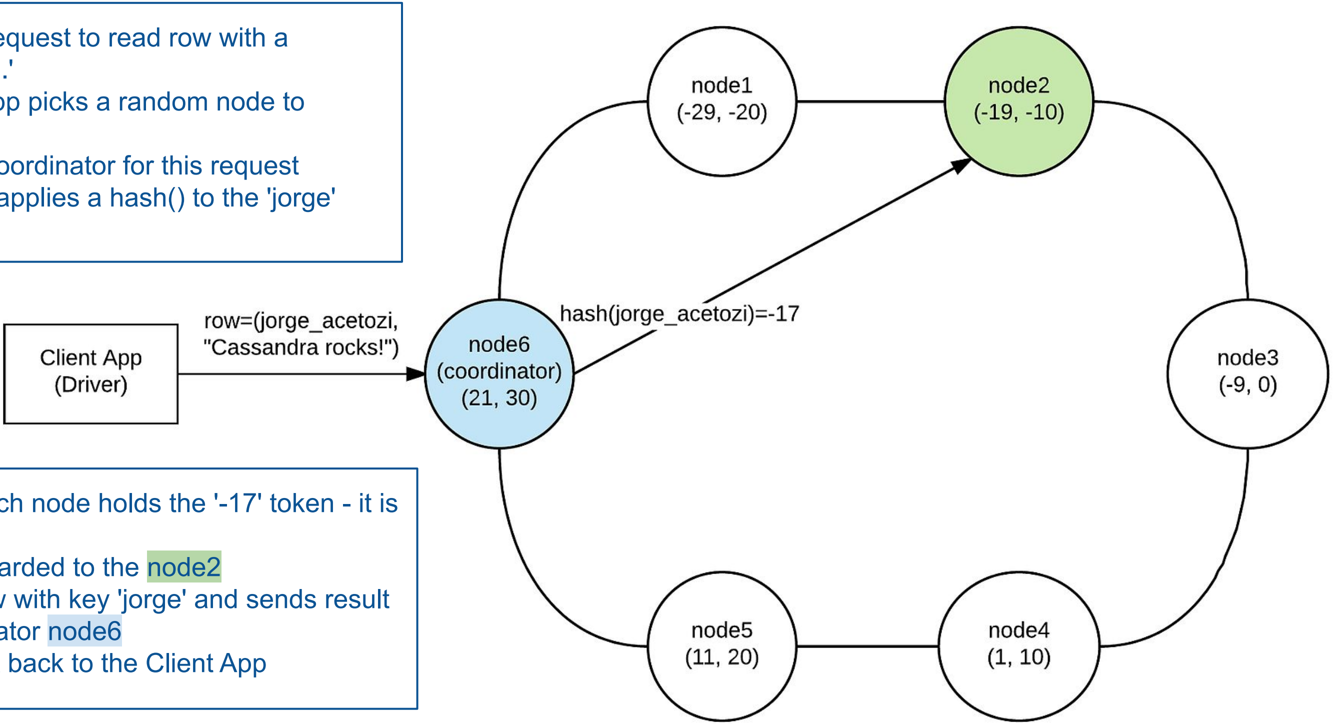


@Marina Popova

Cassandra - Data Distribution

Example of Partitioning: (from https://www.jorgeacetozi.com/single-post/cassandra-architecture-and-write-path-anatomy)

- Client App issues a request to read row with a partition key = 'jorge...'
- Driver in the Client App picks a random node to handle it - it is node6
- node6 became the Coordinator for this request
- Partitioner on node6 applies a hash() to the 'jorge' key - gets value -17



- Partitioner finds which node holds the '-17' token - it is node2
- read request is forwarded to the node2
- node2 reads the row with key 'jorge' and sends result back to the Coordinator node6
- node6 returns result back to the Client App

Cassandra - Data Distribution

Cassandra provides the following out-of-the-box partitioners:

- Murmur3Partitioner (default): uniformly distributes data across the cluster based on MurmurHash hash values.
- RandomPartitioner: uniformly distributes data across the cluster based on MD5 hash values.
- ByteOrderedPartitioner (legacy): keeps an ordered distribution of data lexically by key bytes

In Summary:

- **Token**: a 128 bit integer ID used to identify each partition and node
- **Partition**: a unit of data storage on a node (analogous to a set of 'rows')
- **Partition key**: identifier of a row of data - used to calculate a matching token to identify a specific partition
- **Partitioner**: a system *on each node* hashing **primary (partition) key** values (of a 'row') into a token to be used as a partition key
- The 2^127-1 value token range for a cluster is treated as a **ring**

Cassandra - Data Replication

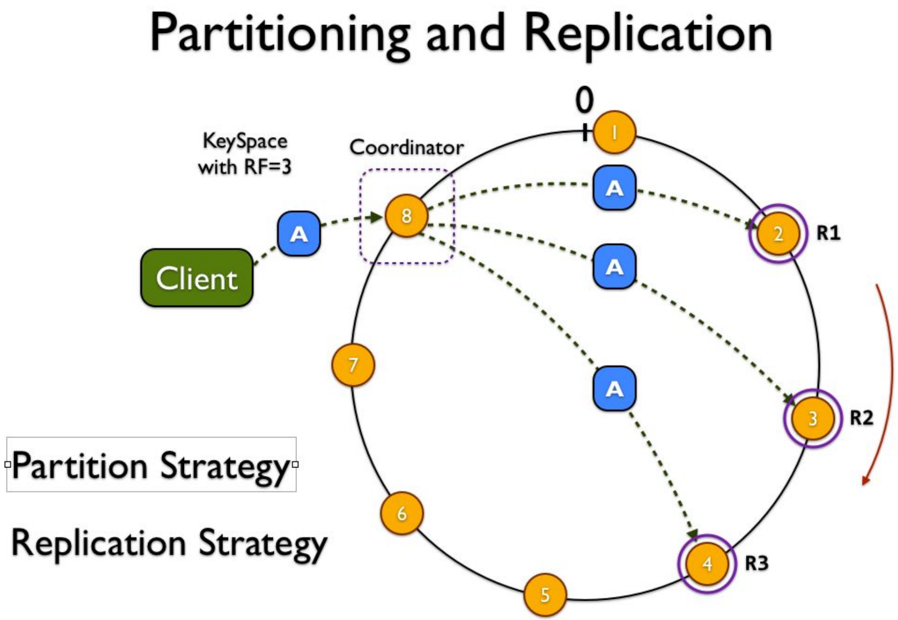
Cassandra provides built-in and **customizable replication**, which stores redundant copies of data across nodes in a Cassandra ring

Cassandra places replicas of data on different nodes based on the following two factors:

- Where to place next replica is determined by the **Replication Strategy**
- the total number of replicas placed on different nodes is determined by the **Replication Factor**

Replication Strategies:

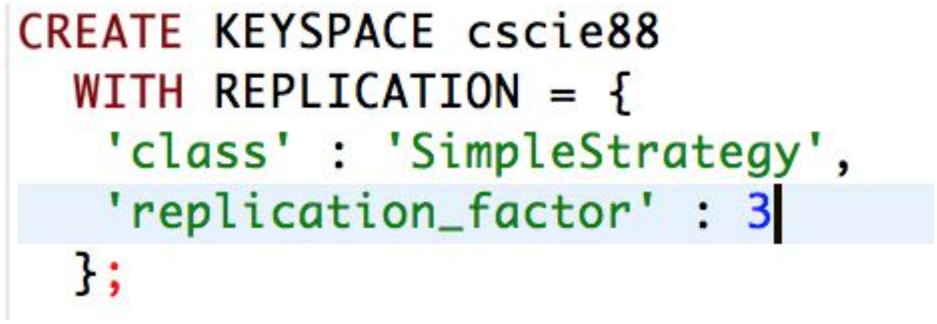
- SimpleStrategy: used for a single data center deployments
- NetworkTopologyStrategy: used for multiple data centers deployments



@Marina Popova

Cassandra - Data Replication

These settings are done when "keyspace" is first created:



```
CREATE KEYSPACE cscie88
  WITH REPLICATION = {
   'class' : 'SimpleStrategy',
   'replication_factor' : 3
  };
```

Cassandra - Consistency Levels

How Replication Strategy works:
it takes the partitioner's decision (the node that will handle the request first based on the token range) and places the remaining replicas clockwise in relation to this node

Partition *key*, *replication factor* and *replication strategy* determine which node a specific request is sent to

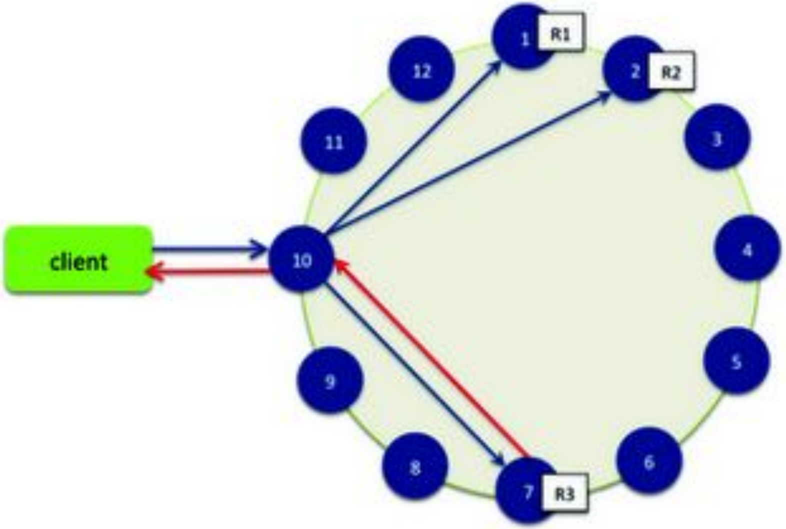
Consistency Level determines how many nodes must acknowledge the request before response is returned to the client/driver

- CL may vary for each request: **tunable consistency!**
- options:
  - ANY: write to any node (highest availability, lowest consistency)
  - ONE: check closest replica node (highest availability, lowest consistency)
  - QUORUM (RF/2+1) (balanced consistency and availability)
  - ALL: check all replica, fail if any is down (lowest availability, highest consistency)

Cassandra - Consistency Levels

Meaning is slightly different based on the request type:

- **write request**: how many nodes must acknowledge they received and wrote the partition?
- **read request**: how many replica nodes must acknowledge and send their most recent partition data? (read results merge to most current by coordinator



Cassandra - Consistency Levels

QUORUM consistency level

At the QUORUM level : writes/reads have to happen on the number of nodes that make up a quorum. A quorum is calculated, and then rounded down to a whole number, as following:

Q = (sum_of_replication_factors / 2) + 1

- sum_of_replication_factors = sum of all the replication_factor settings for each data center
- this determines availability - how many nodes down can be tolerated

Exercise:
(one DC)

Replication Factor	Quorum	How many nodes down can be tolerated ?
3	2	1
6	4	2

@Marina Popova

Cassandra - Consistency Levels

QUORUM consistency level

If consistency is top priority, you can ensure that a read always reflects the most recent write by using the following formula:

(nodes_written + nodes_read) > replication_factor

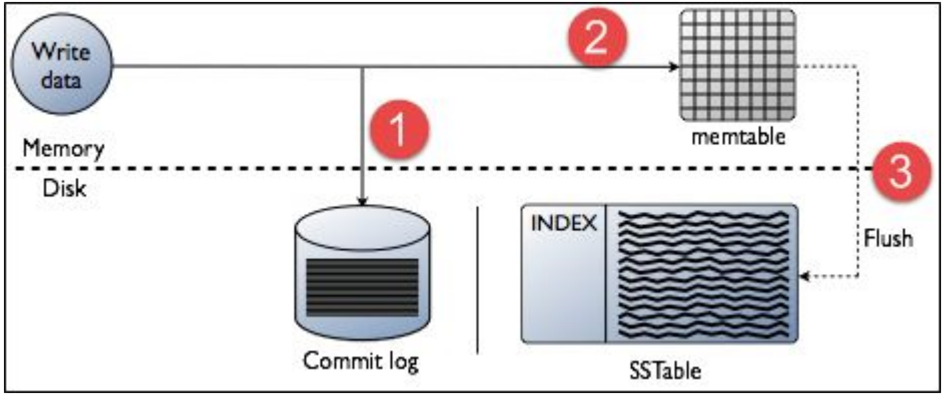
Example:

- using QUORUM for both write and read operations
- replication factor of 3 → Q = 2
- this means: 2 nodes are always written and 2 nodes are always read
- the combination of nodes written and read (2+2 = 4) > 3 (RF) → ensures **strong read consistency**

Cassandra - Write Operations Details

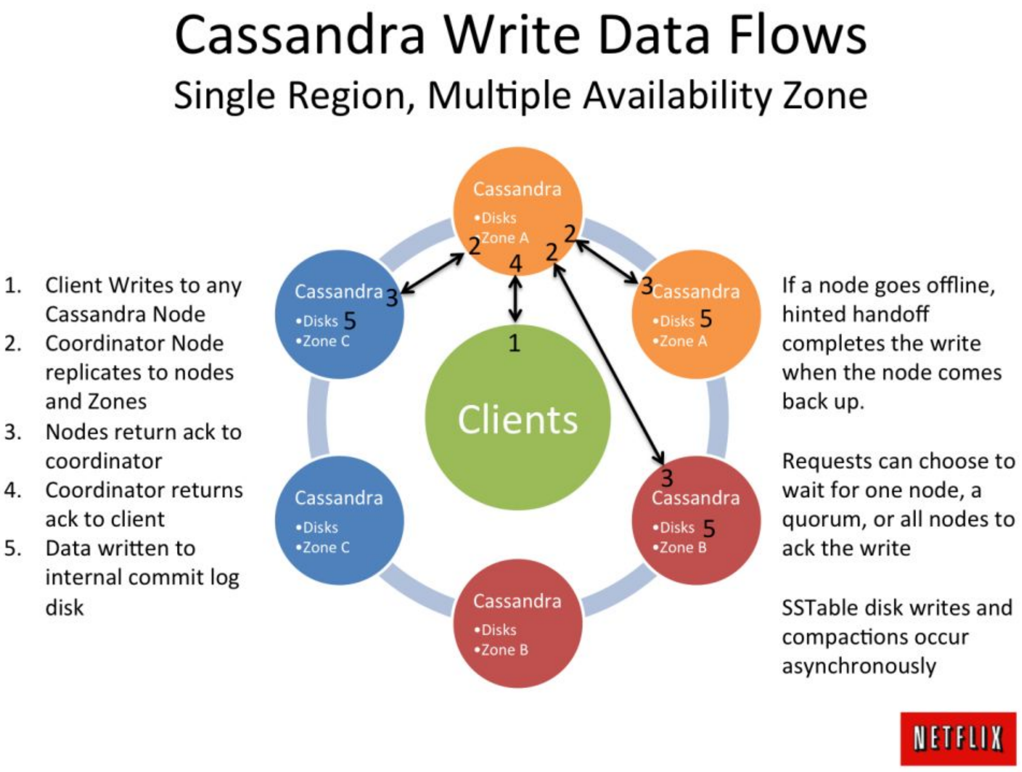
write process steps:

1. when write request comes to the node - it is logged in the **commit log**
   a. commit log keeps transaction records for backup purposes
2. then Cassandra writes the data into **memTable**
   a. memTable is a tmp in-memory storage
3. when memTable is full (or explicit flush called), data is flushed to the **SSTable data file**



Cassandra - Write Operations

- The coordinator sends a write request to all replicas. If all the replicas are up, they will receive write request regardless of the consistency level
- **Consistency level** determines how many nodes must respond back with the **success** acknowledgment.
- The node will respond back with the success acknowledgment if the data is written successfully to **the commit log and memTable**



Cassandra Write Data Flows

Single Region, Multiple Availability Zone

1. Client Writes to any Cassandra Node
2. Coordinator Node replicates to nodes and Zones
3. Nodes return ack to coordinator
4. Coordinator returns ack to client
5. Data written to internal commit log disk

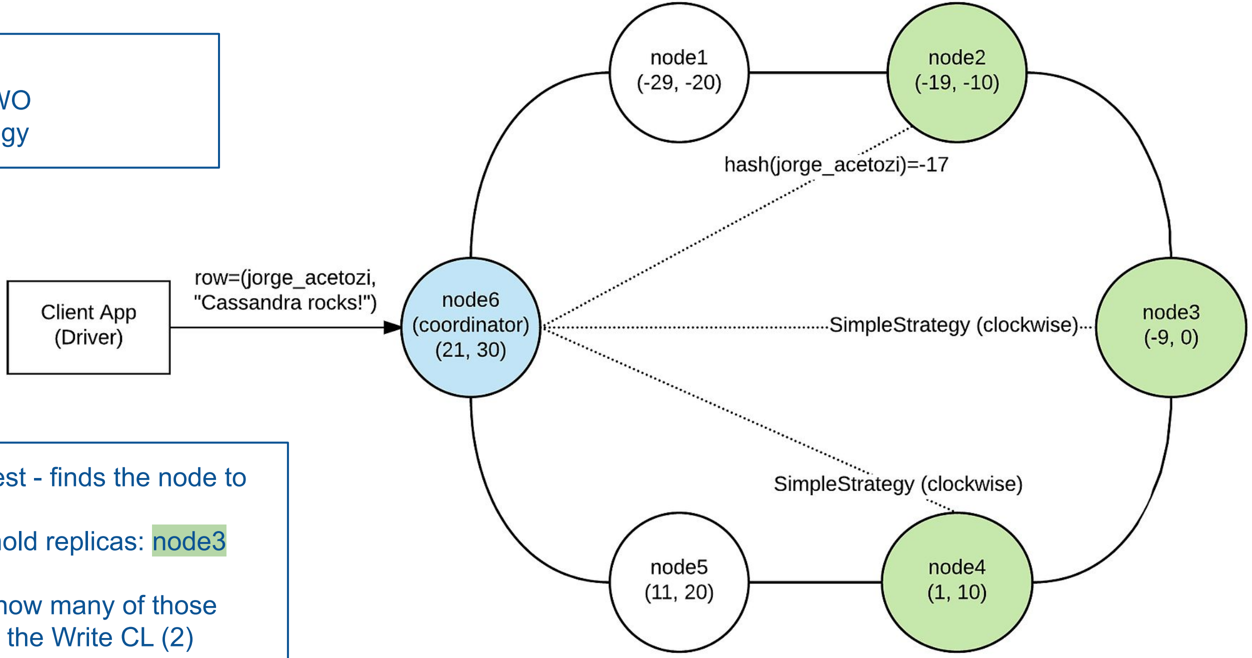
If a node goes offline, hinted handoff completes the write when the node comes back up.

Requests can choose to wait for one node, a quorum, or all nodes to ack the write

SSTable disk writes and compactions occur asynchronously

Cassandra - Write Requests - example

- REPLICATION FACTOR = 3
- WRITE CONSISTENCY LEVEL = TWO
- Replication Strategy = SimpleSgtrategy



- node6 is the Coordinator for this request - finds the node to store the row - node2
- also determines which nodes should hold replicas: node3 and node4
- asks the Failure Detector component how many of those nodes are available and compare with the Write CL (2)
- if >= than 2 nodes are available - they will all write the row and Coordinator returns SUCCESS to the Driver
- if LESS than 2 nodes are available - Coordinater returns ERROR to the Driver

@Marina Popova

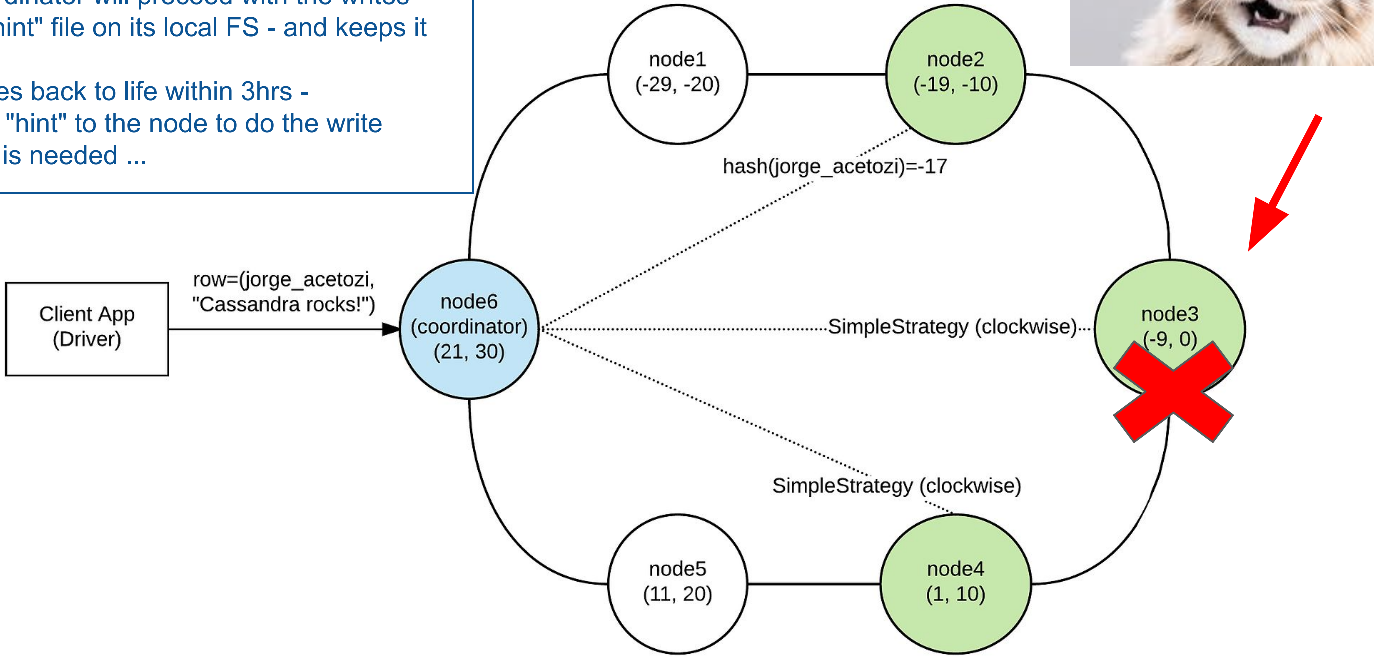
Cassandra - Write Requests - example

what if MORE OR EQUAL to the Write Consistency Level (2) nodes are available, but NOT ALL NODES required by the RF (3) ??

a **Hinted Handoff** is initiated ...

- since the number of available nodes (2) is NOT LESS than the Write CL (2) - Coordinator will proceed with the writes
- Coordinator writes a "hint" file on its local FS - and keeps it for 3 hrs (default)
- if the BAD node3 comes back to life within 3hrs - Coordinator sends the "hint" to the node to do the write
- if not - a **Read Repair** is needed ...

What if node3 is "down" ??



Client App (Driver)
row=(jorge_acetozi, "Cassandra rocks!")
node6 (coordinator) (21, 30)
node1 (-29, -20)
node2 (-19, -10)
hash(jorge_acetozi)=-17
SimpleStrategy (clockwise)
node3 (-9, 0)
SimpleStrategy (clockwise)
node5 (11, 20)
node4 (1, 10)

@Marina Popova

Cassandra - Read Operations

There are three types of read requests that a coordinator sends to replicas:

1. Direct request
2. Digest request
3. Read repair request

- The coordinator sends **direct request** to one of the replicas
- After that, the coordinator sends the **digest request** to the number of replicas specified by the consistency level:
    - to check whether the returned data is an updated data
- After that, the coordinator sends digest request to all the remaining replicas:
    - If any node gives out of date value, a background read repair request will update that data.

This process is called **read repair mechanism**

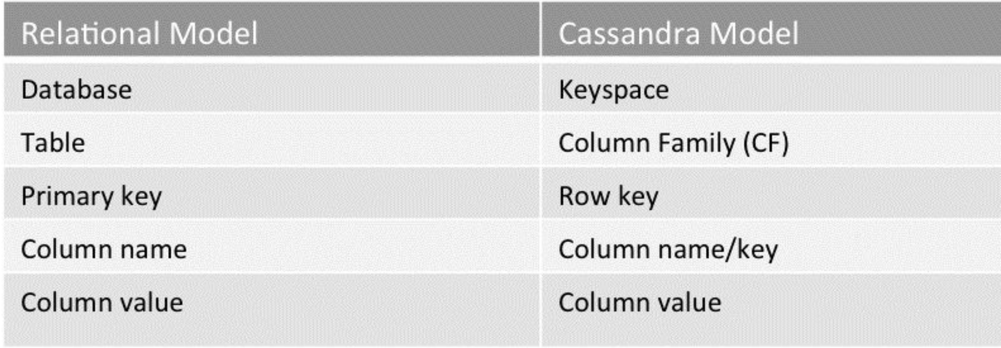
Cassandra Data Model

Cassandra is classified as a **column based database (wide column store):**

Basic concepts:

- basic data structure: set of **rows and columns**
- columns consist of a pair of **column key** and **column value**
- every row is identified by a unique key, a string without a size limit, called **partition key**
- Each set of columns are called **column families** (old style) or **tables** (new style)
- Related tables (CFs) live in a **keyspace**

The following relational model analogy is often used to introduce Cassandra to Newcomers:



Relational Model	Cassandra Model
Database	Keyspace
Table	Column Family (CF)
Primary key	Row key
Column name	Column name/key
Column value	Column value

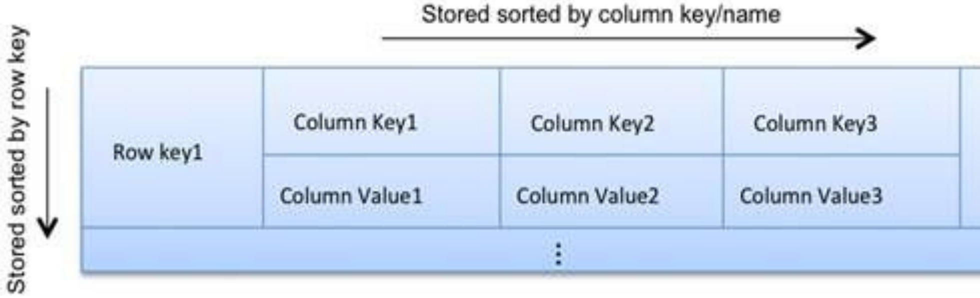
Cassandra Data Model

- Column Family/ Table should be thought of as a **nested sorted map data structure**
- partition (row) keys are the top-level entries in that map
- Cassandra uses the partition keys to partition a column family across a cluster
- all columns for a key are physically stored together, making it inexpensive to access ranges of columns
- different keys/rows can have different sets of columns, and it’s possible to have thousands—or even millions—of columns for a given key
- Cassandra data model is a schema-optional - you do not have to model all the columns up-front

Simplified view:

SortedMap<RowKey,

SortedMap<ColumnKey, ColumnValue>>



Stored sorted by column key/name

Stored sorted by row key

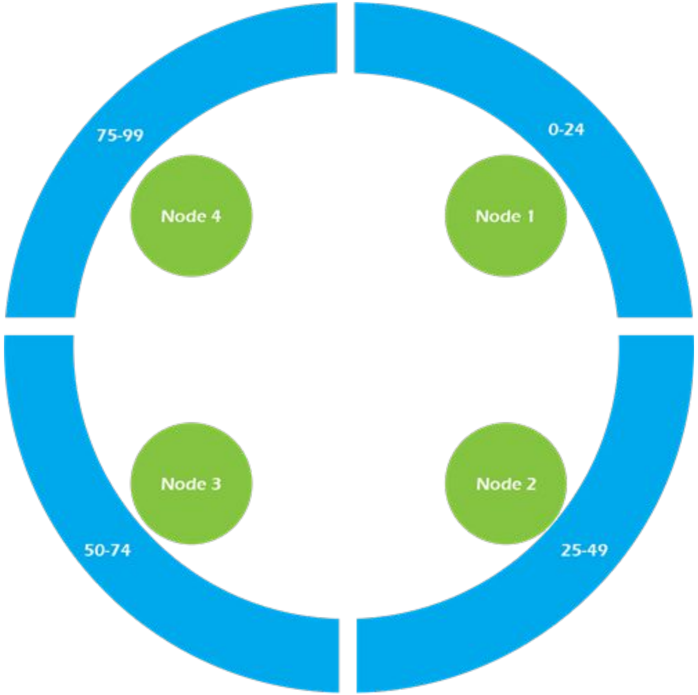
Row key1	Column Key1	Column Key2	Column Key3
	Column Value1	Column Value2	Column Value3
	⋮		

Cassandra Data Model - Keys

Partition key

- The partition key is responsible for **distributing data among nodes** and for **data locality**
- A partition key is the same as the primary key when the **primary key consists of a single column**.
- Partition keys belong to a node
- In Cassandra cluster, each node should have an equal part of the partition key hashes
- partition key will always belong to one node and that partition's data will always be found on that node

Why is that important? If there wasn't an absolute location of a partition's data, then it would require **searching every node** in the cluster for your data



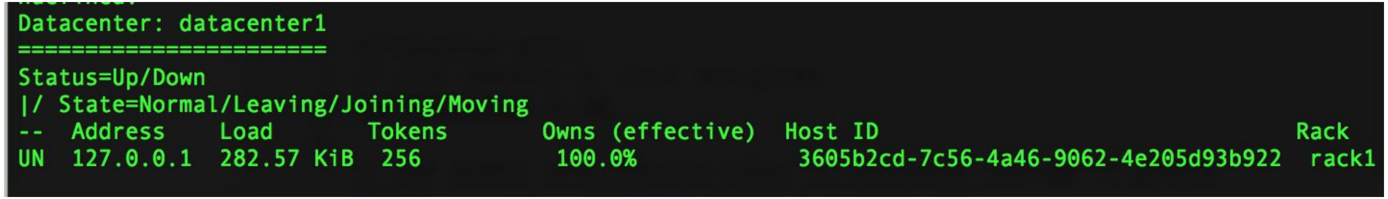
@Marina Popova

Cassandra - tools

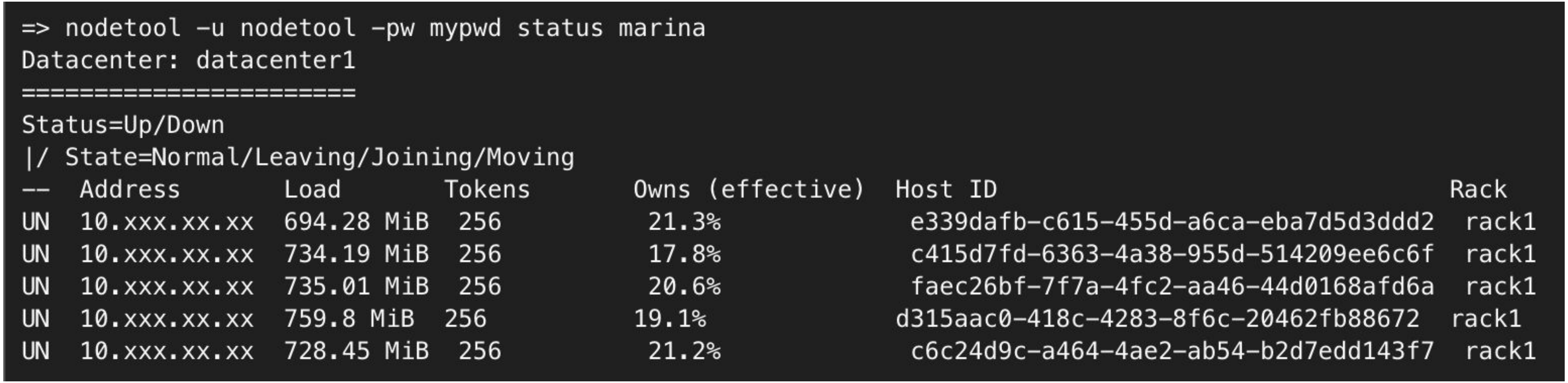
./bin/nodetool status - show status of the cluster

Local vs clustered view:

token ownership info



```
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load       Tokens       Owns (effective)  Host ID                               Rack
UN  127.0.0.1  282.57 KiB  256          100.0%            3605b2cd-7c56-4a46-9062-4e205d93b922  rack1
```



```
=> nodetool -u nodetool -pw mypwd status marina
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address         Load        Tokens       Owns (effective)  Host ID                               Rack
UN  10.xxx.xx.xx  694.28 MiB  256          21.3%             e339dafb-c615-455d-a6ca-eba7d5d3ddd2  rack1
UN  10.xxx.xx.xx  734.19 MiB  256          17.8%             c415d7fd-6363-4a38-955d-514209ee6c6f  rack1
UN  10.xxx.xx.xx  735.01 MiB  256          20.6%             faec26bf-7f7a-4fc2-aa46-44d0168afd6a  rack1
UN  10.xxx.xx.xx  759.8 MiB  256          19.1%             d315aac0-418c-4283-8f6c-20462fb88672  rack1
UN  10.xxx.xx.xx  728.45 MiB  256          21.2%             c6c24d9c-a464-4ae2-ab54-b2d7edd143f7  rack1
```

@Marina Popova

Cassandra Data Model - Keys

How do we define the Partition Key??

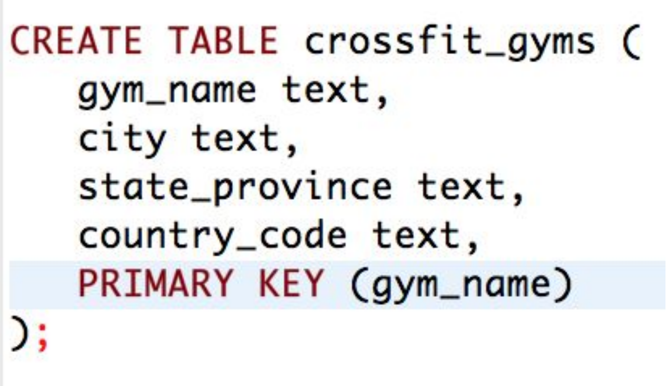
It is the first element in the PRIMARY KEY definition - when creating a table in Cassandra

Example: (Ref: https://shermandigital.com/blog/designing-a-cassandra-data-model/)

CQL: Cassandra Query Language

- primary language for communicating with Cassandra
- Common means of using it is via "cqlsh" - CQL shell
- Or GUI tools like DataStax DevCenter or DBeaver

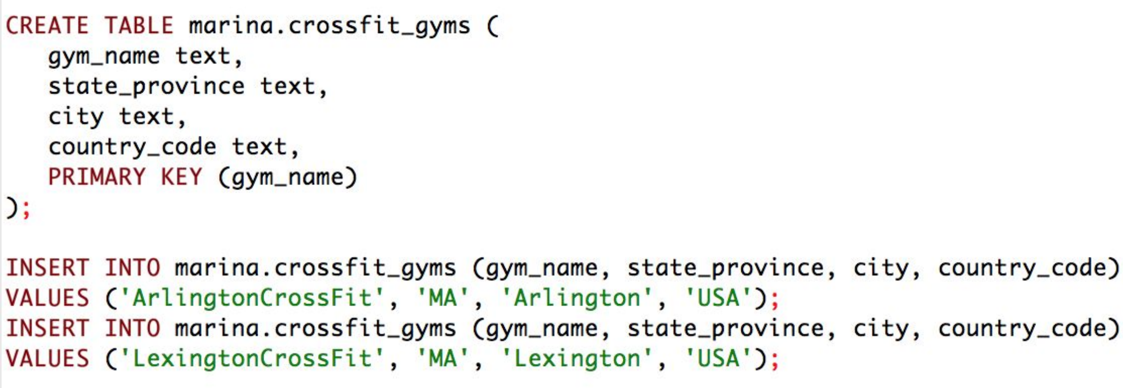
https://docs.datastax.com/en/cql/3.1/cql/cql_intro_c.html



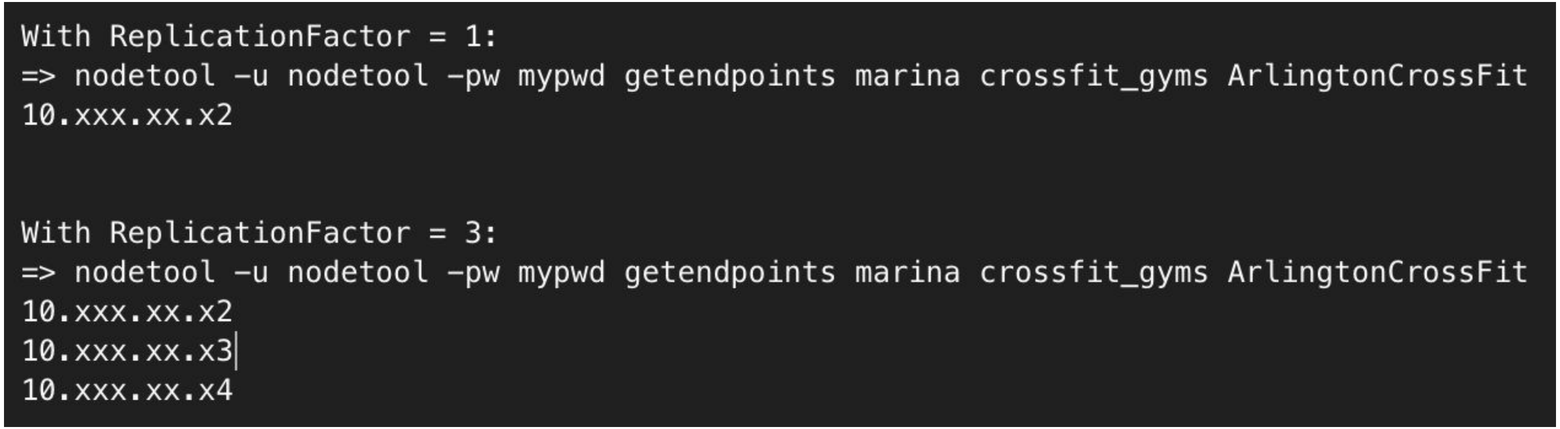
```
CREATE TABLE crossfit_gyms (
   gym_name text,
   city text,
   state_province text,
   country_code text,
   PRIMARY KEY (gym_name)
);
```

Cassandra Tools

./bin/nodetool getendpoints
-- show which nodes own the specified key of your table:



```
CREATE TABLE marina.crossfit_gyms (
   gym_name text,
   state_province text,
   city text,
   country_code text,
   PRIMARY KEY (gym_name)
);

INSERT INTO marina.crossfit_gyms (gym_name, state_province, city, country_code)
VALUES ('ArlingtonCrossFit', 'MA', 'Arlington', 'USA');
INSERT INTO marina.crossfit_gyms (gym_name, state_province, city, country_code)
VALUES ('LexingtonCrossFit', 'MA', 'Lexington', 'USA');
```



```
With ReplicationFactor = 1:
=> nodetool -u nodetool -pw mypwd getendpoints marina crossfit_gyms ArlingtonCrossFit
10.xxx.xx.x2


With ReplicationFactor = 3:
=> nodetool -u nodetool -pw mypwd getendpoints marina crossfit_gyms ArlingtonCrossFit
10.xxx.xx.x2
10.xxx.xx.x3
10.xxx.xx.x4
```

Cassandra Data Model - Keys

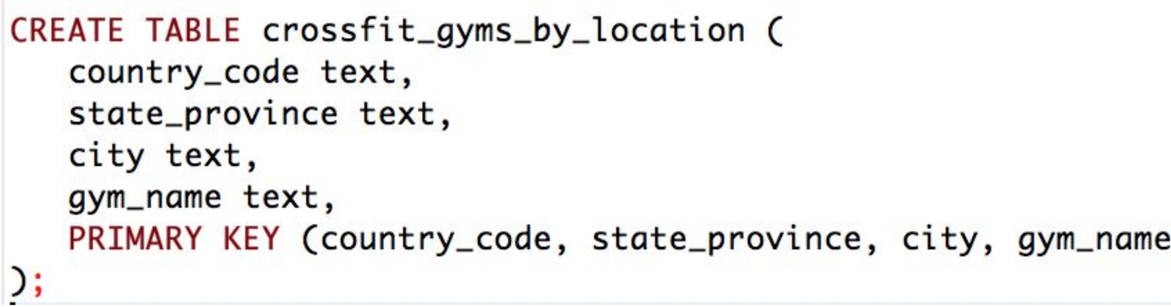
Compound key

Compound keys include **multiple columns in the primary key**, but these additional columns do not necessarily affect the partition key.

- only the first column

is considered the partition key

- the rest of columns are **clustering keys**



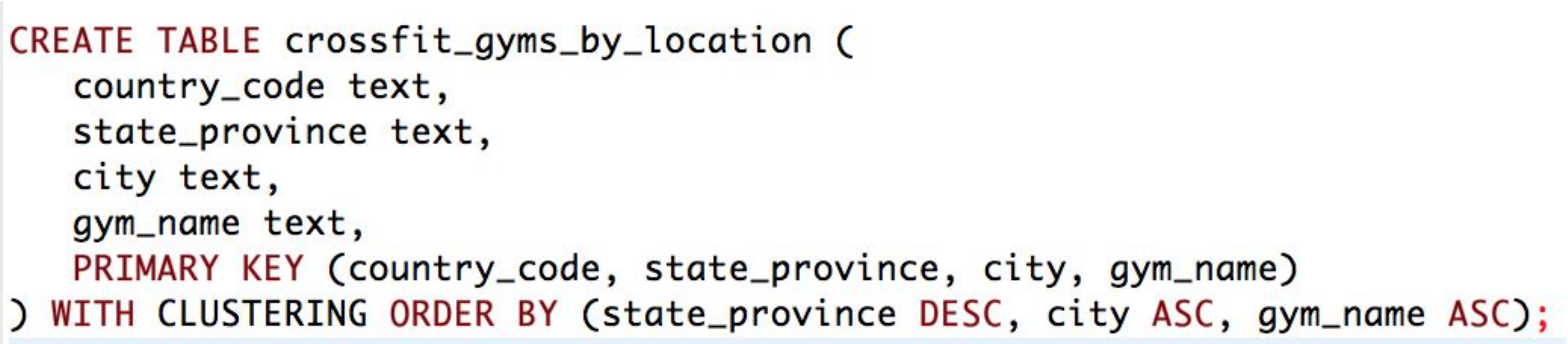
```
CREATE TABLE crossfit_gyms_by_location (
   country_code text,
   state_province text,
   city text,
   gym_name text,
   PRIMARY KEY (country_code, state_province, city, gym_name
);
```

This means that while the primary key represents a unique gym record/row, all gyms within a country reside on the same partition.

Cassandra Data Model - Keys

Clustering key

- **Clustering keys are responsible for sorting data within a partition**
- Each primary key column after the partition key is considered a clustering key.
- Clustering keys are sorted in ascending order by default
- To specify sorting order - use WITH clause:



```
CREATE TABLE crossfit_gyms_by_location (
   country_code text,
   state_province text,
   city text,
   gym_name text,
   PRIMARY KEY (country_code, state_province, city, gym_name)
) WITH CLUSTERING ORDER BY (state_province DESC, city ASC, gym_name ASC);
```

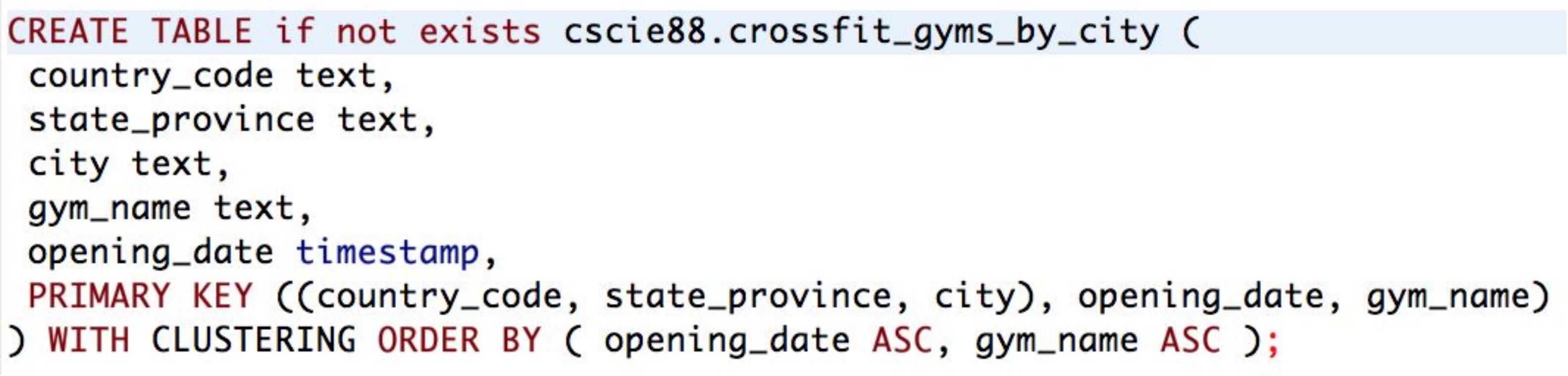
@Marina Popova

Cassandra Data Model

Composite key

- Composite keys are **partition keys that consist of multiple columns**
- They are used to avoid too wide rows

The crossfit_gyms_by_location example only used country_code for partitioning.
The result is that all gyms in the same country reside within a single partition.
To distribute the rows more uniformly - we will add state and city to the partitioning key too:



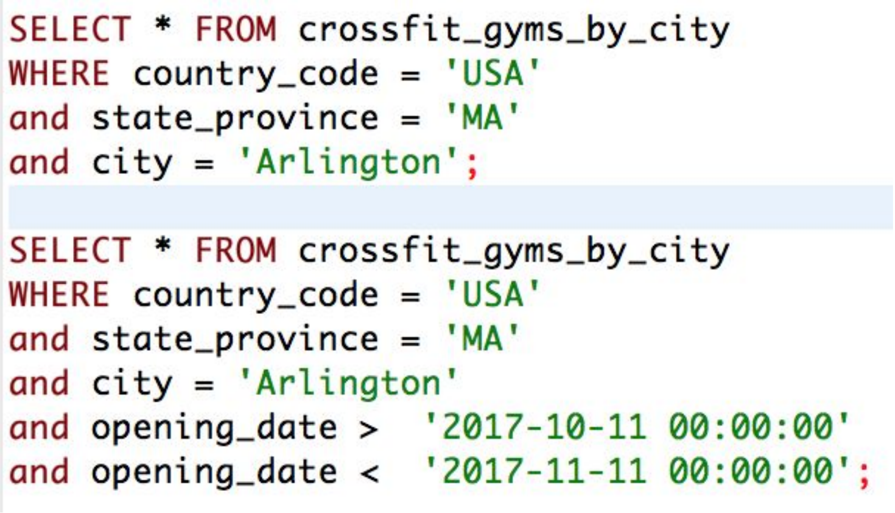
```
CREATE TABLE if not exists cscie88.crossfit_gyms_by_city (
 country_code text,
 state_province text,
 city text,
 gym_name text,
 opening_date timestamp,
 PRIMARY KEY ((country_code, state_province, city), opening_date, gym_name)
) WITH CLUSTERING ORDER BY ( opening_date ASC, gym_name ASC );
```

Cassandra Data Model

How do you query data?

- in CQL query, you must include **all** partition key columns
- you can then apply an additional filter by adding each clustering key in the order in which the clustering keys appear

Example queries:



```
SELECT * FROM crossfit_gyms_by_city
WHERE country_code = 'USA'
and state_province = 'MA'
and city = 'Arlington';

SELECT * FROM crossfit_gyms_by_city
WHERE country_code = 'USA'
and state_province = 'MA'
and city = 'Arlington'
and opening_date >  '2017-10-11 00:00:00'
and opening_date <  '2017-11-11 00:00:00';
```

@Marina Popova

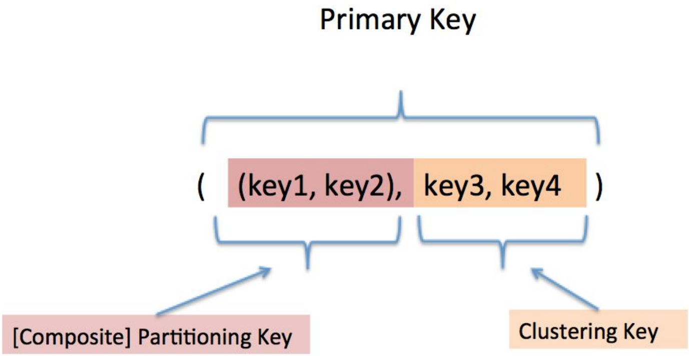
Cassandra Data Model

The reason the order of clustering keys matters is because the clustering keys provide the sort order of the result set. Because of the clustering key’s responsibility for sorting, we know **all data matching the first clustering key will be adjacent!**

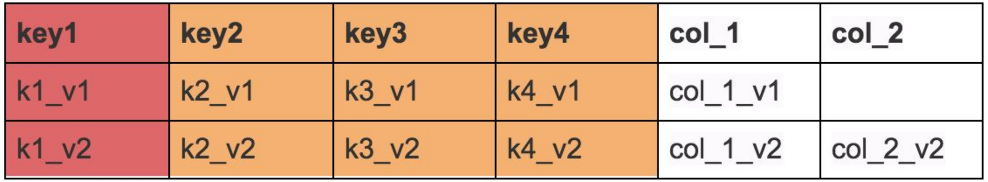
Because we know the order, CQL can easily **truncate sections of the partition** that don’t match our query to satisfy the WHERE conditions pertaining to columns that are not part of the partition key

Cassandra Data Modeling - Keys

Summary for Keys:

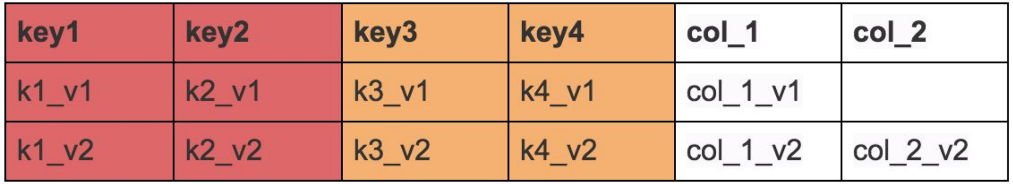


PRIMARY KEY (key1, key2, key3, key4)



key1	key2	key3	key4	col_1	col_2
k1_v1	k2_v1	k3_v1	k4_v1	col_1_v1	
k1_v2	k2_v2	k3_v2	k4_v2	col_1_v2	col_2_v2

PRIMARY KEY ((key1, key2), key3, key4)



key1	key2	key3	key4	col_1	col_2
k1_v1	k2_v1	k3_v1	k4_v1	col_1_v1	
k1_v2	k2_v2	k3_v2	k4_v2	col_1_v2	col_2_v2

@Marina Popova

Cassandra Data Model

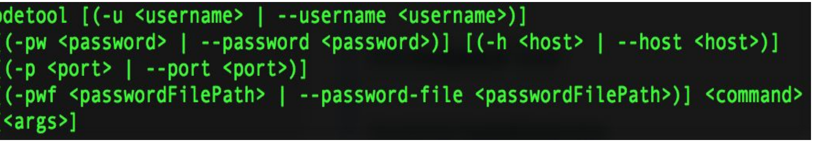
More accurate Data representation:

Map<byte[], SortedMap<Clustering, Row>>

- At the top-level, a table is a map of partitions indexed by their partition key.
- Partition is a sorted map of rows indexed by their “clustering”.
- The *Clustering* holds the values for the clustering columns of the CQL row it represents.
- And the Row object represents a given CQL row, associating to each column their value and timestamp.

Cassandra Data Model

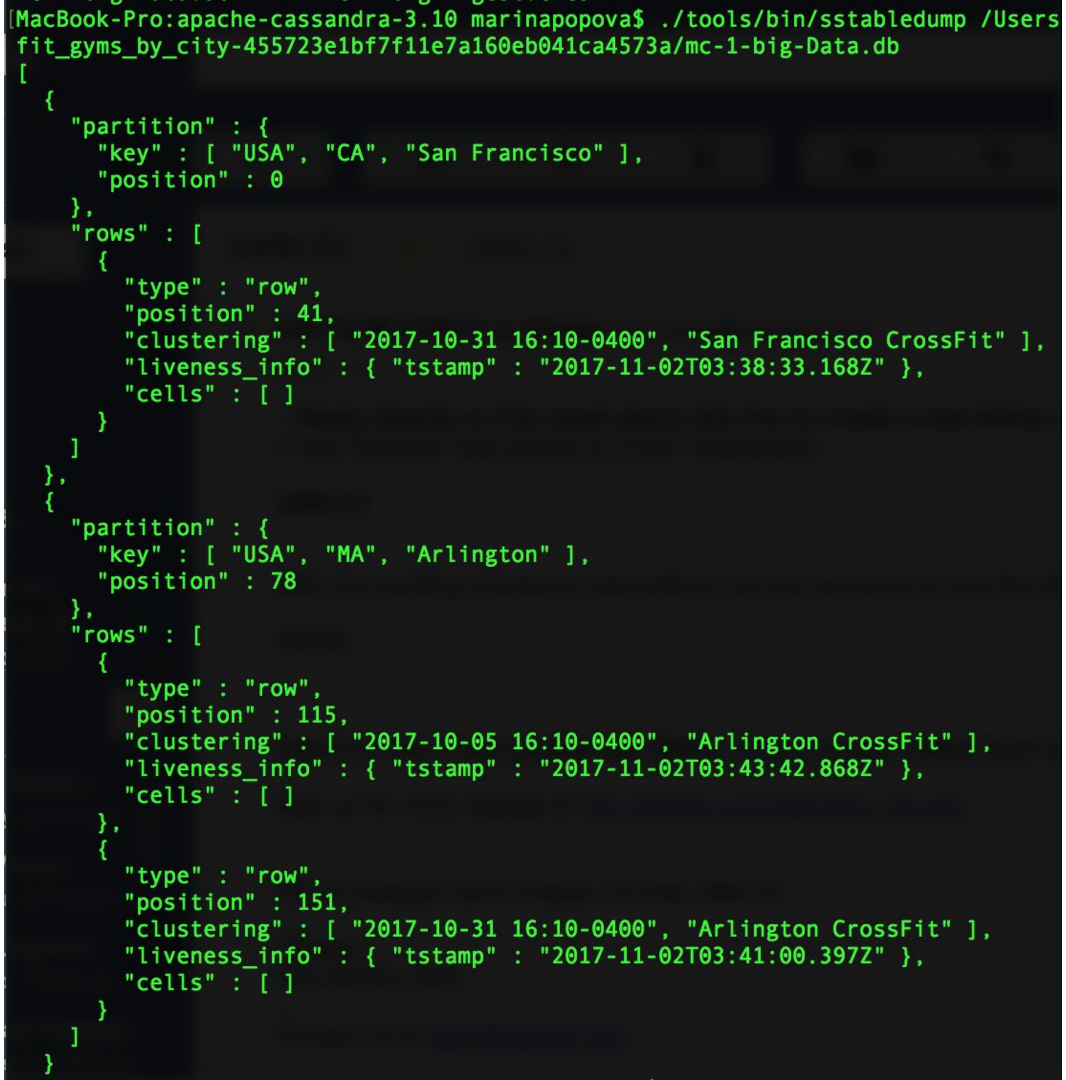
Cassandra Tools



```
detool [(-u <username> | --username <username>)]
(-pw <password> | --password <password>)] [(-h <host> | --host <host>)]
(-p <port> | --port <port>)]
(-pwf <passwordFilePath> | --password-file <passwordFilePath>)] <command>
<args>]
```

./bin/**nodetool** flush

./tools/bin/**sstabledump** .../apache-cassandra-3.10/data/data/cscie88/ crossfit_gyms_by_city-455723e1bf7f11e7a160eb041ca4573a/mc-1-big-Data.db



```
[MacBook-Pro:apache-cassandra-3.10 marinapopova$ ./tools/bin/sstabledump /Users
fit_gyms_by_city-455723e1bf7f11e7a160eb041ca4573a/mc-1-big-Data.db
[
  {
    "partition" : {
      "key" : [ "USA", "CA", "San Francisco" ],
      "position" : 0
    },
    "rows" : [
      {
        "type" : "row",
        "position" : 41,
        "clustering" : [ "2017-10-31 16:10-0400", "San Francisco CrossFit" ],
        "liveness_info" : { "tstamp" : "2017-11-02T03:38:33.168Z" },
        "cells" : [ ]
      }
    ]
  },
  {
    "partition" : {
      "key" : [ "USA", "MA", "Arlington" ],
      "position" : 78
    },
    "rows" : [
      {
        "type" : "row",
        "position" : 115,
        "clustering" : [ "2017-10-05 16:10-0400", "Arlington CrossFit" ],
        "liveness_info" : { "tstamp" : "2017-11-02T03:43:42.868Z" },
        "cells" : [ ]
      },
      {
        "type" : "row",
        "position" : 151,
        "clustering" : [ "2017-10-31 16:10-0400", "Arlington CrossFit" ],
        "liveness_info" : { "tstamp" : "2017-11-02T03:41:00.397Z" },
        "cells" : [ ]
      }
    ]
  }
```

Cassandra Data Model

Cassandra Data Modeling Goals:

Ref: https://www.datastax.com/dev/blog/the-most-important-thing-to-know-in-cassandra-data-modeling-the-primary-key

- **Spread data evenly around the cluster**
  - Partitions are distributed around the cluster based on a hash of the partition key.
  - To distribute work across nodes, it's desirable for every node in the cluster to have roughly the same amount of data.
- **Minimize the number of partitions read**
  - Partitions are groups of columns that share the same partition key
  - Since each partition may reside on a different node, the query coordinator will generally need to issue separate commands to separate nodes for each partition we query.
- **Satisfy a query by reading a single partition**
  - This means we will use roughly one table per query
  - Supporting multiple query patterns usually means we need more than one table
  - Data duplication is encouraged
  - This technique is called **Query Based Modeling**

Cassandra Time Series

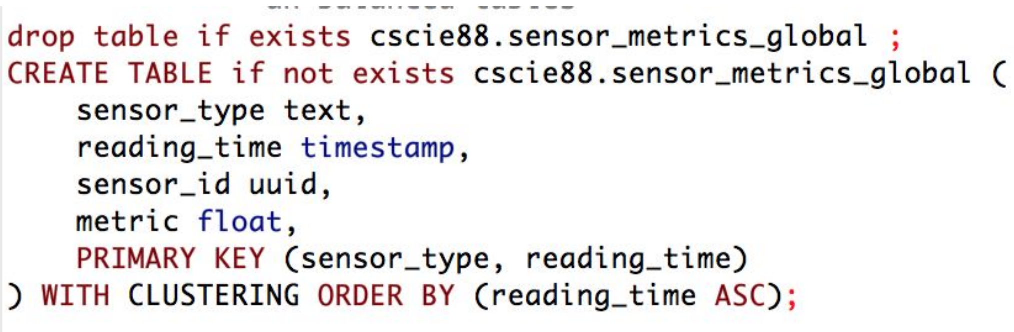
Time Series with Cassandra
http://massivetechinterview.blogspot.com/2015/10/cassandra.html

- Time Series data usually means that there is a **date/time component in your data model**
- time series data is generally **immutable**

Example: sensors data

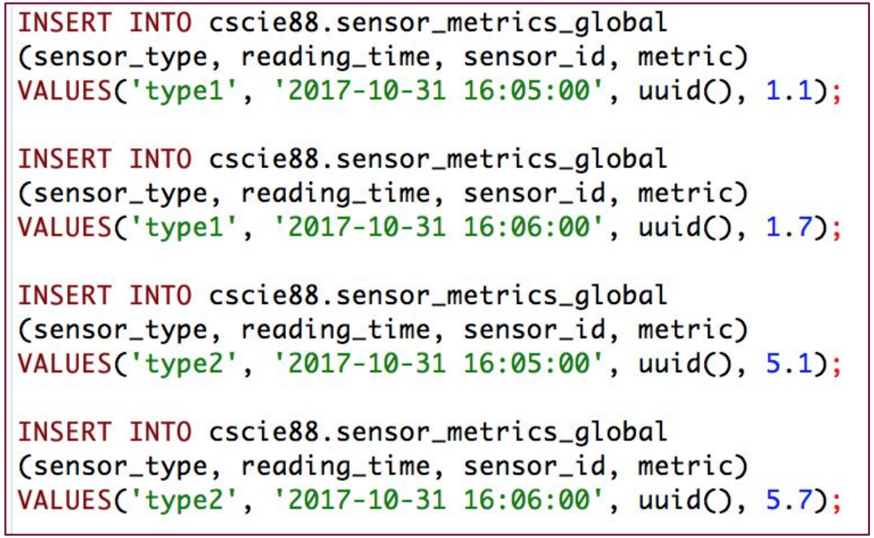
Partitioning Key: sensor_type
Clustering Key: reading_time

reading_time is sorted!!!

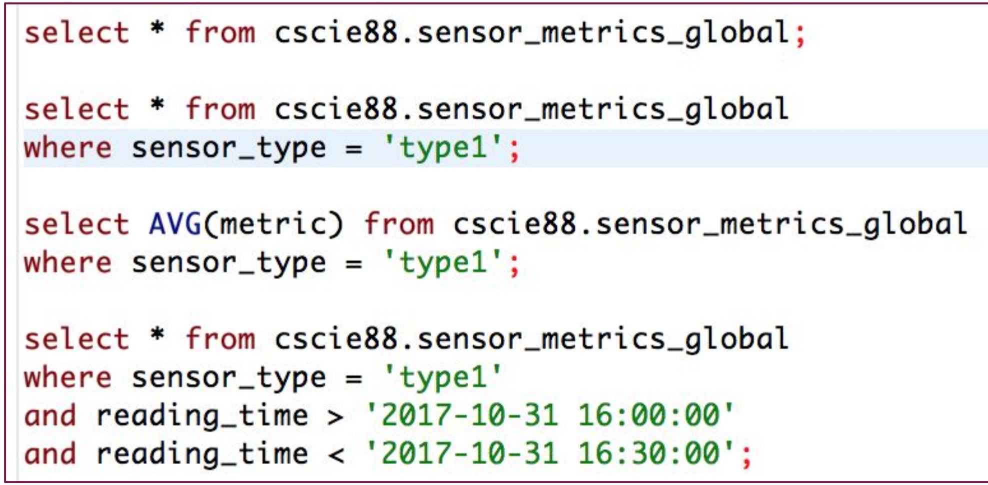


```
drop table if exists cscie88.sensor_metrics_global ;
CREATE TABLE if not exists cscie88.sensor_metrics_global (
    sensor_type text,
    reading_time timestamp,
    sensor_id uuid,
    metric float,
    PRIMARY KEY (sensor_type, reading_time)
) WITH CLUSTERING ORDER BY (reading_time ASC);
```

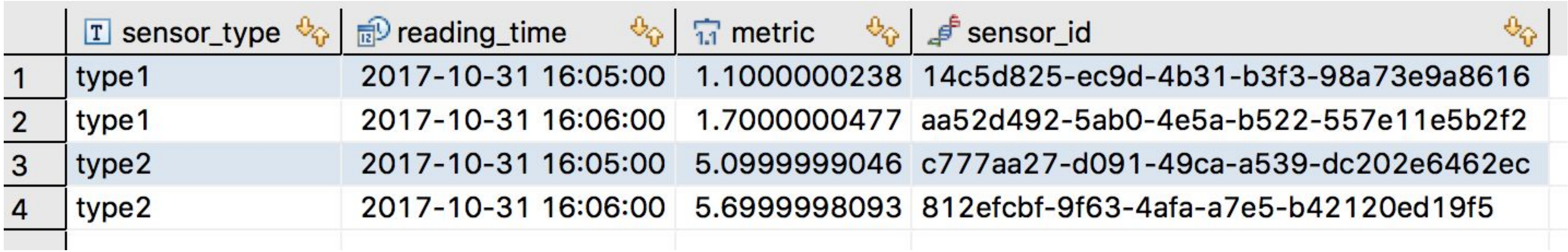
Cassandra Time Series



```
INSERT INTO cscie88.sensor_metrics_global
(sensor_type, reading_time, sensor_id, metric)
VALUES('type1', '2017-10-31 16:05:00', uuid(), 1.1);

INSERT INTO cscie88.sensor_metrics_global
(sensor_type, reading_time, sensor_id, metric)
VALUES('type1', '2017-10-31 16:06:00', uuid(), 1.7);

INSERT INTO cscie88.sensor_metrics_global
(sensor_type, reading_time, sensor_id, metric)
VALUES('type2', '2017-10-31 16:05:00', uuid(), 5.1);

INSERT INTO cscie88.sensor_metrics_global
(sensor_type, reading_time, sensor_id, metric)
VALUES('type2', '2017-10-31 16:06:00', uuid(), 5.7);
```



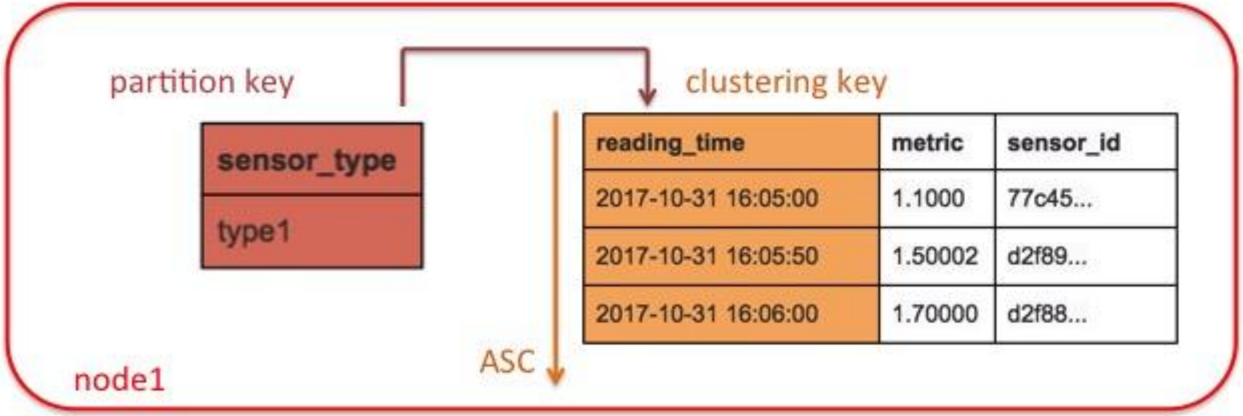
```
select * from cscie88.sensor_metrics_global;

select * from cscie88.sensor_metrics_global
where sensor_type = 'type1';

select AVG(metric) from cscie88.sensor_metrics_global
where sensor_type = 'type1';

select * from cscie88.sensor_metrics_global
where sensor_type = 'type1'
and reading_time > '2017-10-31 16:00:00'
and reading_time < '2017-10-31 16:30:00';
```



	sensor_type	reading_time	metric	sensor_id
1	type1	2017-10-31 16:05:00	1.1000000238	14c5d825-ec9d-4b31-b3f3-98a73e9a8616
2	type1	2017-10-31 16:06:00	1.7000000477	aa52d492-5ab0-4e5a-b522-557e11e5b2f2
3	type2	2017-10-31 16:05:00	5.0999999046	c777aa27-d091-49ca-a539-dc202e6462ec
4	type2	2017-10-31 16:06:00	5.6999998093	812efcbf-9f63-4afa-a7e5-b42120ed19f5

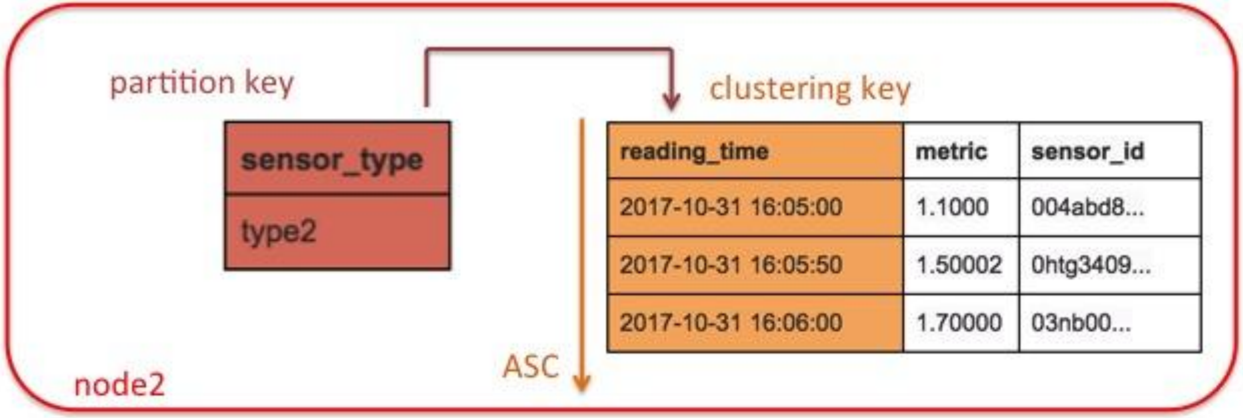
Cassandra

How the data looks on disk:



sensor_type
type1

reading_time	metric	sensor_id
2017-10-31 16:05:00	1.1000	77c45...
2017-10-31 16:05:50	1.50002	d2f89...
2017-10-31 16:06:00	1.70000	d2f88...



sensor_type
type2

reading_time	metric	sensor_id
2017-10-31 16:05:00	1.1000	004abd8...
2017-10-31 16:05:50	1.50002	0htg3409...
2017-10-31 16:06:00	1.70000	03nb00...

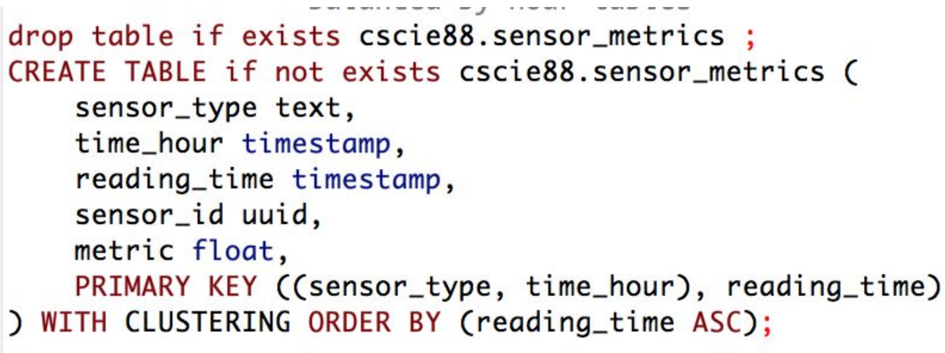
@Marina Popova

Cassandra Time Series

Problems with this model:
-- unbounded row growth
-- each new CQL row for a given sensor is actually adding columns to the same storage row

New version: add a more granular part to the Partitioning key

Partitioning Key: (sensor_type, time_hour)
Clustering Key: reading_time

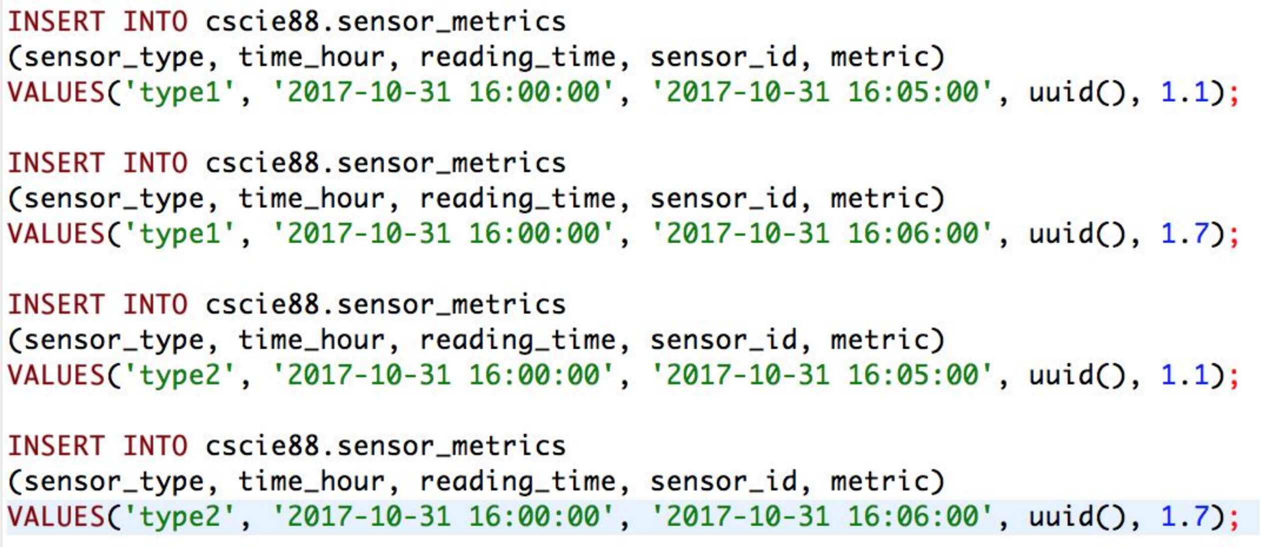


```
drop table if exists cscie88.sensor_metrics ;
CREATE TABLE if not exists cscie88.sensor_metrics (
    sensor_type text,
    time_hour timestamp,
    reading_time timestamp,
    sensor_id uuid,
    metric float,
    PRIMARY KEY ((sensor_type, time_hour), reading_time)
) WITH CLUSTERING ORDER BY (reading_time ASC);
```

When choosing values for your time buckets, a rule of thumb is to select an interval that allows you to perform the bulk of your queries using **only one or two buckets**. The more buckets you query, the more nodes will be involved to produce your result

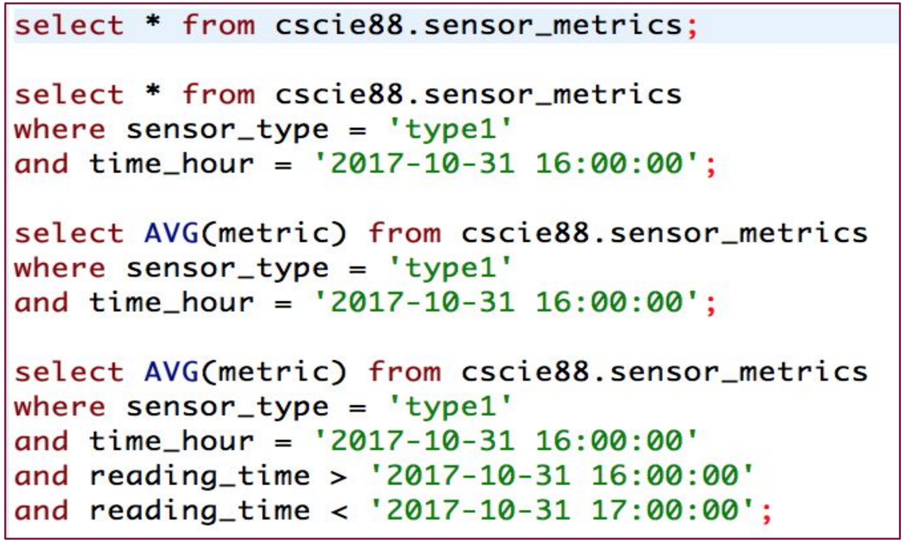
@Marina Popova

Cassandra Time Series

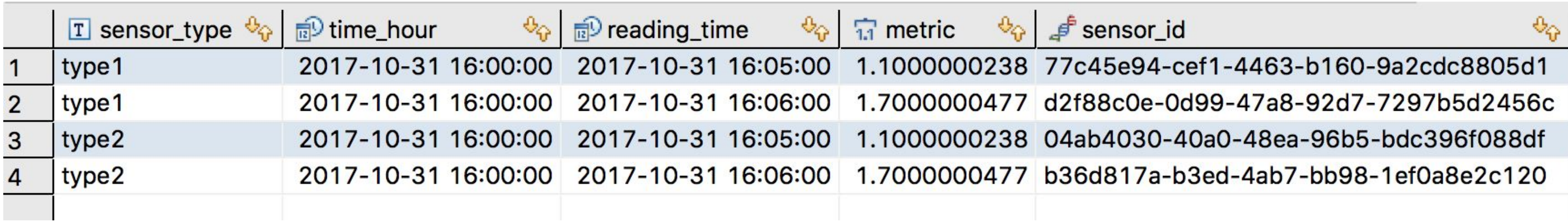


```
INSERT INTO cscie88.sensor_metrics
(sensor_type, time_hour, reading_time, sensor_id, metric)
VALUES('type1', '2017-10-31 16:00:00', '2017-10-31 16:05:00', uuid(), 1.1);

INSERT INTO cscie88.sensor_metrics
(sensor_type, time_hour, reading_time, sensor_id, metric)
VALUES('type1', '2017-10-31 16:00:00', '2017-10-31 16:06:00', uuid(), 1.7);

INSERT INTO cscie88.sensor_metrics
(sensor_type, time_hour, reading_time, sensor_id, metric)
VALUES('type2', '2017-10-31 16:00:00', '2017-10-31 16:05:00', uuid(), 1.1);

INSERT INTO cscie88.sensor_metrics
(sensor_type, time_hour, reading_time, sensor_id, metric)
VALUES('type2', '2017-10-31 16:00:00', '2017-10-31 16:06:00', uuid(), 1.7);
```

Cassandra Time Series



```
select * from cscie88.sensor_metrics;

select * from cscie88.sensor_metrics
where sensor_type = 'type1'
and time_hour = '2017-10-31 16:00:00';

select AVG(metric) from cscie88.sensor_metrics
where sensor_type = 'type1'
and time_hour = '2017-10-31 16:00:00';

select AVG(metric) from cscie88.sensor_metrics
where sensor_type = 'type1'
and time_hour = '2017-10-31 16:00:00'
and reading_time > '2017-10-31 16:00:00'
and reading_time < '2017-10-31 17:00:00';
```

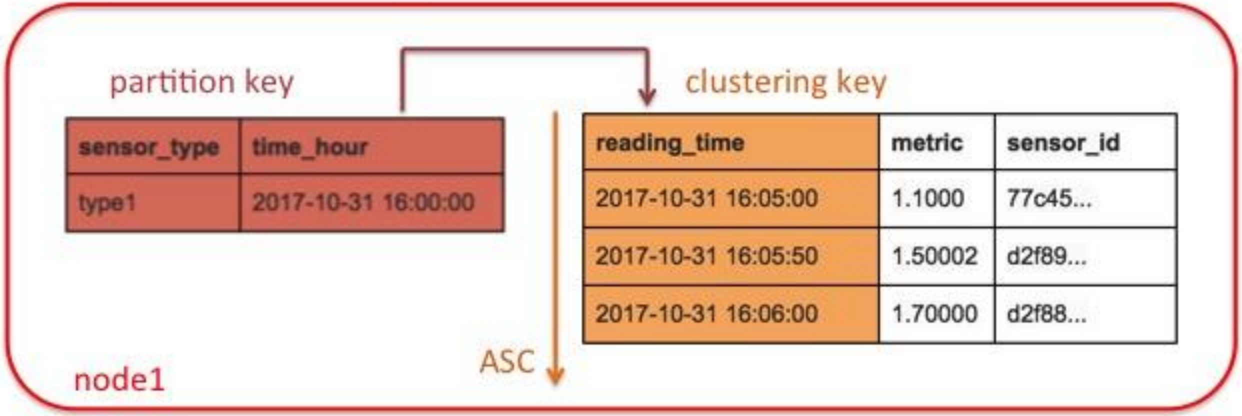


	sensor_type	time_hour	reading_time	metric	sensor_id
1	type1	2017-10-31 16:00:00	2017-10-31 16:05:00	1.1000000238	77c45e94-cef1-4463-b160-9a2cdc8805d1
2	type1	2017-10-31 16:00:00	2017-10-31 16:06:00	1.7000000477	d2f88c0e-0d99-47a8-92d7-7297b5d2456c
3	type2	2017-10-31 16:00:00	2017-10-31 16:05:00	1.1000000238	04ab4030-40a0-48ea-96b5-bdc396f088df
4	type2	2017-10-31 16:00:00	2017-10-31 16:06:00	1.7000000477	b36d817a-b3ed-4ab7-bb98-1ef0a8e2c120

@Marina Popova

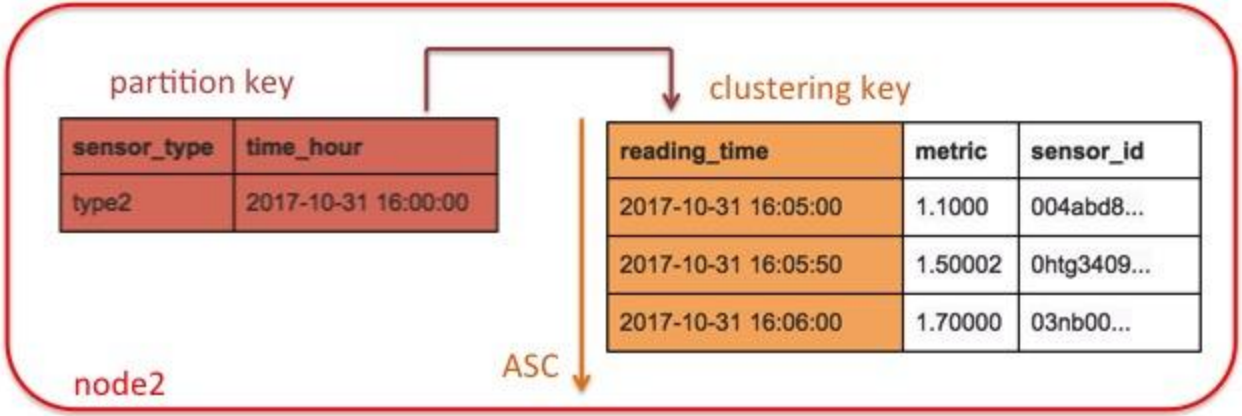
Cassandra

How the data looks on disk:



sensor_type	time_hour
type1	2017-10-31 16:00:00

reading_time	metric	sensor_id
2017-10-31 16:05:00	1.1000	77c45...
2017-10-31 16:05:50	1.50002	d2f89...
2017-10-31 16:06:00	1.70000	d2f88...



sensor_type	time_hour
type2	2017-10-31 16:00:00

reading_time	metric	sensor_id
2017-10-31 16:05:00	1.1000	004abd8...
2017-10-31 16:05:50	1.50002	0htg3409...
2017-10-31 16:06:00	1.70000	03nb00...

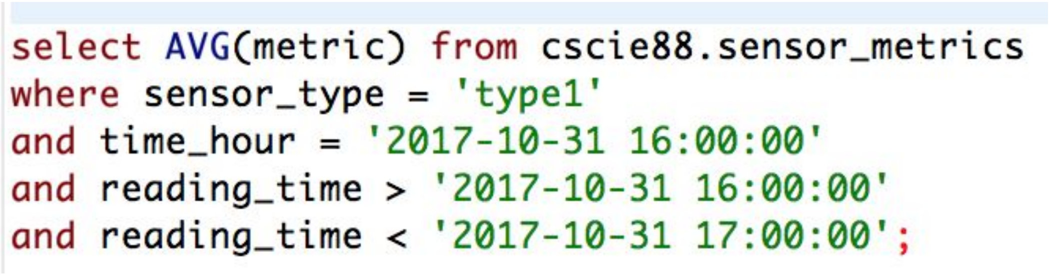
@Marina Popova

Cassandra Time Series

What make "time series"- type modeling so efficient?

- Very fast parallel writes per event
- Clustering key sorting!

This enables time-range queries as following:



```
select AVG(metric) from cscie88.sensor_metrics
where sensor_type = 'type1'
and time_hour = '2017-10-31 16:00:00'
and reading_time > '2017-10-31 16:00:00'
and reading_time < '2017-10-31 17:00:00';
```