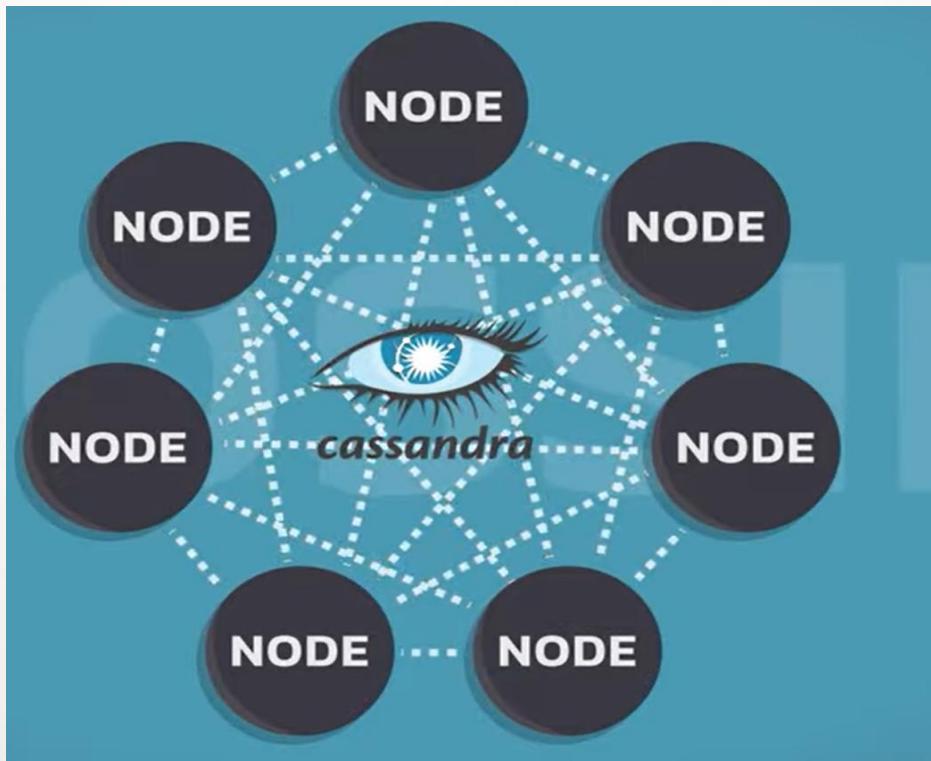


NOSQL Databases

Cassandra



High performance. Delivered.

consulting | technology | outsourcing



Cassandra Goals

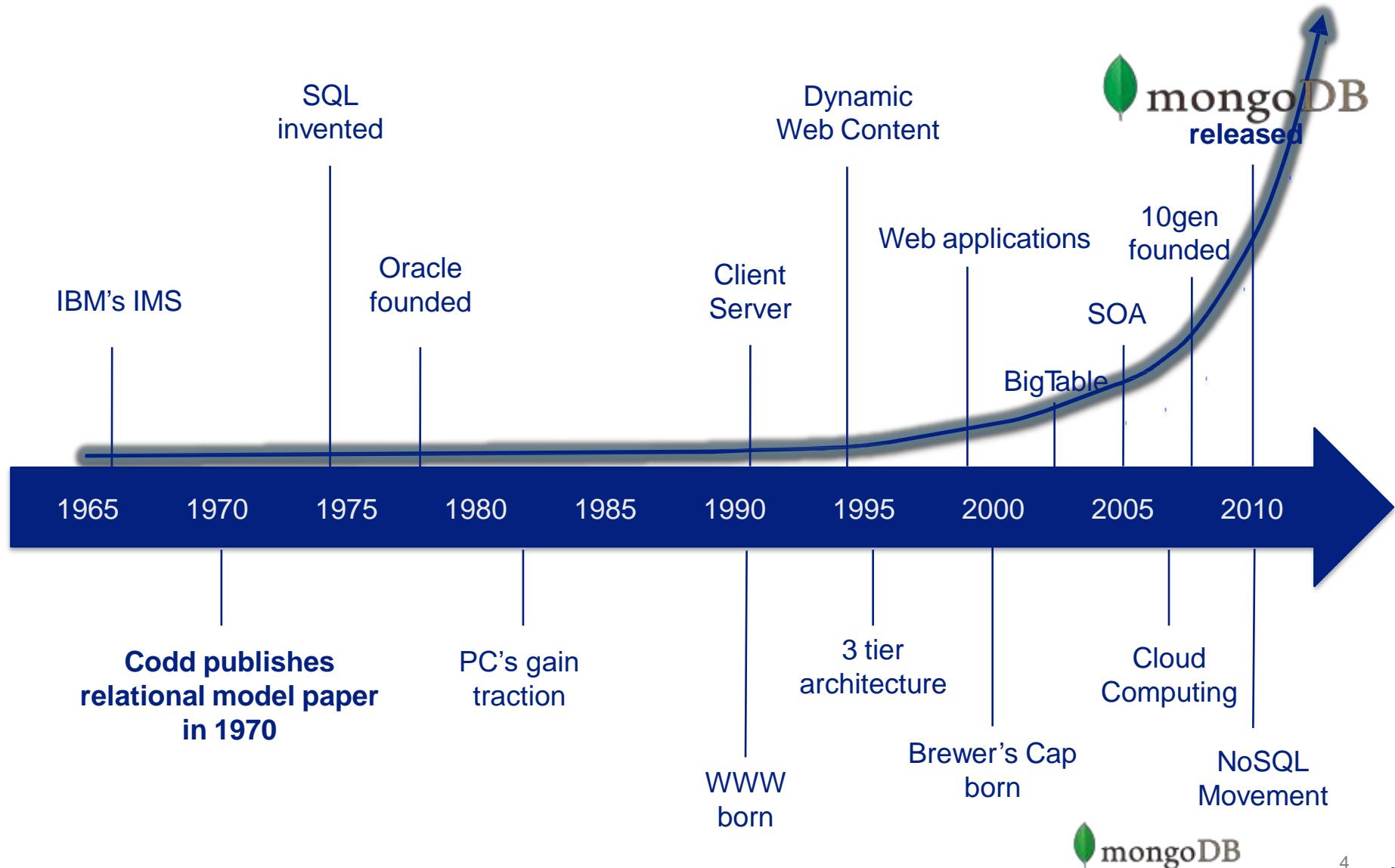
- What is a NoSQL database?
- NoSQL vs. SQL
- Types and examples
- When to use NoSQL
- NoSQL and Cloud
- Introduction to Cassandra
- Getting Started with Cassandra
- Installing Cassandra



Cassandra Goals

- Communicating with Cassandra
- Understanding Data Modelling in Cassandra
- Cassandra Multi node Cluster Setup
- Cassandra Monitoring and Maintenance
- Understanding Backup, Restore and Performance Tuning

Dawn of Databases to Present





Relational Database Strengths

- Data stored in a RDBMS is very compact (disk was more expensive)
- SQL and RDBMS made queries flexible with rigid schemas
- Rigid schemas helps optimize joins and storage
- Massive ecosystem of tools, libraries and integrations
- Been around 40 years!

Enter Big Data

- Gartner uses the 3Vs to define
- Volume - Big/difficult/extreme volume is relative
- Variety
 - Changing or evolving data
 - Uncontrolled formats
 - Does not easily adhere to a single schema
 - Unknown at design time
- Velocity
 - High or volatile inbound data
 - High query and read operations
 - Low latency

RDBMS challenges

DATA VARIETY & VOLATILITY

- Extremely difficult to find a single fixed schema



VOLUME & NEW ARCHITECTURES

- Systems scaling horizontally, not vertically
- Commodity servers
- Cloud Computing

TRANSACTIONAL MODEL

- $N \times$ Inserts or updates
- Distributed transactions



Enter NoSQL

- Non-relational been hanging around (MUMPS?)
- Modern NoSQL theory and offerings started in early 2000s
- Modern usage of term introduced in 2009
- NoSQL = Not Only SQL
- A collection of very different products
- Alternatives to relational databases when they are a bad fit
Motives
- Horizontally scalable (commodity server/cloud computing)
- Flexibility



What is NoSQL

- NoSQL Database is a non-relational Data Management System, that does not require a fixed schema.
- It avoids joins, and is easy to scale.
- The major purpose of using a NoSQL database is for distributed data stores with humongous data storage needs.
- NoSQL is used for Big data and real-time web apps.
- For example, companies like Twitter, Facebook and Google collect terabytes of user data every single day.
- NoSQL database stands for "Not Only SQL" or "Not SQL."

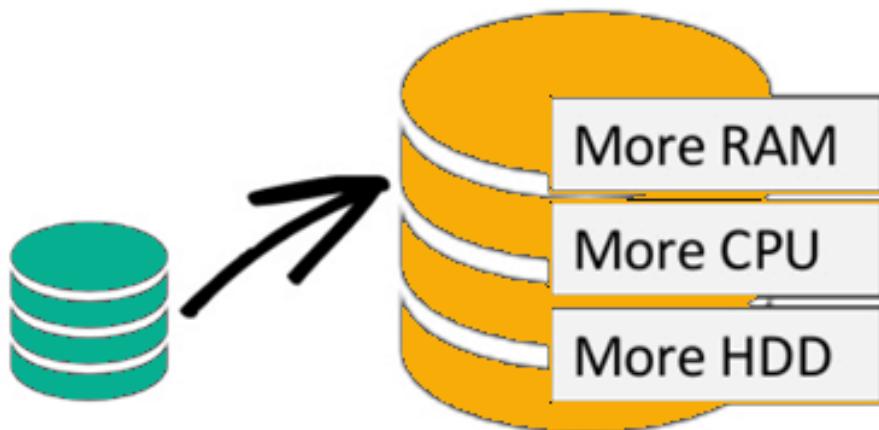


Why NoSQL

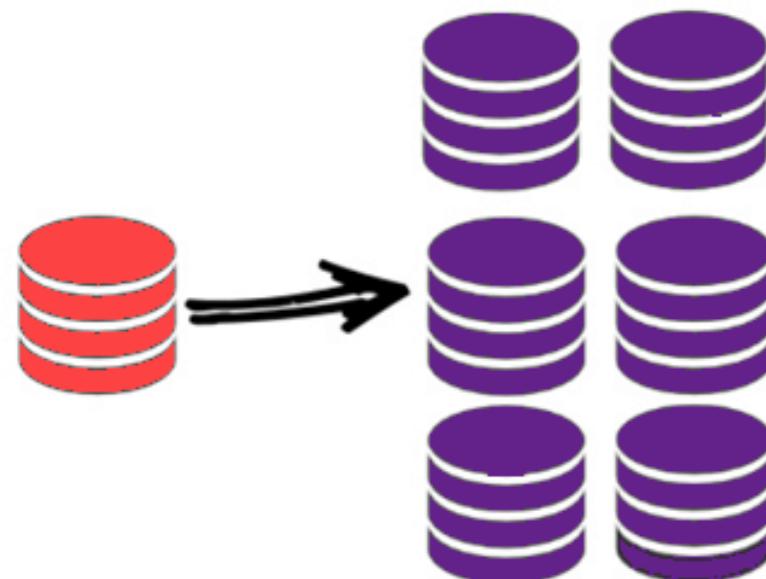
- The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data.
- The system response time becomes slow when you use RDBMS for massive volumes of data.
- To resolve this problem, we could "scale up" our systems by upgrading our existing hardware. This process is expensive.
- The alternative for this issue is to distribute database load on multiple hosts whenever the load increases. This method is known as "scaling out."

Why NoSQL

Scale-Up (*vertical* scaling):



Scale-Out (*horizontal* scaling):

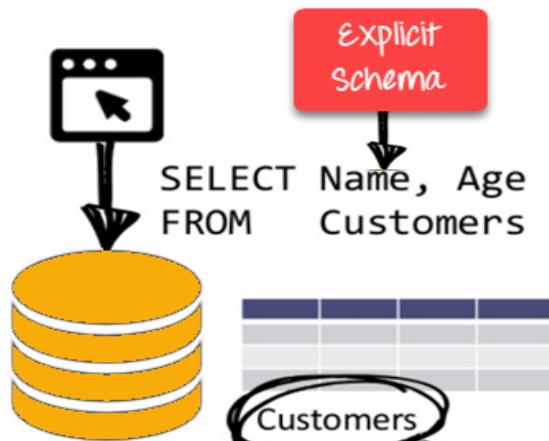


Commodity
Hardware

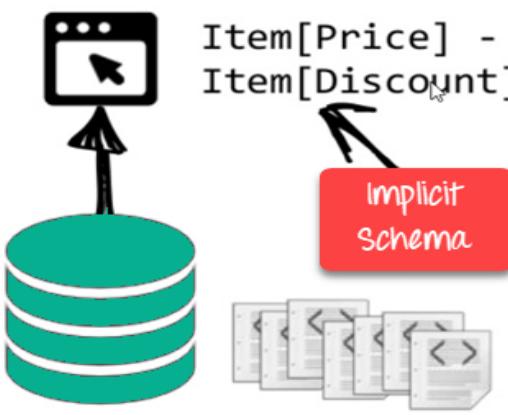
Schema-free

- NoSQL databases are either schema-free or have relaxed schemas
- Do not require any sort of definition of the schema of the data
- Offers heterogeneous structures of data in the same domain

RDBMS:



NoSQL DB:





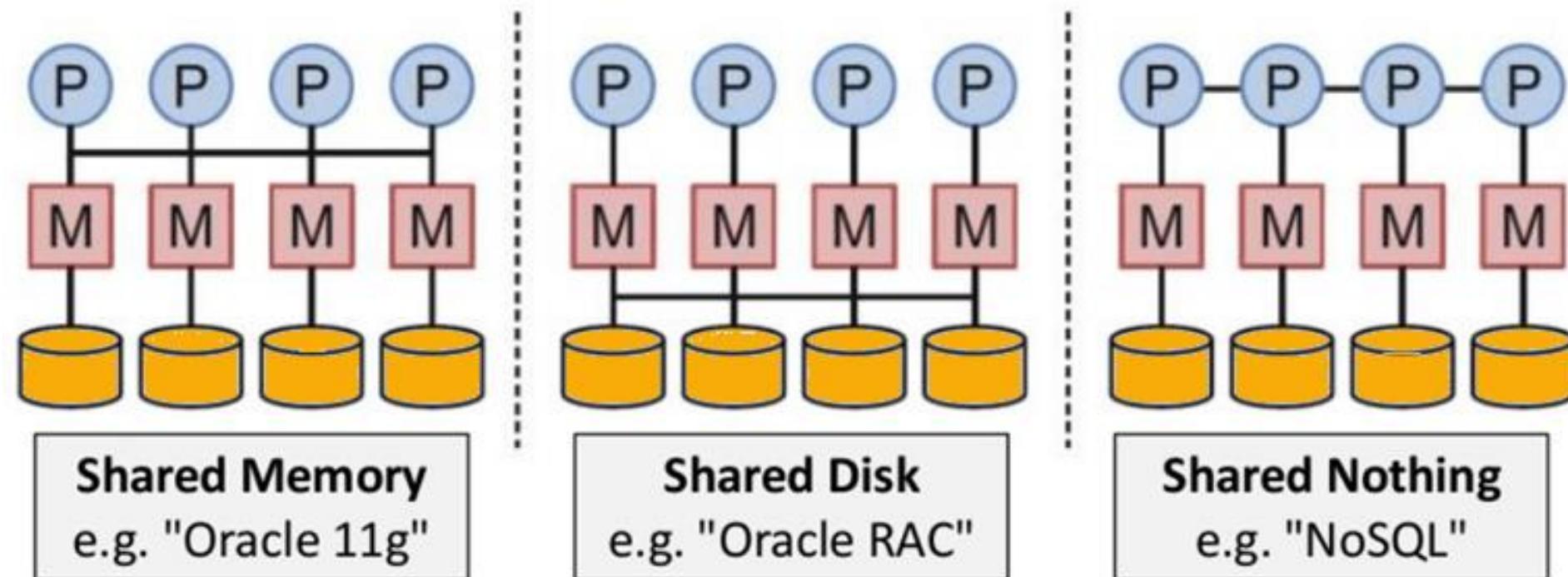
Simple API

- Offers easy to use interfaces for storage and querying data provided
- APIs allow low-level data manipulation & selection methods
- Text-based protocols mostly used with HTTP REST with JSON
- Mostly used no standard based NoSQL query language
- Web-enabled databases running as internet-facing services

Distributed

- Multiple NoSQL databases can be executed in a distributed fashion
- Offers auto-scaling and fail-over capabilities
- Often ACID concept can be sacrificed for scalability and throughput
- Mostly no synchronous replication between distributed nodes Asynchronous Multi-Master Replication, peer-to-peer, HDFS Replication
- Only providing eventual consistency
- Shared Nothing Architecture. This enables less coordination and higher distribution.

Distributed





Types of NoSQL Databases

- NoSQL Databases are mainly categorized into four types: Key-value pair, Column-oriented, Graph-based and Document-oriented.
- Every category has its unique attributes and limitations.
- None of the above-specified database is better to solve all the problems.
- Users should select the database based on their product needs.
- Types of NoSQL Databases:
 - Key-value Pair Based
 - Column-oriented Graph
 - Graphs based
 - Document-oriented

Types of NoSQL Databases

Key Value



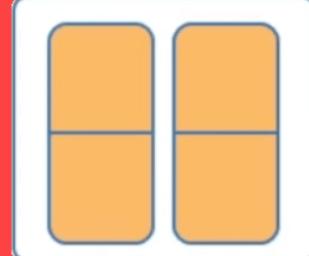
Example:
Riak, Tokyo Cabinet, Redis
server, Memcached,
Scalarmis

Document-Based



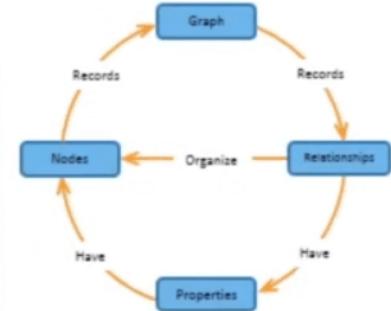
Example:
MongoDB, CouchDB,
OrientDB, RavenDB

Column-Based



Example:
BigTable, Cassandra,
Hbase,
Hypertable

Graph-Based

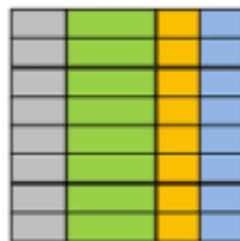


Example:
Neo4J, InfoGrid, Infinite
Graph, Flock DB

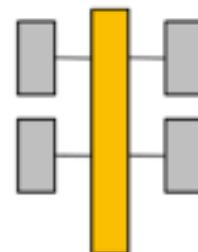
What is NoSQL

SQL Database

Relational

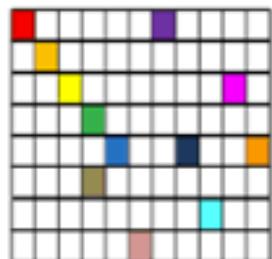


Analytical (OLAP)

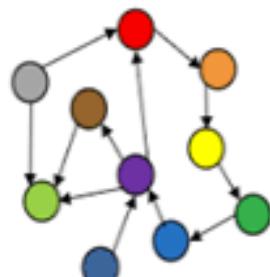


NoSQL Database

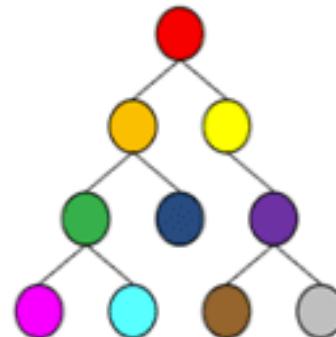
Column-Family



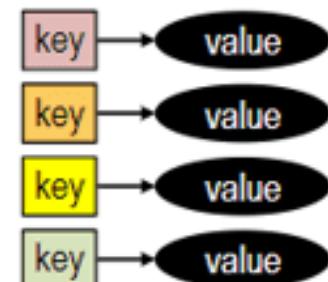
Graph



Document



Key-Value





Key Value Pair Based

- Data is stored in key/value pairs. It is designed in such a way to handle lots of data and heavy load.
- Key-value pair storage databases store data as a hash table where each key is unique, and the value can be a JSON, BLOB(Binary Large Objects), string, etc.
- For example, a key-value pair may contain a key like "Website" associated with a value like "virtusa".

Key Value Pair Based

- It is one of the most basic NoSQL database example.
- This kind of NoSQL database is used as a collection, dictionaries, associative arrays, etc.
- Key value stores help the developer to store schema-less data. They work best for shopping cart contents.
- Redis, Dynamo, Riak are some NoSQL examples of key-value store DataBases. They are all based on Amazon's Dynamo paper.

Column-based



- Column-oriented databases work on columns and are based on BigTable paper by Google. Every column is treated separately. Values of single column databases are stored contiguously.

ColumnFamily			
Row Key	Column Name		
	Key	Key	Key
Value	Value	Value	Value
	Column Name		
Value	Key	Key	Key
	Value	Value	Value

Column-based

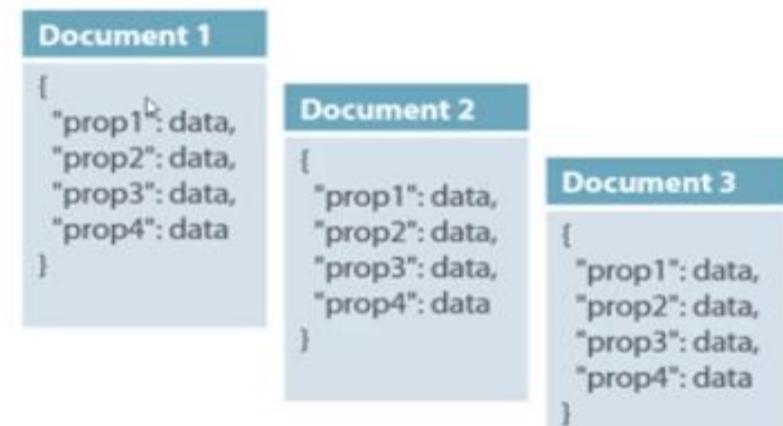


- They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN etc. as the data is readily available in a column.
- Column-based NoSQL databases are widely used to manage data warehouses, business intelligence, CRM, Library card catalogs, HBase, Cassandra, HBase, Hypertable are NoSQL query examples of column based database.

Document-Oriented

- Document-Oriented NoSQL DB stores and retrieves data as a key value pair but the value part is stored as a document.
- The document is stored in JSON or XML formats. The value is understood by the DB and can be queried.

Col1	Col2	Col3	Col4
Data	Data	Data	Data
Data	Data	Data	Data
Data	Data	Data	Data

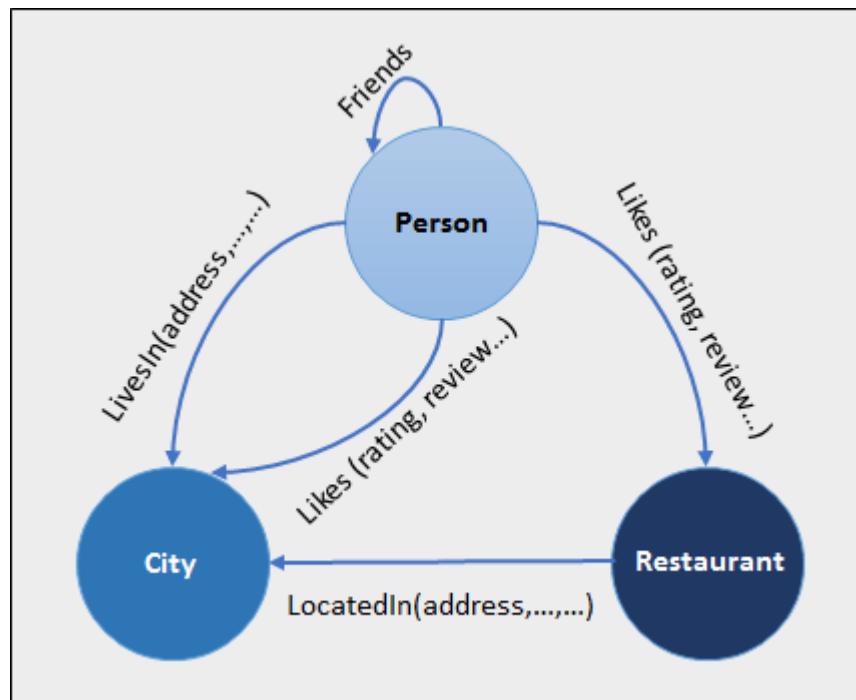


Document-Oriented

- In the above diagram on left we can see we have rows and columns.
- In the right, we have a document database which has a similar structure to JSON.
- Document database, we have data store like JSON object.
- The document type is mostly used for CMS systems, blogging platforms, real-time analytics & e-commerce applications.
- It should not use for complex transactions which require multiple operations or queries against varying aggregate structures.
- Amazon SimpleDB, CouchDB, MongoDB, Riak, Lotus Notes, MongoDB, are popular Document originated DBMS systems.

Graph-Based

- A graph type database stores entities as well the relations amongst those entities.
- The entity is stored as a node with the relationship as edges.
- An edge gives a relationship between nodes. Every node and edge has a unique identifier.



Graph-Based

- Compared to a relational database where tables are loosely connected, a Graph database is a multi-relational in nature.
- Traversing relationship is fast as they are already captured into the DB, and there is no need to calculate them.
- Graph base database mostly used for social networks, logistics, spatial data.
- Neo4J, Infinite Graph, OrientDB, FlockDB are some popular graph-based databases.

SQL vs NoSQL

RDBMS	NoSQL
<ul style="list-style-type: none"> • Data is stored in a relational model, with rows and columns. • A row contains information about an item while columns contain specific information, such as 'Model', 'Date of Manufacture', 'Color'. • Follows fixed schema. Meaning, the columns are defined and locked before data entry. In addition, each row contains data for each column. • Supports vertical scaling. Scaling an RDBMS across multiple servers is a challenging and time-consuming process. • Atomicity, Consistency, Isolation & Durability(ACID) Compliant 	<ul style="list-style-type: none"> • Data is stored in a host of different databases, with different data storage models. • Follows dynamic schemas. Meaning, you can add columns anytime. • Supports horizontal scaling. You can scale across multiple servers. Multiple servers are cheap commodity hardware or cloud instances, which make scaling cost-effective compared to vertical scaling. • Not ACID Compliant.



What is the CAP Theorem?

- CAP theorem is also called brewer's theorem. It states that it is impossible for a distributed data store to offer more than two out of three guarantees
 1. Consistency
 2. Availability
 3. Partition Tolerance



What is the CAP Theorem?

- Consistency:
 - The data should remain consistent even after the execution of an operation. This means once data is written, any future read request should contain that data. For example, after updating the order status, all the clients should be able to see the same data.
- Availability:
 - The database should always be available and responsive. It should not have any downtime.



What is the CAP Theorem?

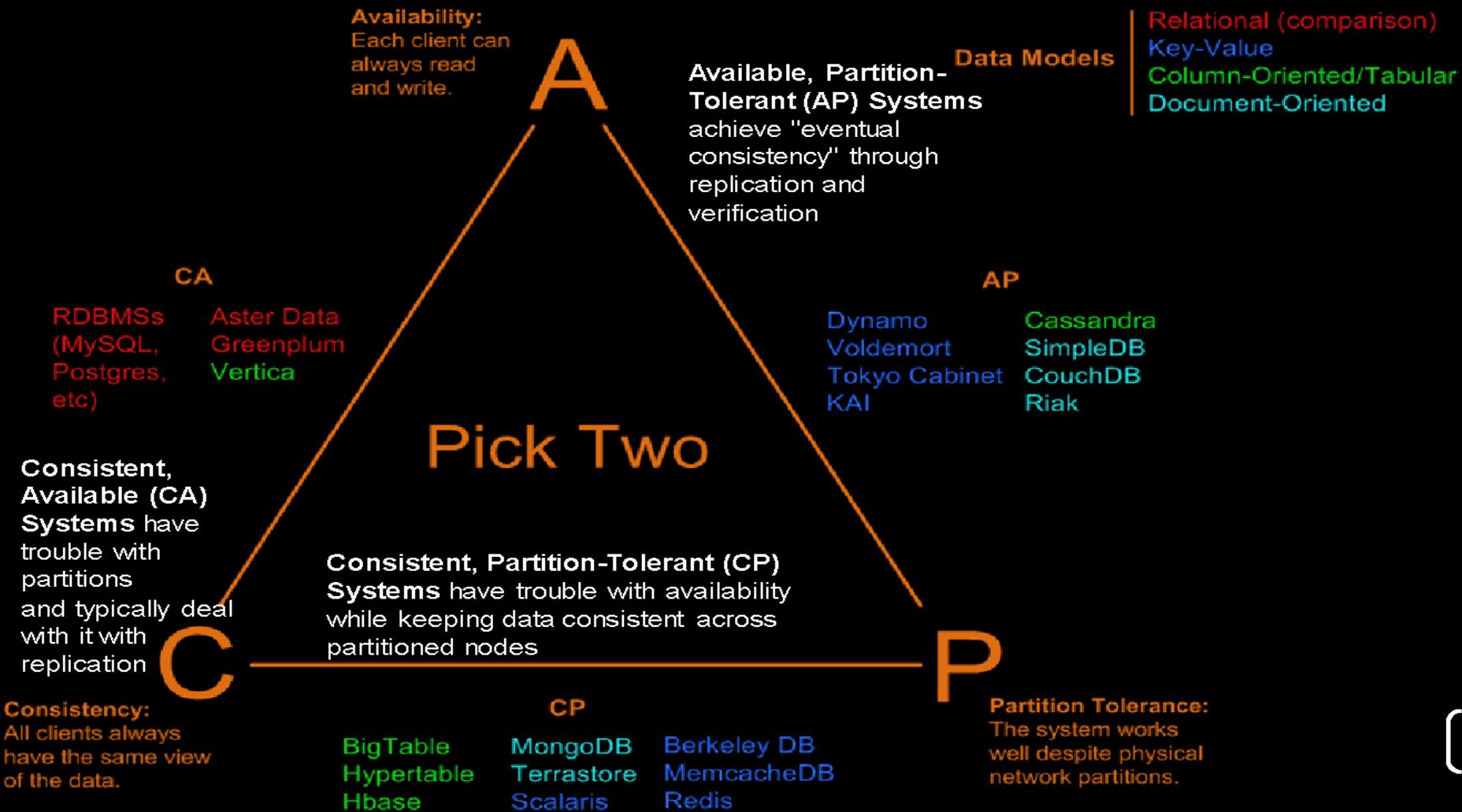
- Partition Tolerance
 - Partition Tolerance means that the system should continue to function even if the communication among the servers is not stable.
 - For example, the servers can be partitioned into multiple groups which may not communicate with each other. Here, if part of the database is unavailable, other parts are always unaffected.



What is the CAP Theorem?

- Partition Tolerance
 - Partition Tolerance means that the system should continue to function even if the communication among the servers is not stable.
 - For example, the servers can be partitioned into multiple groups which may not communicate with each other. Here, if part of the database is unavailable, other parts are always unaffected.

Visual Guide to NoSQL Systems



Apache Cassandra

Scalability



Availability



Distributed



Cloud-native





Apache Cassandra

- Apache Cassandra is **an open source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tunable consistent, column-oriented database** that bases its distribution design on Amazon's Dynamo and its data model on Google's Bigtable.



Apache Cassandra

- Cassandra **is distributed**, which means that it can run on multiple machines while appearing to users as a unified whole.
- Cassandra is distributed and decentralized, there is no single point of failure, which supports high availability



Apache Cassandra

- Scalability is an architectural feature of a system that can continue serving a greater number of requests with little degradation in performance.
- **Vertical scaling** simply adding more hardware capacity and memory to your existing machine is the easiest way to achieve this.
- **Horizontal scaling** means adding more machines that have all or some of the data on them so that no one machine must bear the entire burden of serving requests.
- But then the software itself must have an internal mechanism for keeping its data in sync with the other nodes in the cluster.



Apache Cassandra

- **Elastic scalability** refers to a special property of horizontal scalability.
- It means that cluster can seamlessly scale up and scale back down.
- To do this, the cluster must be able to accept new nodes that can begin participating by getting a copy of some or all the data and start serving new user requests without major disruption or reconfiguration of the entire cluster.
- Without restarting process, without changing application queries scaling possible.
- No need to manually rebalance the data.
- Just add another machine—Cassandra will find it and start sending it work.



Apache Cassandra

- Cassandra is **highly available**.
- We can replace failed nodes in the cluster with no downtime.
- We can replicate data to multiple data centers to offer improved local performance.
- If one data center experiences a catastrophe such as fire or flood still, we can retrieve data from another DC.



Apache Cassandra

- **Consistency** essentially means that a read always returns the most recently written value.
- Consider two customers are attempting to put the same item into their shopping carts on an ecommerce site.
- If I place the last item in stock into my cart an instant after you do, you should get the item added to your cart, and I should be informed that the item is no longer available for purchase.
- This is guaranteed to happen when the state of a write is consistent among all nodes that have that data.

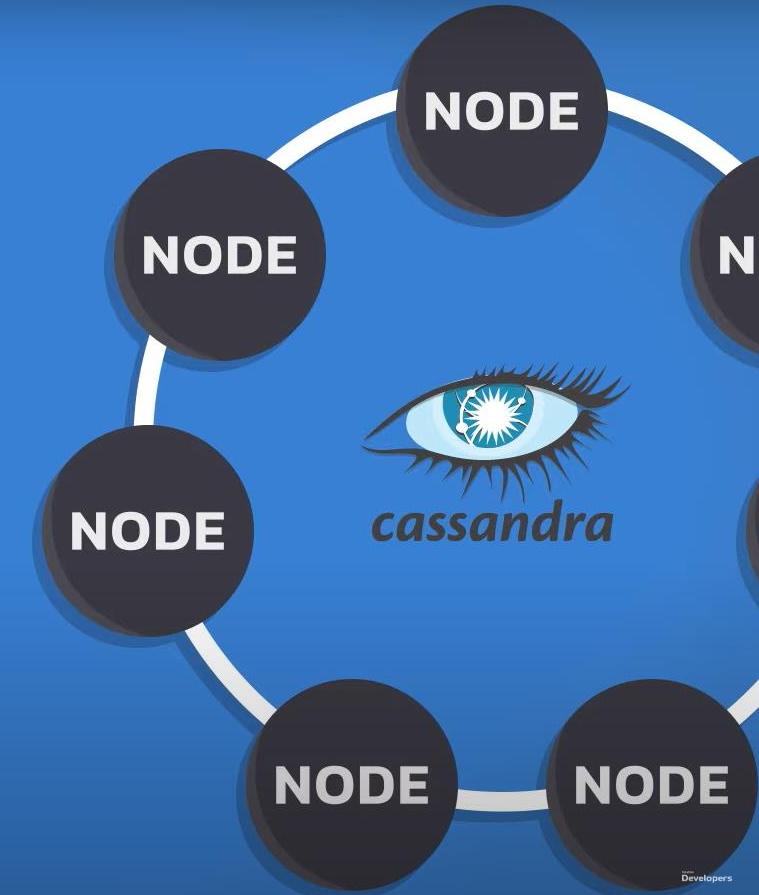


Apache Cassandra

- Cassandra is more accurately termed “tunably consistent,” which means it allows you to easily decide the level of consistency you require, in balance with the level of availability.

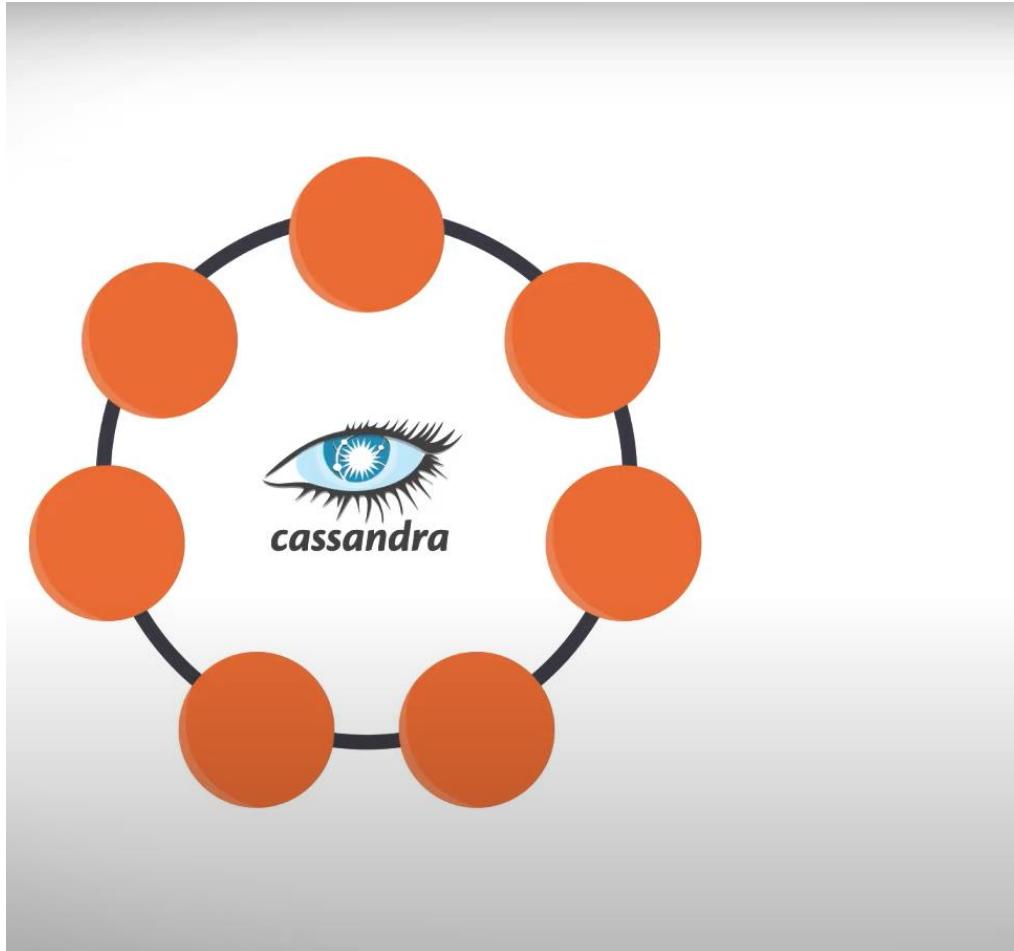
Why Apache Cassandra

- Petabyte Database
- High Availability
- Geographic Distribution
- Performance
- Vendor Independant





Why Apache Cassandra

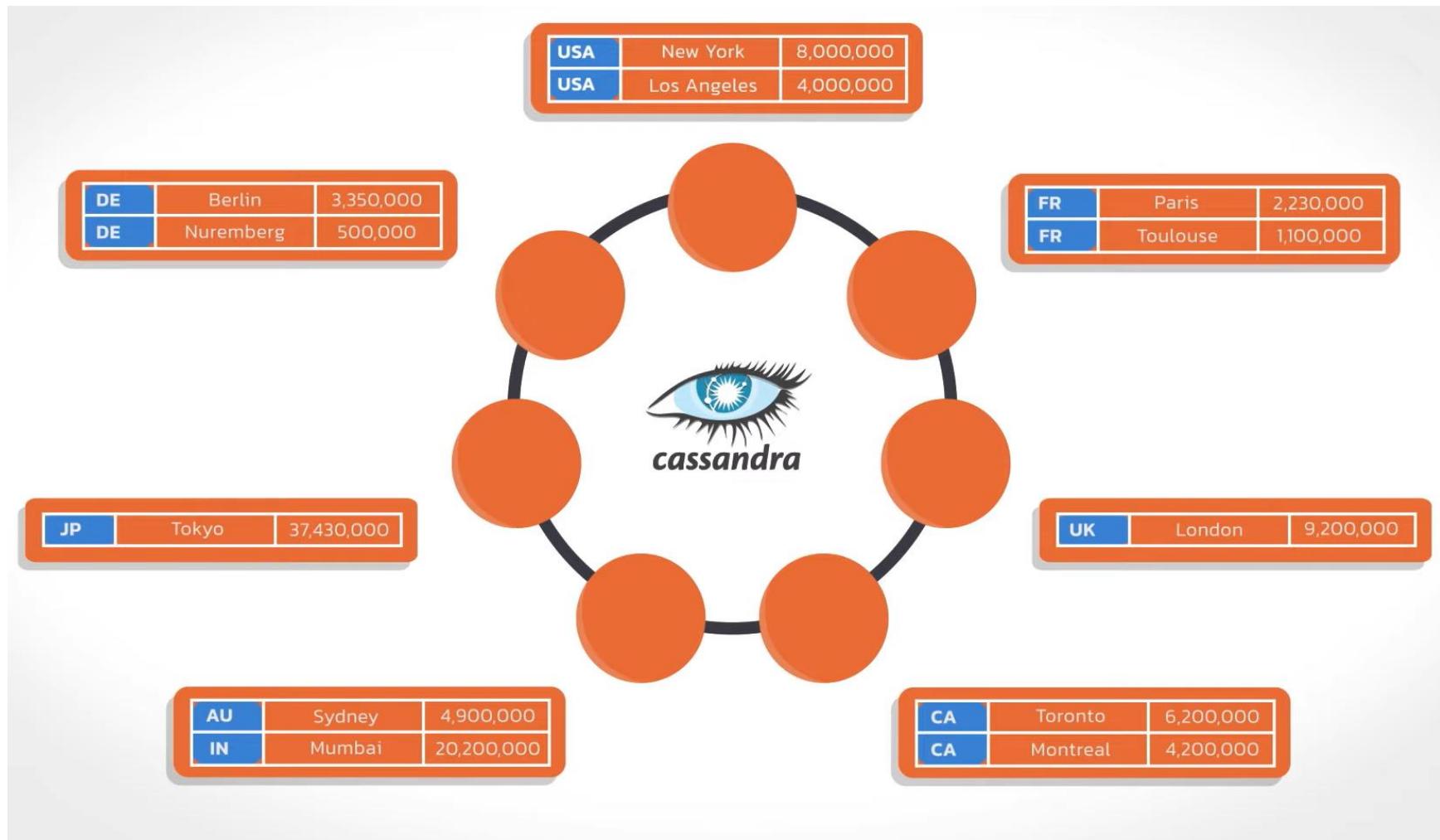


PARTITIONS		
Country	City	Population
USA	New York	8,000,000
USA	Los Angeles	4,000,000
FR	Paris	2,230,000
DE	Berlin	3,350,000
UK	London	9,200,000
AU	Sydney	4,900,000
DE	Nuremberg	500,000
CA	Toronto	6,200,000
CA	Montreal	4,200,000
FR	Toulouse	1,100,000
JP	Tokyo	37,430,000
IN	Mumbai	20,200,000

Partition Key

Developers

Why Apache Cassandra





Column Based Database Cassandra

- Cassandra was initially developed at Facebook by two Indians Avinash Lakshman (one of the authors of Amazon's Dynamo) and Prashant Malik.
- It was developed to power the Facebook inbox search feature.
- The following points specify the most important happenings in Cassandra history:
 - It was developed for Facebook inbox search feature.
 - It was open sourced by Facebook in July 2008.
 - It was accepted by Apache Incubator in March 2009.
 - Cassandra is a top-level project of Apache since February 2010.
 - The latest version of Apache Cassandra is 3.2.1.



Who Uses Cassandra

- Mahalo uses it for its primary near-time data store.
- Facebook still uses it for inbox search, though they are using a proprietary fork.
- Digg uses it for its primary near-time data store.
- Rackspace uses it for its cloud service, monitoring, and logging.
- Reddit uses it as a persistent cache.
- Cloudkick uses it for monitoring statistics and analytics.
- Ooyala uses it to store and serve near real-time video analytics data.
- SimpleGeo uses it as the main data store for its real-time location infrastructure.
- Onespots uses it for a subset of its main data store.



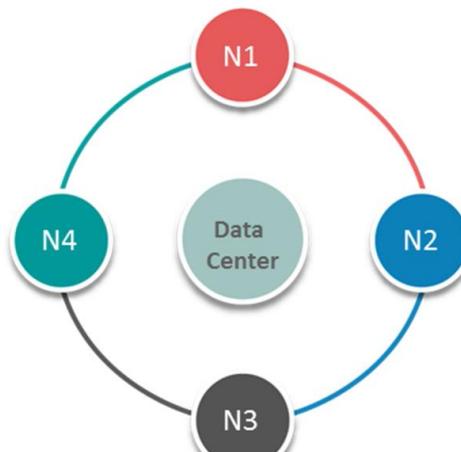
Column Based Database Cassandra

Version	Original release date	Latest version	Release date	Status ^[16]
0.6	2010-04-12	0.6.13	2011-04-18	No longer maintained
0.7	2011-01-10	0.7.10	2011-10-31	No longer maintained
0.8	2011-06-03	0.8.10	2012-02-13	No longer maintained
1.0	2011-10-18	1.0.12	2012-10-04	No longer maintained
1.1	2012-04-24	1.1.12	2013-05-27	No longer maintained
1.2	2013-01-02	1.2.19	2014-09-18	No longer maintained
2.0	2013-09-03	2.0.17	2015-09-21	No longer maintained
2.1	2014-09-16	2.1.22	2020-08-31	No longer maintained
2.2	2015-07-20	2.2.19	2020-11-04	No longer maintained
3.0	2015-11-09	3.0.28	2022-05-13	Still supported
3.11	2017-06-23	3.11.14	2022-05-13	Still supported
4.0	2021-07-26	4.0.7	2022-08-25	Latest release
4.1	2022-10-05	4.1-beta1	2022-10-05	Beta

Legend: Old version Older version, still maintained Latest version Latest preview version

Cassandra Architecture

- The architecture of Cassandra contributes to a database that *scales* and *performs with continuous availability*
- It has a *masterless “ring”* distributed architecture that is elegant, and easy to set up and maintain
- Cassandra’s built-for-scale architecture is *capable of handling large amounts of data* and *thousands of concurrent* users/operations per second, across multiple data centers easily

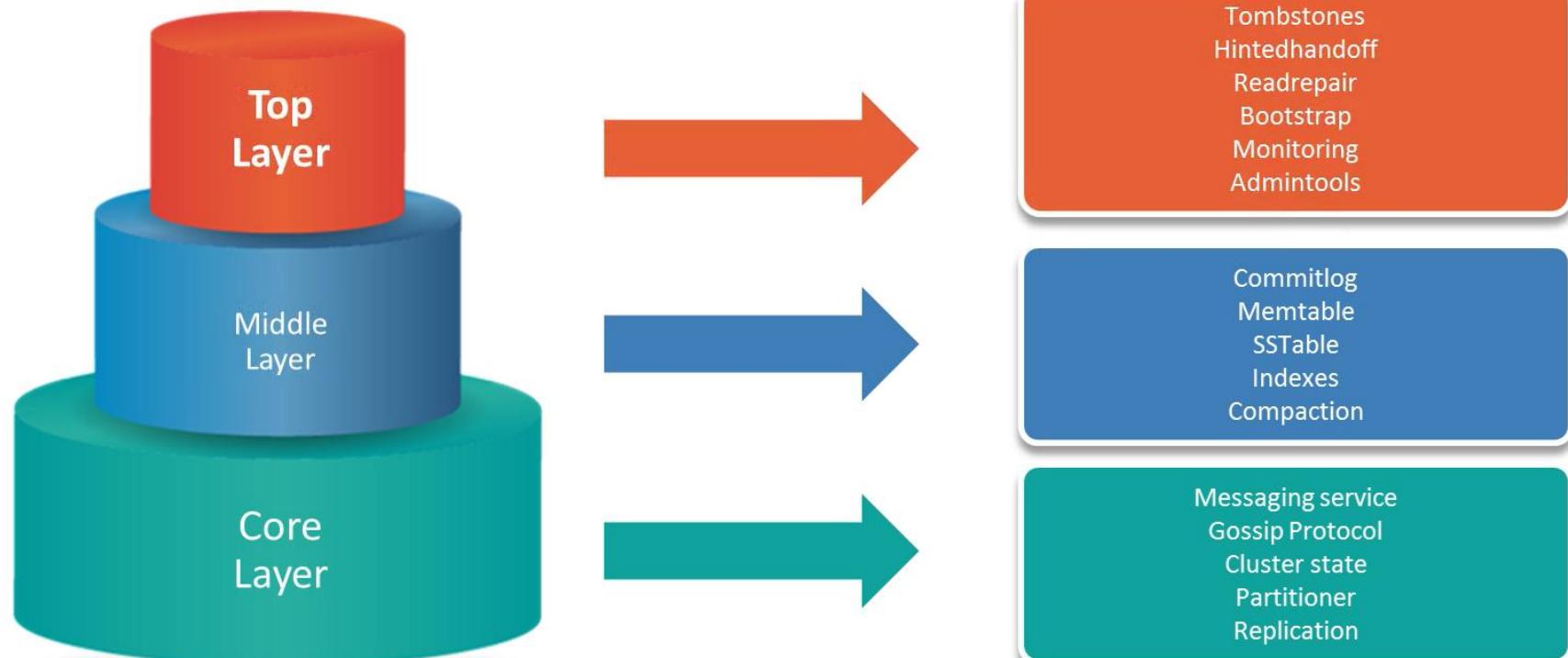




Cassandra Architecture

- Cassandra was designed to handle big data workloads across multiple nodes without a single point of failure.
- It has a peer-to-peer distributed system across its nodes, and data is distributed among all the nodes in a cluster.
- In Cassandra, each node is independent and at the same time interconnected to other nodes.
- All the nodes in a cluster play the same role.
- Every node in a cluster can accept read and write requests, regardless of where the data is actually located in the cluster.
- In the case of failure of one node, Read/Write requests can be served from other nodes in the network.

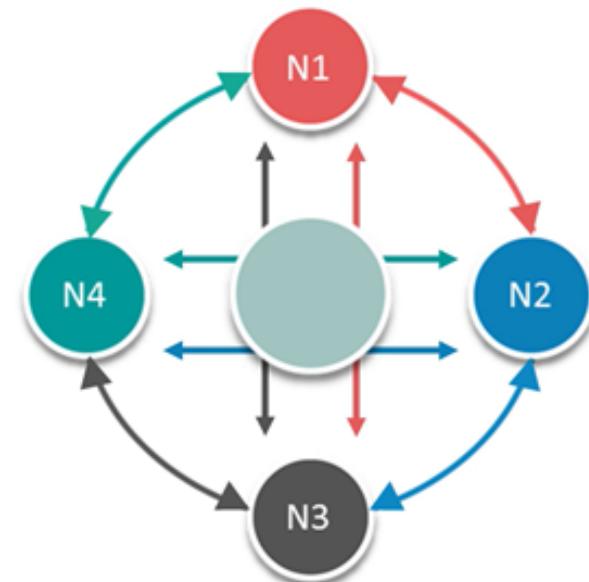
Cassandra Architecture



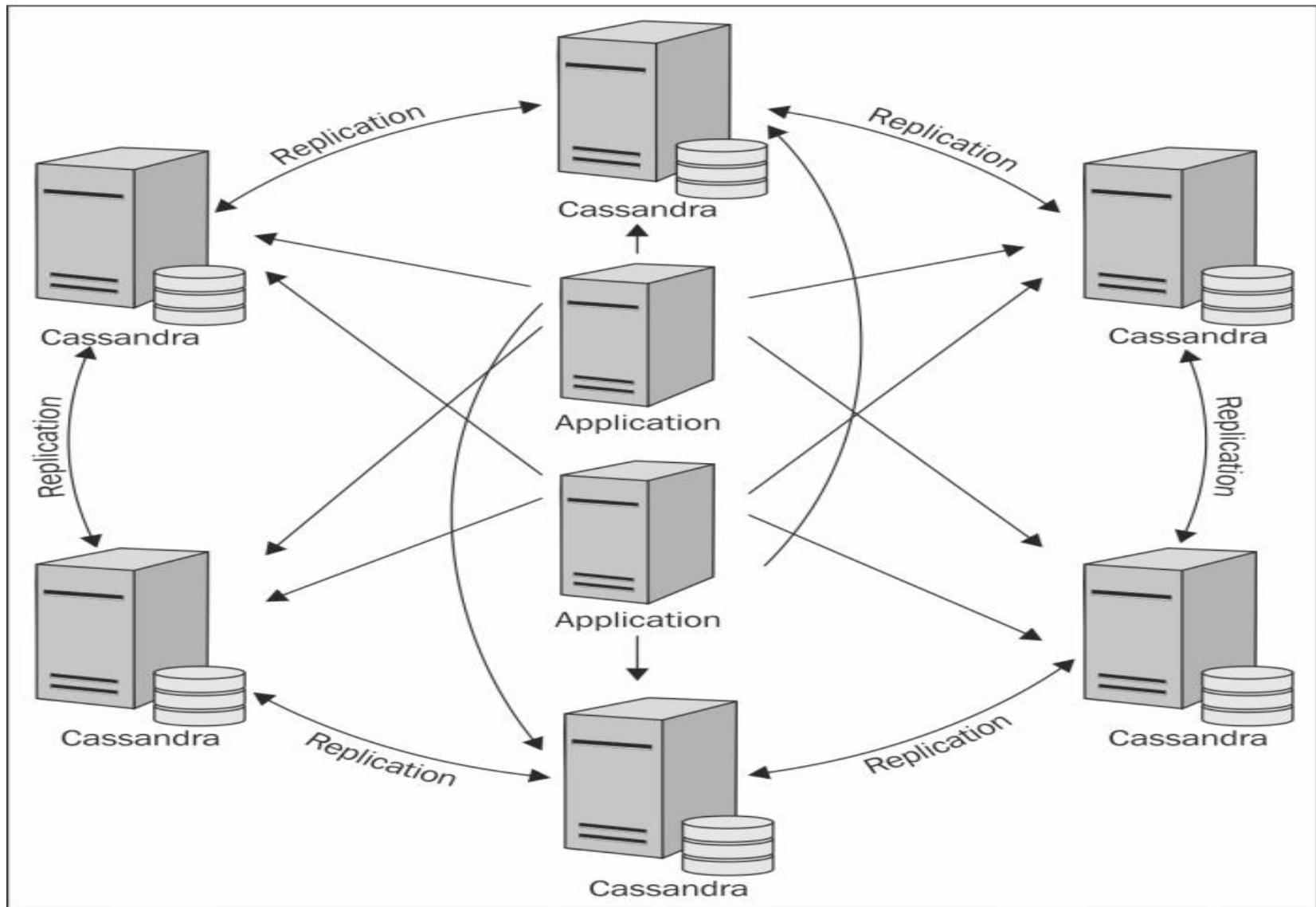
Gossip Protocol

Gossip Protocol in Cassandra is a *peer-to-peer communication protocol* in which nodes can choose among themselves with whom they want to exchange their state information

The nodes exchange information about themselves and about the other nodes that they have gossiped about, so all nodes quickly learn about all other nodes in the cluster

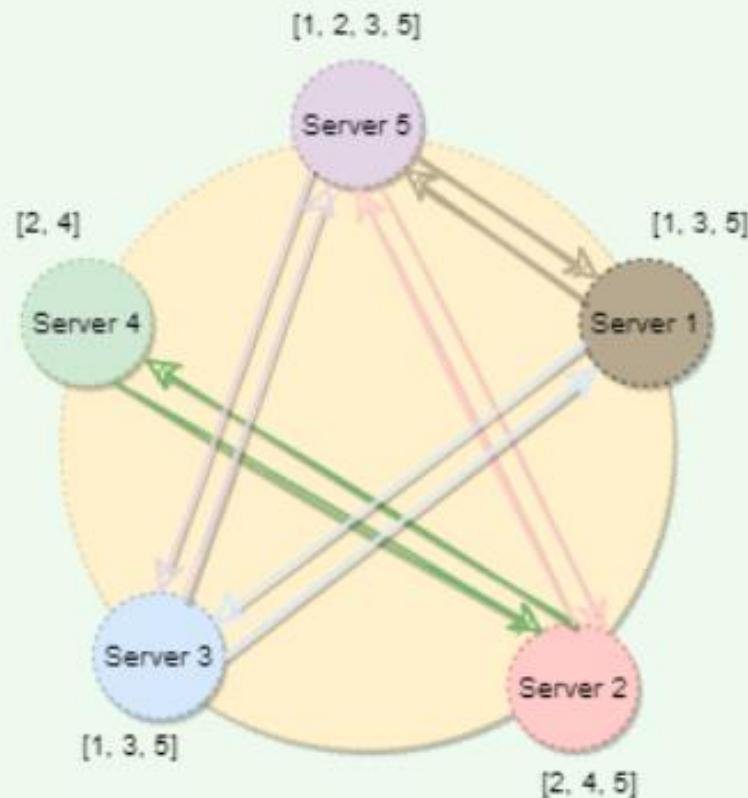


Gossip Protocol



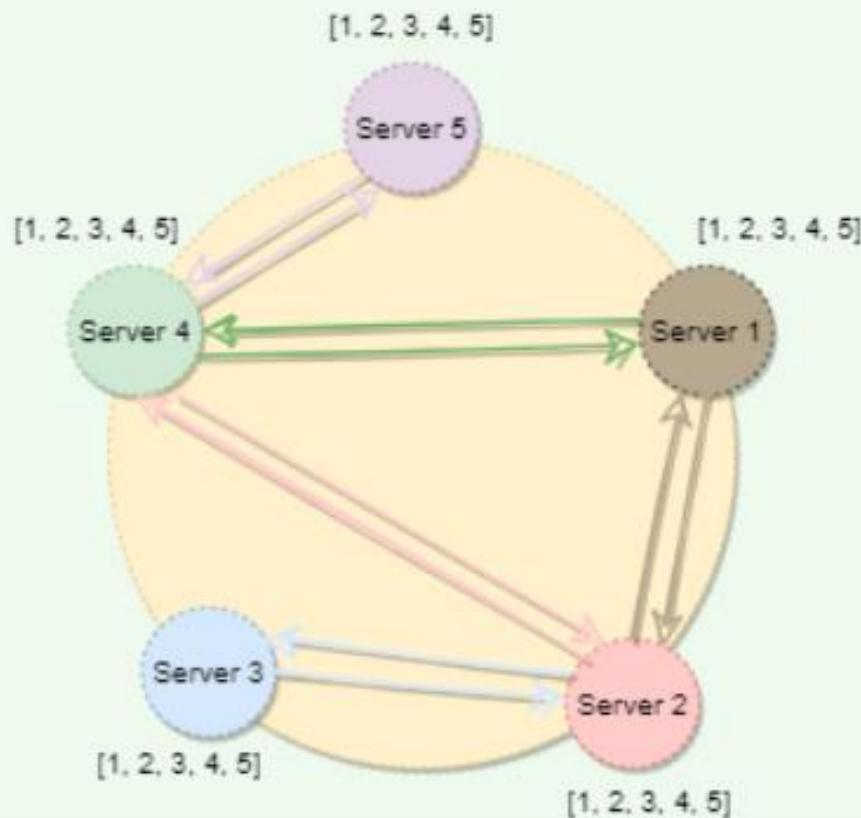
Gossip Protocol

Every second each server exchanges information with one randomly selected server



Gossip Protocol

Every second each server exchanges information about all the servers it knows about



Exchanging state information

Gossip Protocol

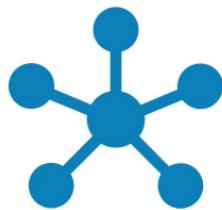
- The modern distributed systems use this peer-to-peer gossip protocol to make sure that the information is disseminated to all the members in the network.
- Gossip protocol is referred to as Epidemic Protocol as it disseminates or spreads the data the same way as an epidemic spreads a virus in a biological habitat.
- There are 3 main types of gossip protocol:
- Dissemination Protocols :
 - These protocols are also referred to as rumor-mongering protocols because they use gossip to spread information throughout the network, they flood the members of the network with gossips in a way that produces the worst-case load.

Gossip Protocol

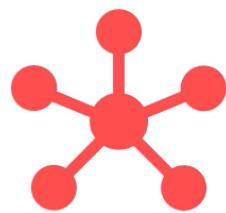
- Anti-Entropy Protocols :
 - These are used to repair the replicated data by comparing them and modifying the comparisons.
- Protocols that compute aggregates :
 - These protocols work by or compute an aggregate of the network by sampling information at the nodes and they combine the values to acquire a system-wide value – the largest value for some measurement nodes are making, smallest, etc.

Gossip Protocol Operation

Consider the following Nodes in a Cluster:



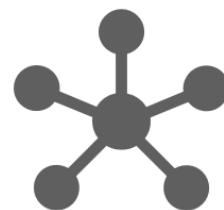
Node A



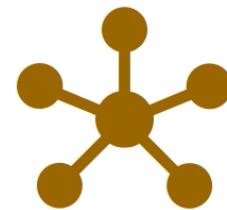
Node B



Node C



Node D

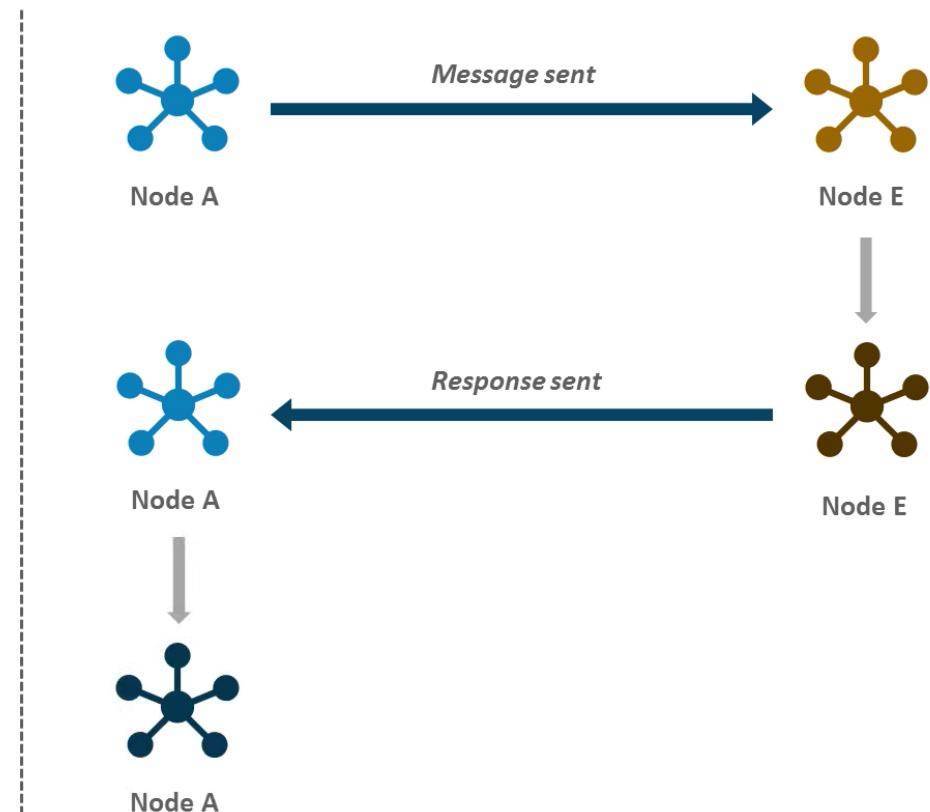


Node E

Each node has some data associated with it and *periodically gossips* this data with another node

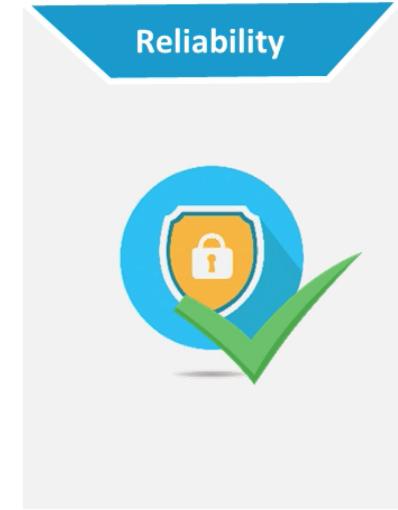
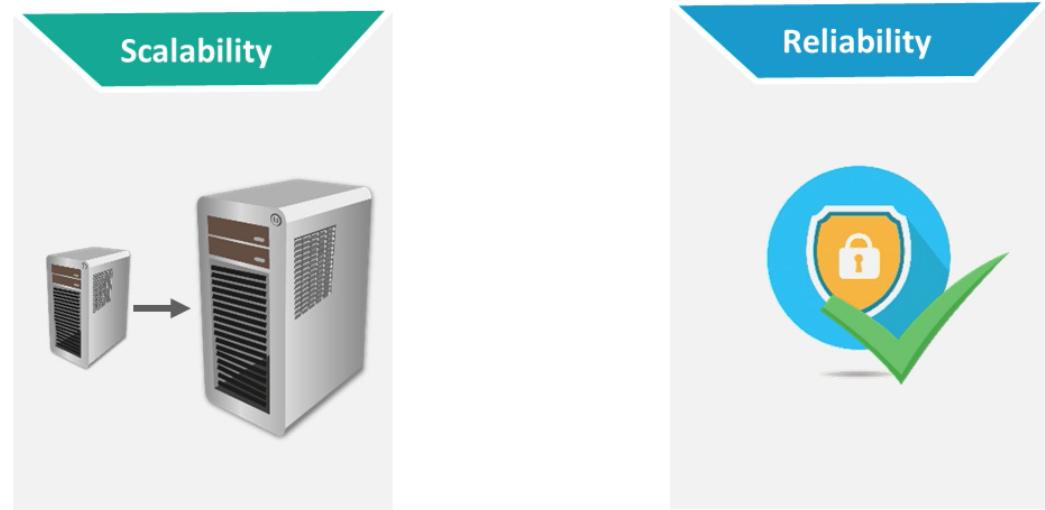
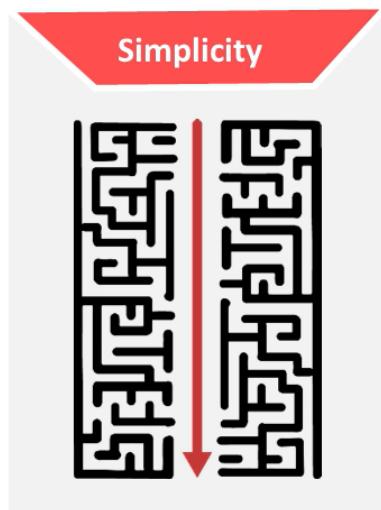
Gossip Protocol Operation

- 1 Node A randomly selects a *Node E* from a list of nodes known to it
- 2 A sends a *message* to E containing the data from A
- 3 E updates its data set with the received information
- 4 E sends back a *response* to A containing its data
- 5 A updates its data set with the received information

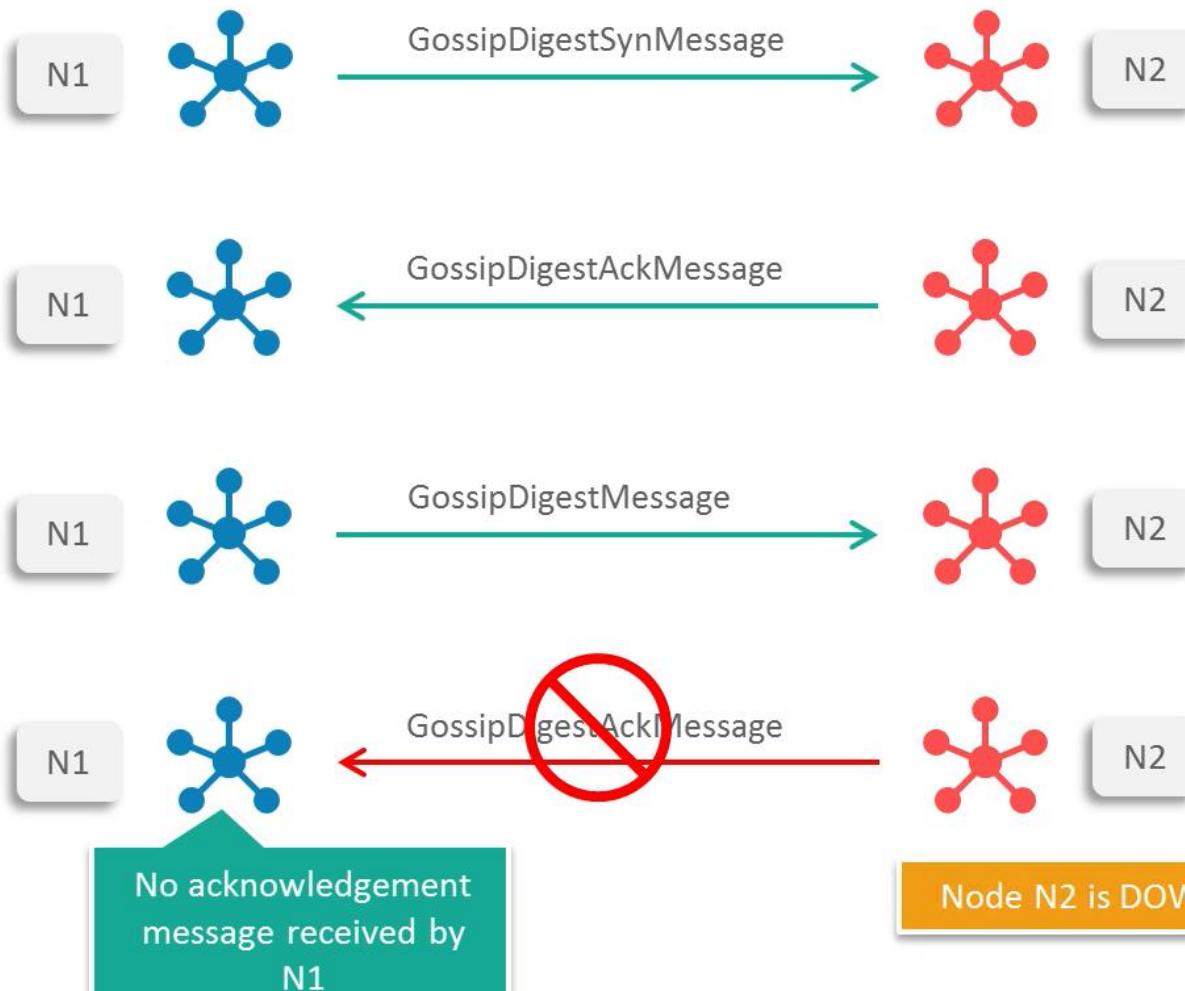


Gossip Protocol Features

Gossip Protocol is becoming increasingly popular in *distributed application* mainly because of these 3 features:



Gossip Protocol Failure Detection

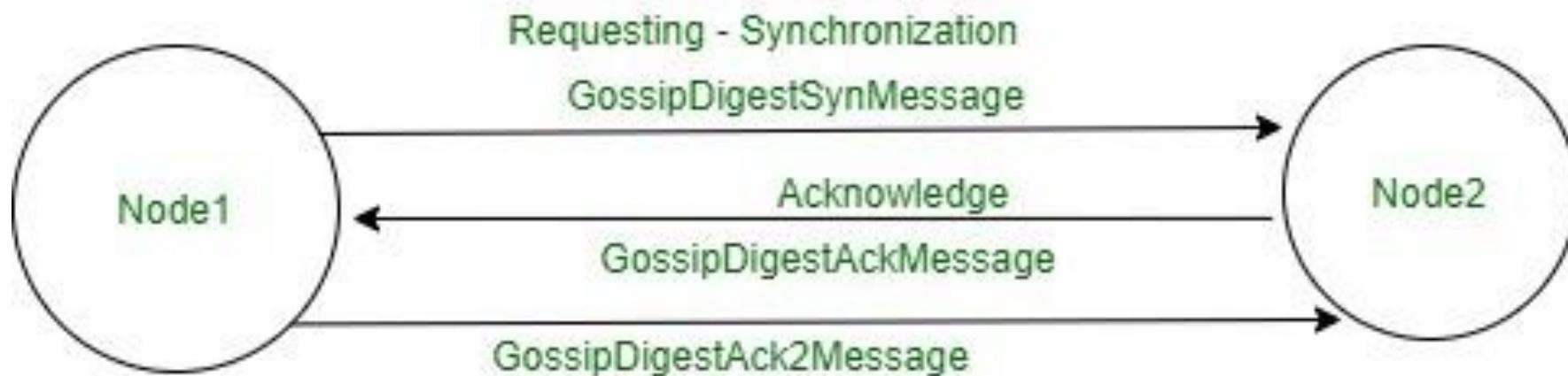




Gossip Protocol Message

- Cassandra Gossip coordinates the following information among the nodes:
 - Node liveness information
 - Token ownership
 - Cluster membership
 - Table schema
 - Infrastructure info, eg. IP, rack, dc, etc.

TCP Three-Way Handshake Process



3 way communication using Gossip protocol



TCP 3-way handshake protocol

It is most likely a TCP 3-way handshake protocol.

1. SYN -

In the synchronization part, it will send the GossipDigestSynMessage request to receiving node in a cluster, and receiving node will get this request and will respond accordingly.

2. ACK -

After requesting synchronization and once receiving node will get the request then it will acknowledge the request and will send the GossipDigestAckMessage.

3. ACK2 -

Finally, the initial node will get acknowledge then it will send GossipDigestAck2Message to the receiving node to acknowledge the acknowledgment response from receiving node.



TCP 3-way handshake protocol

- Initially, To receive endpoint state information, it will be done when Cassandra starts up, it registers itself with the Gossiper.
- After that, Periodically, typically once per second then Gossiper will choose the random node in a ring to start a Gossip session.
- After that, Initiating node will send the request `GossipDigestSynMessage` to the receiving node which simply means that it is requesting a synchronization.
- After that, when receiving node will get the request then it will acknowledge the request with `GossipDigestAckMessage` message to acknowledge the request.
- Finally, when Initiating node will get the acknowledgment from receiving node then again it will send the acknowledgment with `GossipDigestAck2Message`.



Gossip Protocol Failure Detection

- Failure detection is a method for locally determining from gossip state and history if another node in the system is down or has come back up.
- Cassandra uses this information to avoid routing client requests to unreachable nodes whenever possible. (Cassandra can also avoid routing requests to nodes that are alive, but performing poorly, through the dynamic snitch.)
- The gossip process tracks state from other nodes both directly (nodes gossiping directly to it) and indirectly (nodes communicated about secondhand, third-hand, and so on).



Gossip Protocol Failure Detection

- Rather than have a fixed threshold for marking failing nodes, Cassandra uses an accrual detection mechanism to calculate a per-node threshold that considers network performance, workload, and historical conditions.
- During gossip exchanges, every node maintains a sliding window of inter-arrival times of gossip messages from other nodes in the cluster.



Gossip Protocol Failure Detection

- Configuring the `phi_convict_threshold` property adjusts the sensitivity of the failure detector.
- Lower values increase the likelihood that an unresponsive node will be marked as down, while higher values decrease the likelihood that transient failures causing node failure.
- Use the default value for most situations, but increase it to 10 or 12 for Amazon EC2 (due to frequently encountered network congestion).
- In unstable network environments (such as EC2 at times), raising the value to 10 or 12 helps prevent false failures. Values higher than 12 and lower than 5 are not recommended.



Gossip Protocol Failure Detection

- Node failures can result from various causes such as hardware failures and network outages.
- Node outages are often transient but can last for extended periods.
- Because a node outage rarely signifies a permanent departure from the cluster it does not automatically result in permanent removal of the node from the ring.
- Other nodes will periodically try to re-establish contact with failed nodes to see if they are back up.
- To permanently change a node's membership in a cluster, administrators must explicitly add or remove nodes from a Cassandra cluster using the nodetool utility.



Gossip Protocol Failure Detection

- When a node comes back online after an outage, it may have missed writes for the replica data it maintains.
- Repair mechanisms exist to recover missed data, such as hinted handoffs and manual repair with nodetool repair.
- The length of the outage will determine which repair mechanism is used to make the data consistent.

Partitioners

- A partitioner determines how data is distributed across the nodes in the cluster (including replicas).
- Basically, a partitioner is a function for deriving a token representing a row from its partition key, typically by hashing.
- Each row of data is then distributed across the cluster by the value of the token.

Cassandra Distributed Database

Cassandra distributes the data across the entire cluster

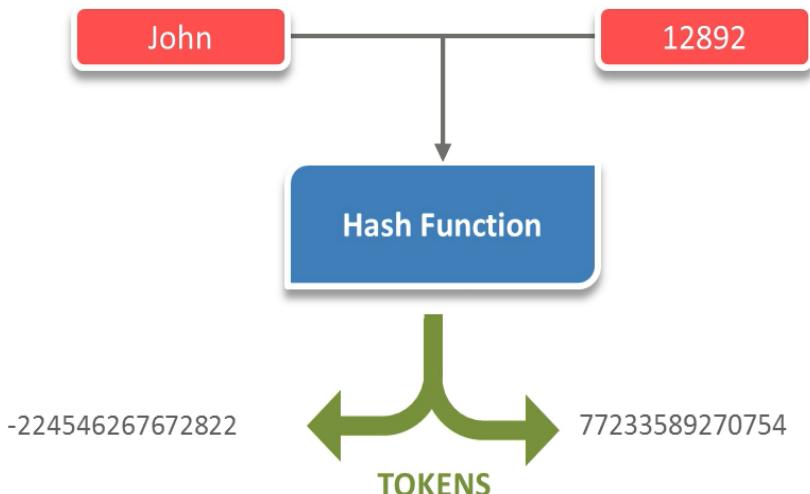
Data is stored on nodes in partitions, each identified by a partition Key and distributed across the cluster by value of token

Partition: It is *a hash function located on each node*

which hashes tokens from designated values in rows being added

It converts a *variable length input to a fixed length value.*

Token: *Integer value generated by a hashing algorithm,* identifying a *partition's location* within a cluster



Partitioners

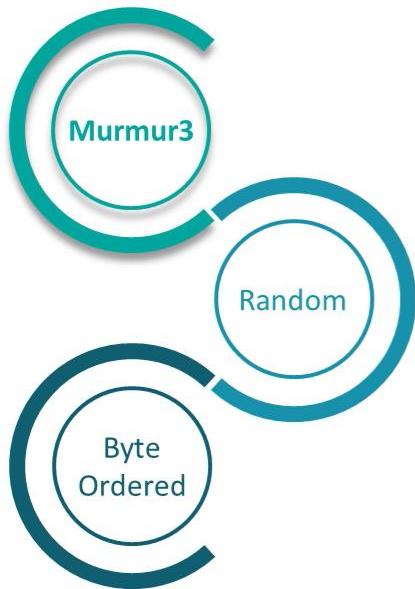
- Both the Murmur3Partitioner and RandomPartitioner use tokens to help assign equal portions of data to each node and evenly distribute data from all the tables throughout the ring or other grouping, such as a keyspace.
- This is true even if the tables use different partition keys, such as usernames or timestamps.
- Moreover, the read and write requests to the cluster are also evenly distributed and load balancing is simplified because each part of the hash range receives an equal number of rows on average



Partitioners with the TOKEN function in Cassandra

- Partitioners in CQL :
- 1. Murmur3partitioner
- 2. RandomPartitioner
- 3. ByteOrderedPartitioner

Types of Partitioners



Murmur3Partitioner is the *default partitioner*

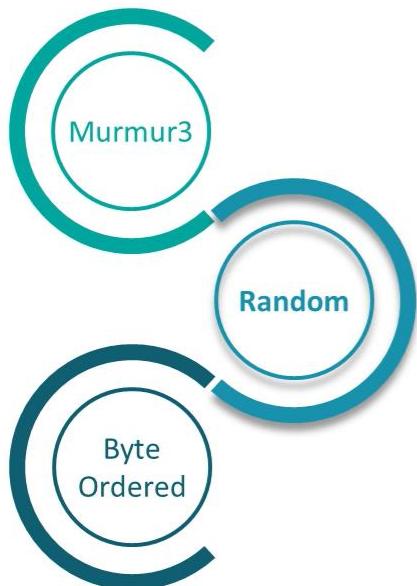
Improved and faster than RandomPartitioner

Uniformly distributes data based on *MurmurHash* function

64 bit hash value partition key

Range : -2^{63} to $2^{63}-1$

Types of Partitioners



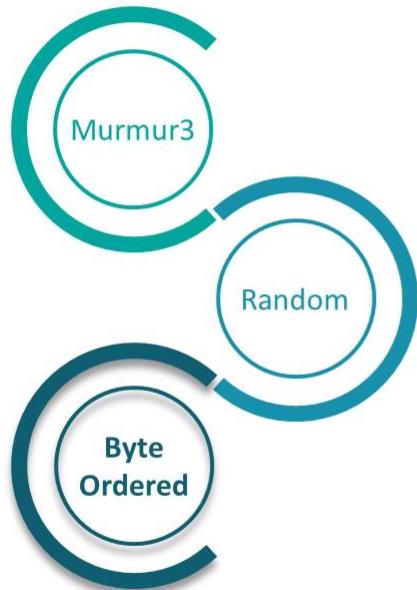
RandomPartitioner was the default partitioner *prior to Cassandra 1.2*

It is used with vnodes

Uniformly Distribution : MD5 hash values

Range: 0 to $2^{127}-1$

Types of Partitioners



ByteOrderedPartitioner is used for ordered partitioning

It *orders rows lexically by key bytes*

Using the ordered partitioner allows ordered scans by primary key

This means *we can scan rows as though we were moving a cursor through a traditional index*

Disadvantages : Byte Ordered

DIFFICULT LOAD BALANCING

- More *administrative overhead* is required to load balance the cluster
- It requires administrators to *manually calculate partition ranges* based on partition key distribution



SEQUENTIAL WRITES CAN CAUSE HOT SPOTS

- In case of write or update of a sequential block of rows, the writes are not be distributed across the cluster *they all go to one node*
- This is frequently a problem for applications *dealing with timestamped data*

UNEVEN LOAD BALANCING FOR MULTIPLE TABLES

- Multiple Tables implies Different Row Keys
- An ordered partitioner that is balanced for one table may cause hot spots and uneven distribution for another table in the same cluster





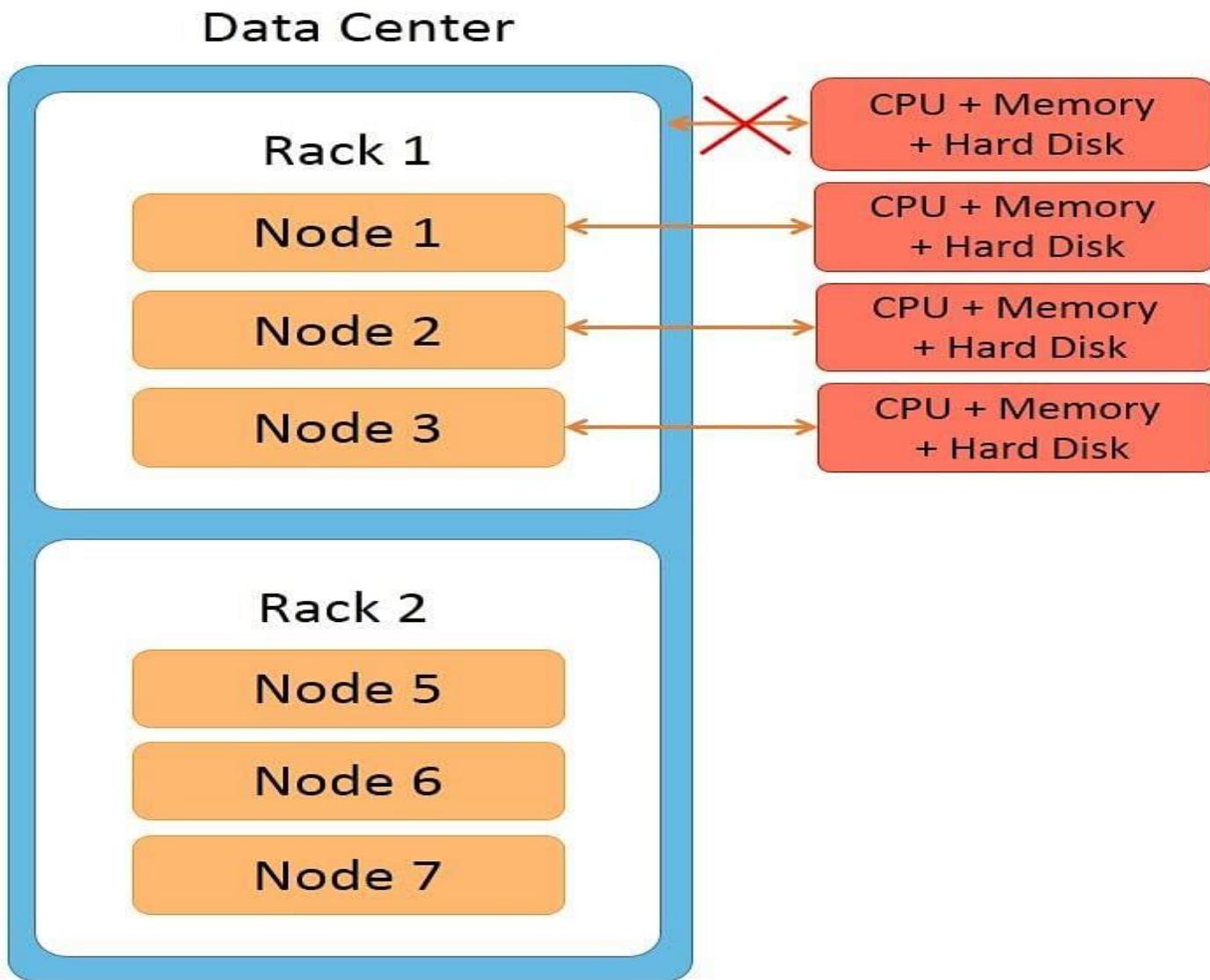
Type of Partitioner

The type of Partitioner to be used can be set in the *cassandra.yaml* file.

```
# The partitioner is responsible for distributing groups of rows (by
# partition key) across nodes in the cluster. You should leave this
# alone for new clusters. The partitioner can NOT be changed without
# reloading all data, so when upgrading you should set this to the
# same partitioner you were already using.
#
# Besides Murmur3Partitioner, partitioners included for backwards
# compatibility include RandomPartitioner, ByteOrderedPartitioner, and
# OrderPreservingPartitioner.
#
partitioner: org.apache.cassandra.dht.Murmur3Partitioner
```

partitioner: org.apache.cassandra.dht.Murmur3Partitioner

Rack





Features of Rack

- All machines in the rack are connected to the network switch of the rack
- The rack's network switch is connected to the cluster.
- All machines on the rack have a common power supply. It is important to notice that a rack can fail due to two reasons: a network switch failure or a power supply failure.
- If a rack fails, none of the machines on the rack can be accessed. So it would seem as though all the nodes on the rack are down.



Network Topology

- Network topology refers to how the nodes, racks and data centers in a cluster are organized. You can specify a network topology for your cluster as follows:
 - Specify in the Cassandra-topology.properties file.
 - Your data centers and racks can be specified for each node in the cluster.
 - Specify <ip-address>=<data center>:<rack name>.
 - For unknown nodes, a default can be specified.
 - You can also specify the hostname of the node instead of an IP address.



Network Topology

```
# Cassandra Node IP=Data Center:Rack  
192.168.1.100=DC1:RAC1  
192.168.2.200=DC2:RAC2  
  
10.20.114.10=DC2:RAC1  
10.20.114.11=DC2:RAC1  
  
# default for unknown nodes  
Default=DC1:rack1
```



Snitch

- A snitch determines *which datacenters and racks, nodes belong to*
- They *inform Cassandra about the network topology* and allows Cassandra to distribute replicas
- Specifically, the Replication strategy *places the replicas based* on the information provided *by the new snitch*

```
# You can use a custom Snitch by setting this to the full class name  
# of the snitch, which will be assumed to be on your classpath.  
endpoint_snitch: SimpleSnitch
```

Types of Snitches

Dynamic snitching

Monitors the performance of reads from the various replicas and chooses the best replica based on this history

SimpleSnitch

The SimpleSnitch is used only for single-datacenter deployments.

RackInferringSnitch

Determines the location of nodes by rack and datacenter corresponding to the IP addresses.

Ec2Snitch

Use the Ec2Snitch with Amazon EC2 in a single region

PropertyFileSnitch

Determines the location of nodes by rack and datacenter

GossipingPropertyFile

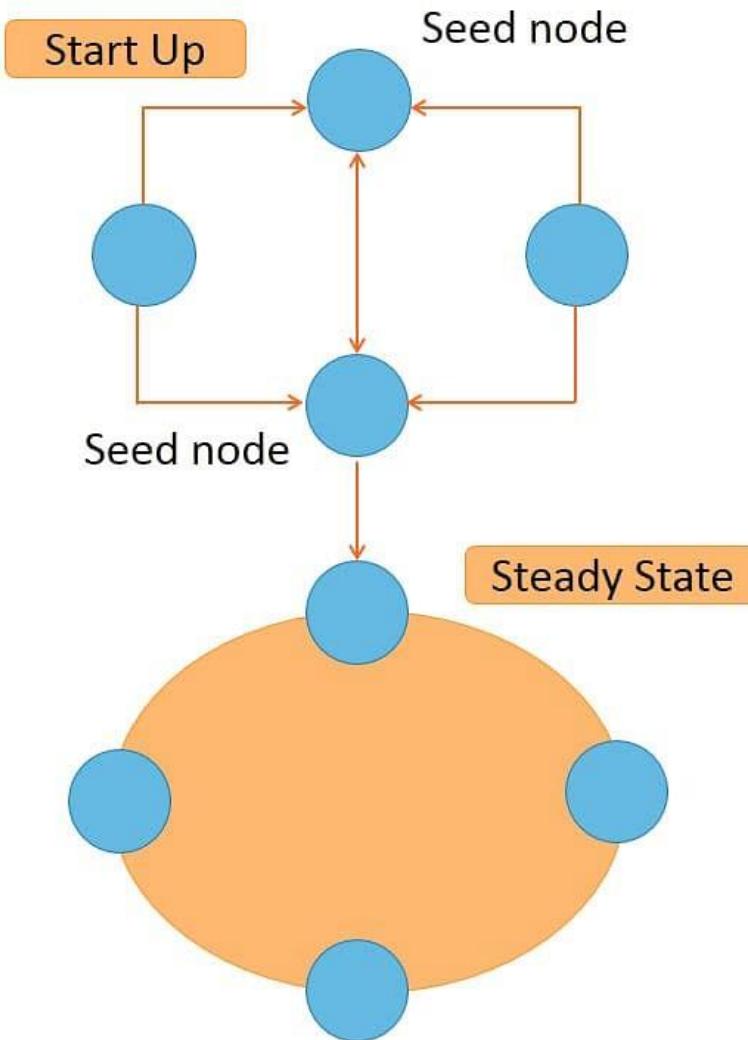
Automatically updates all nodes using gossip when adding new nodes and is recommended for production



Seed Nodes

- Seed nodes are used to bootstrap the gossip protocol.
The features of seed nodes are:
 - They are specified in the configuration file Cassandra.yaml.
 - Seed nodes are used for bootstrapping the gossip protocol when a node is started or restarted.
 - They are used to achieve a steady state where each node is connected to every other node but are not required during the steady state.

Seed Nodes

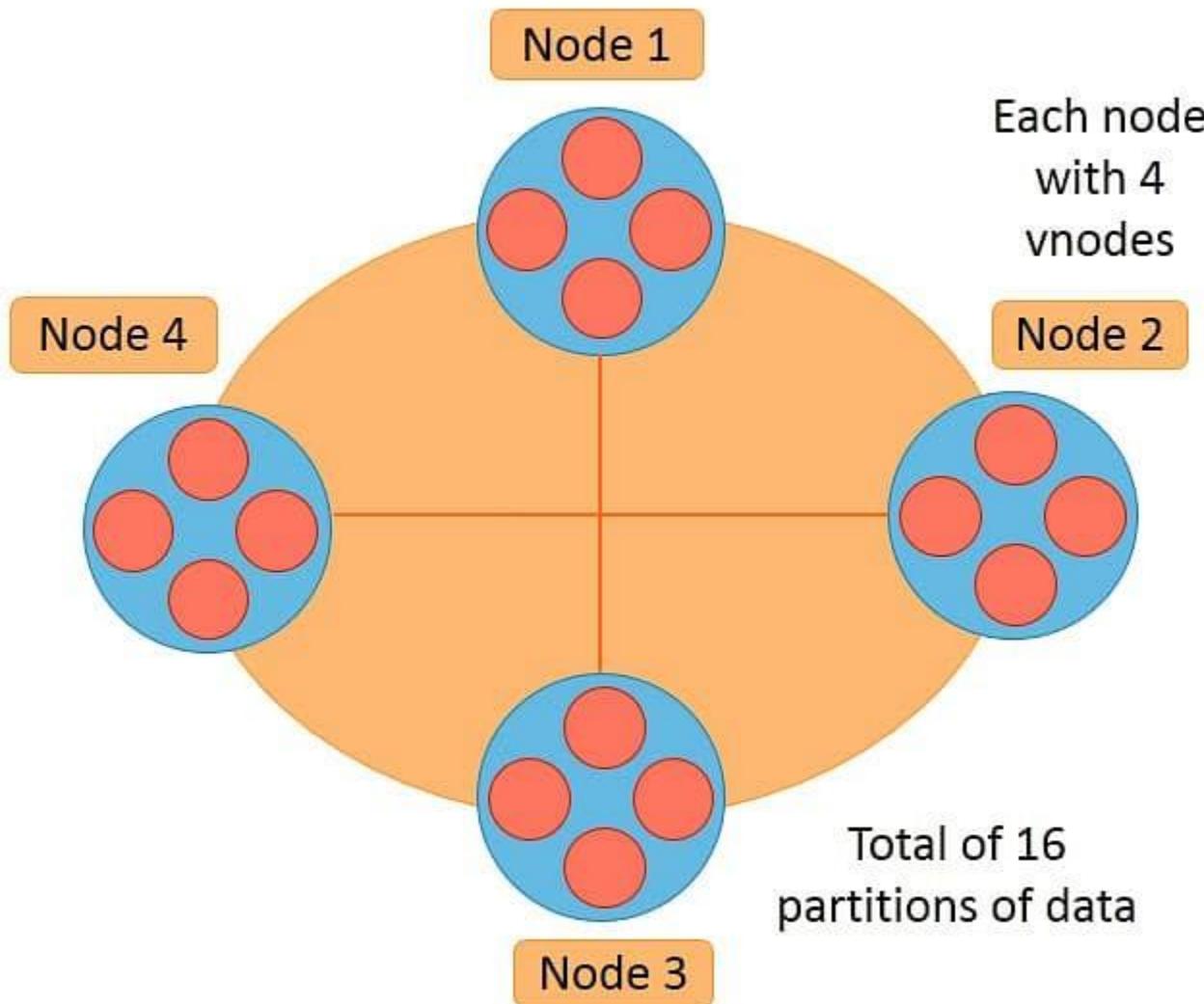




Virtual Nodes

- Virtual nodes in a Cassandra cluster are also called vnodes.
- Vnodes can be defined for each physical node in the cluster.
- Each node in the ring can hold multiple virtual nodes.
- By default, each node has 256 virtual nodes.
- Virtual nodes help achieve finer granularity in the partitioning of data, and data gets partitioned into each virtual node using the hash value of the key.
- On adding a new node to the cluster, the virtual nodes on it get equal portions of the existing data.
- So, there is no need to separately balance the data by running a balancer.

Virtual Nodes





Virtual Nodes

- Each physical node in the cluster has four virtual nodes. So there are 16 vnodes in the cluster.
- If 32TB of data is stored on the cluster, each vnode will get 2TB of data to store.
- If another physical node with 4 virtual nodes is added to the cluster, the data will be distributed to 20 vnodes in total such that each vnode will now have 1.6 TB of data.



Token Generator

- The token generator is used in Cassandra versions earlier than version 1.2 to assign a token to each node in the cluster.
- In these versions, there was no concept of virtual nodes and only physical nodes were considered for distribution of data.
- The token generator tool is used to generate a token for each node in the cluster based on the data centers and number of nodes in each data center.
- A token in Cassandra is a 127-bit integer assigned to a node.
- Starting from version 1.2 of Cassandra, vnodes are also assigned tokens and this assignment is done automatically so that the use of the token generator tool is not required.



Token Generator

- A token generator is an interactive tool which generates tokens for the topology specified.
- Let us now look at an example in which the token generator is run for a cluster with 2 data centers.
- Type token-generator on the command line to run the tool.
- A question is asked next: “How many data centers will participate in this cluster?” In the example, specify 2 as the number of data centers and press enter.
- Next, the question: “How many nodes are in data center number 1?” is asked. Type 5 and press enter.
- The next question is: “How many nodes are in data center number 2?” Type 4 and press enter.

Token Generator

token-generator

How many datacenters will participate in this cluster? 2

How many nodes are in datacenter #1? 5

How many nodes are in datacenter #2? 4

DC #1:

Node #1: 0

Node #2: 34028236692093846346337460743176821145

Node #3: 68056473384187692692674921486353642290

Node #4: 102084710076281539039012382229530463435

Node #5: 136112946768375385385349842972707284580

DC #2:

Node #1: 169417178424467235000914166253263322299

Node #2: 41811290829115311202148688466350243003

Node #3: 84346586694232619135070514395321269435

Node #4: 126881882559349927067992340324292295867

Problems

- Prior to version 1.2, each node used to own only one token, and it was responsible for *handling contiguous token range* in the cluster ring
- The mechanism was *slower* and *relatively harder to configure* in following scenarios:



1

While configuring a cluster, we need to *calculate and assign tokens* to each *node manually*

2

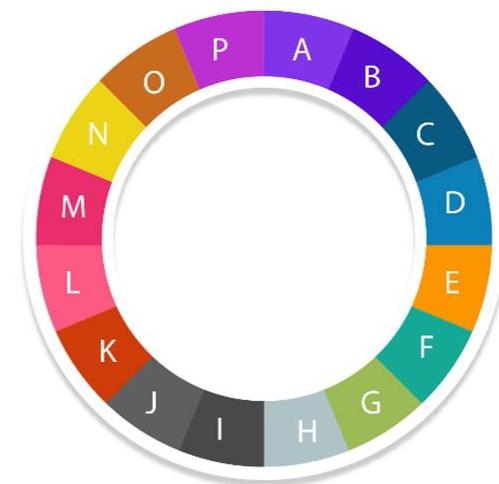
When a *node is added or removed*, we need to do a token calculation *again to rebalance the cluster*

3

When a *node is dead*, all the nodes responsible for same token range *do not participate in rebuilding process*

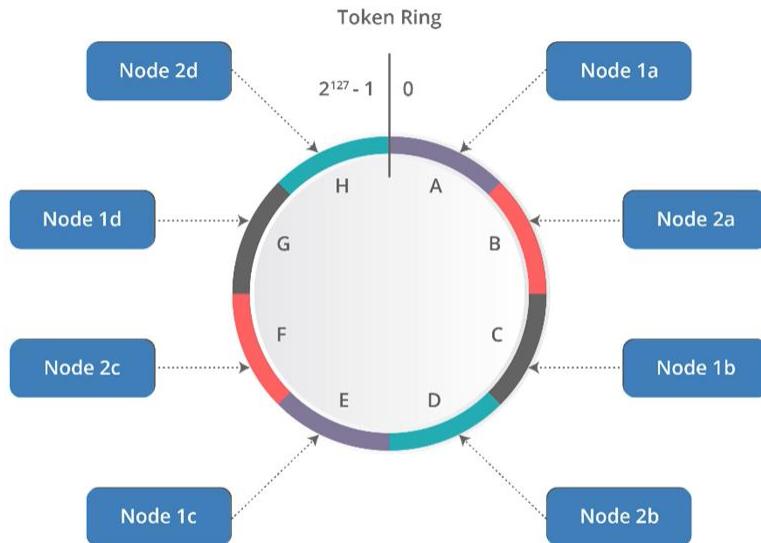
Solution Virtual Nodes

- Starting in version 1.2, *Cassandra allows many tokens per node*
- The new paradigm is called *Virtual Nodes (Vnodes)*
- *Vnodes* allow each node to own a *large number of small partition ranges*, distributed throughout the cluster
- *Vnodes* use *consistent hashing* to *distribute data* but using them doesn't require token generation and assignment



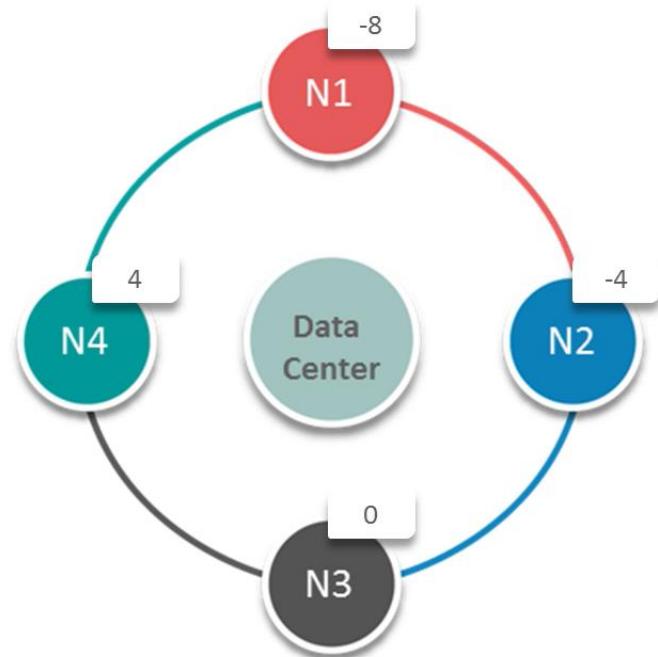
Tokens and Virtual Nodes

- A *token* determines a node's position in the ring. A token is a 64-bit integer ID used to identify each partition. This gives a possible range for tokens from -2^{63} to $2^{63}-1$
- A node claims ownership of the range of values *less than or equal to each token* and *greater than the token of the previous node*
- The Diagram shows notional ring layout including the nodes in a single data center
- This particular arrangement is structured such that consecutive token ranges are spread across nodes in different racks



Virtual Nodes Token Allocation

- The token range from -8 to 7 is distributed among four nodes as shown in the figure
- Here, node 1 has tokens with values greater than or equal to -8 and less than -4
- Node 2 has tokens for a range greater than or equal to -4 and less than 0
- Node 3 has tokens with value greater than or equal to 0 and less than 4
- Node 4 has a range greater than equal to 4 and less than 8



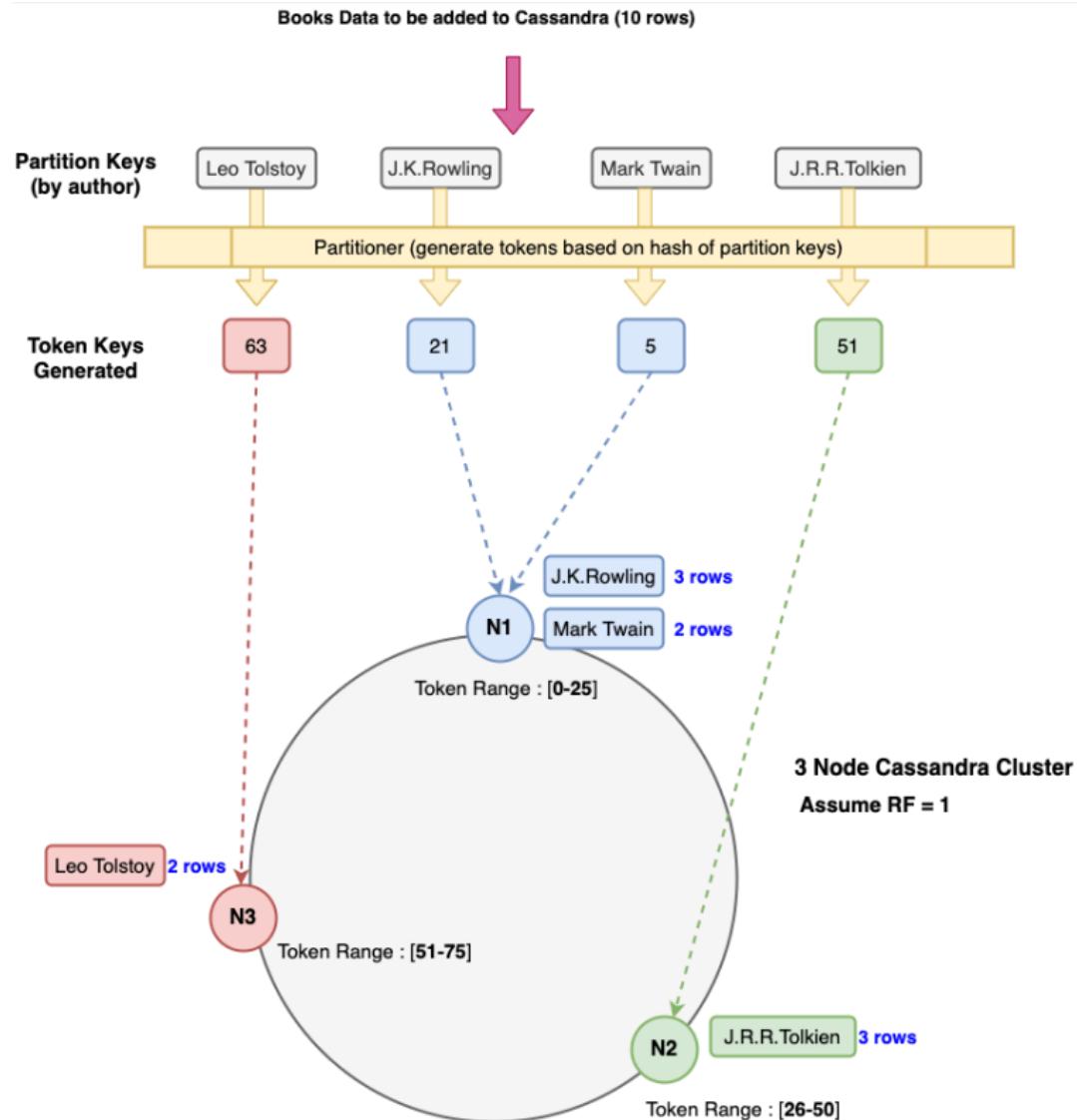


A Very Simple View of how data is partitioned

isbn	title	author	published	publisher	category
978-8175993426	Anna Karenina	Leo Tolstoy	2015	Fingerprint Publishing	History & Criticism
978-1408855669	Harry Porter and the Chamber of Secrets	J.K. Rowling	2014	Bloomsbury	Fantasy
978-1408855652	Harry Porter and the Philosopher's Stone	J.K. Rowling	2014	Bloomsbury	Fantasy
978-1408855676	Harry Porter and the Prisoner of Azkaban	J.K. Rowling	2014	Bloomsbury	Fantasy
978-0007488308	Lord of the Rings: The Fellowship of the Ring	J.R.R. Tolkien	2012	Harpercollins	Classic Fiction
978-0007488346	Lord of the Rings: The Return of the King	J.R.R. Tolkien	2012	Harpercollins	Classic Fiction
978-0007488322	Lord of the Rings: The Two Towers	J.R.R. Tolkien	2013	Harpercollins	Classic Fiction
978-1977066053	The Adventures of Tom Sawyer	Mark Twain	2012	Amazon	Drama & Plays
978-1544712048	The Mysterious Stranger	Mark Twain	2012	Amazon	Drama & Plays
978-8175992832	War and Peace	Leo Tolstoy	2014	Fingerprint Publishing	Classic Fiction

A Very Simple View of how data is partitioned

- We want to store this data in a 3 node Cassandra cluster
- Let's assume our Partitioning Key for this books table is author column





A Very Simple View of how data is partitioned

- Cassandra cluster has 3 nodes, with replication factor of 1 – which means no replication of data (just assume this for sake of simplicity)
- Each node can handle a range of tokens
 - Node N1 – tokens 0 – 25
 - Node N2 – tokens 25 – 50
 - Node N3 – tokens 51 – 75



A Very Simple View of how data is partitioned

- Now when we try to save this 10 rows of data in Cassandra – it first fetches the partition key for each row, which is the author column
- A consistent hashing algorithm is applied to the value in author column to generate the token. e.g.
 - Leo Tolstoy -> Token = 63
 - J.K. Rowling -> Token = 21
 - Mark Twain -> Token = 5
 - J.R.R. Tolkien -> Token = 51



A Very Simple View of how data is partitioned

- For each row of data, the partition key is used to generate the token. Then based on the token range of nodes, the data is stored on respective nodes. e.g.
- Node N1
 - J.K. Rowling – all 3 rows
 - Mark Twain – all 2 rows
- Node N2
 - J.R.R. Tolkien – all 3 rows
- Node N3
 - Leo Tolstoy – all 2 rows

How the choice of partition key impacts data storage



- Choice of partition key is very important in table design in Cassandra – the reason being it CANNOT be changed once a table is created.

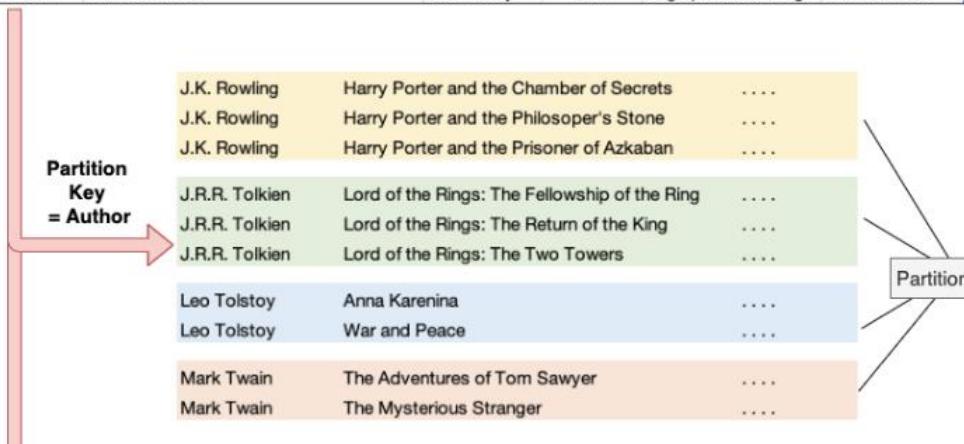
How the choice of partition key impacts data storage



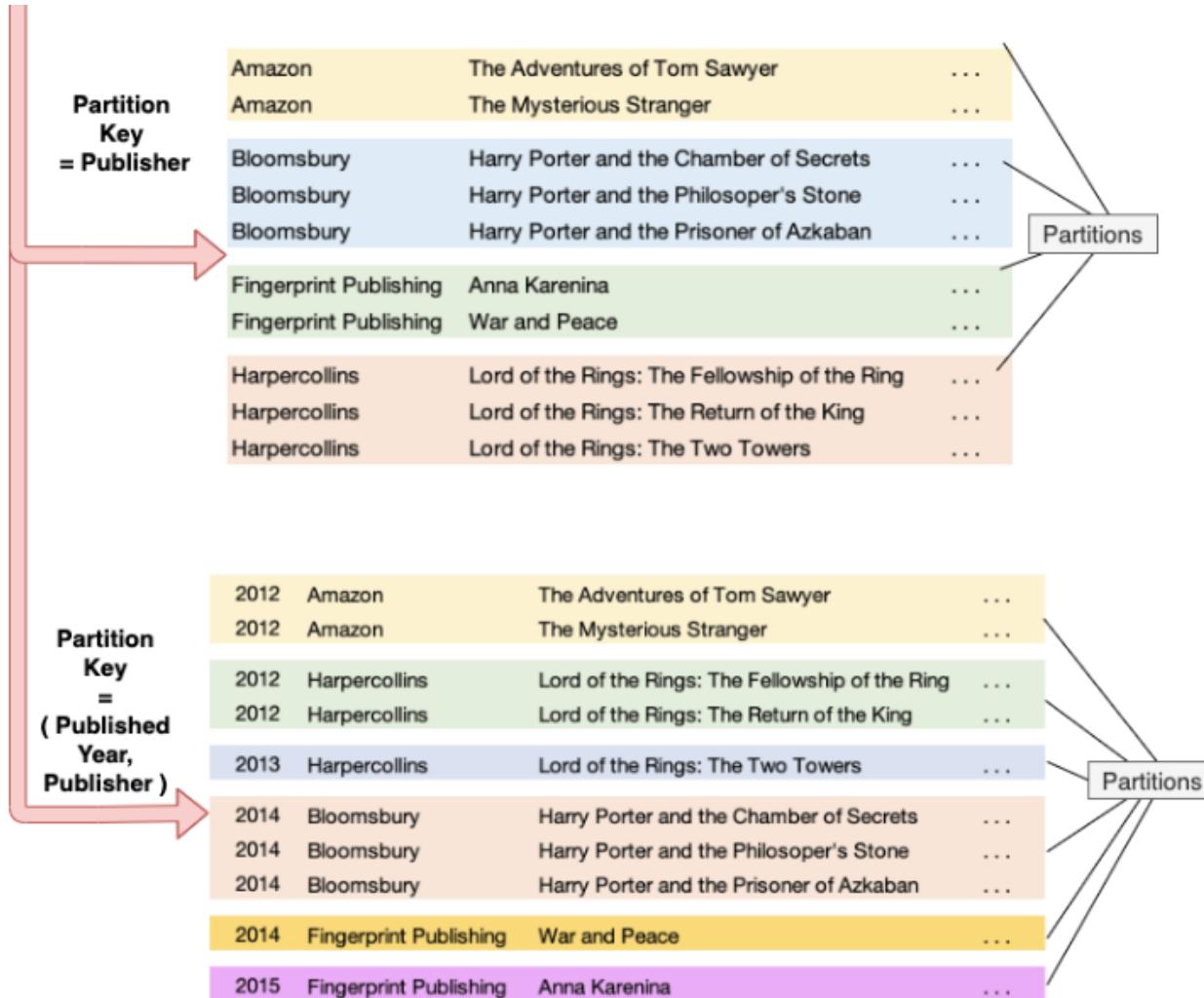
Choice of partition key is very important in table design in Cassandra – the reason being it **CANNOT** be changed once a table is created.

Let's see some examples of how data gets partitioned, based on the choice of partition key.

isbn	title	author	published	publisher	category
978-8175993426	Anna Karenina	Leo Tolstoy	2015	Fingerprint Publishing	History & Criticism
978-1408855669	Harry Porter and the Chamber of Secrets	J.K. Rowling	2014	Bloomsbury	Fantasy
978-1408855652	Harry Porter and the Philosopher's Stone	J.K. Rowling	2014	Bloomsbury	Fantasy
978-1408855676	Harry Porter and the Prisoner of Azkaban	J.K. Rowling	2014	Bloomsbury	Fantasy
978-0007488308	Lord of the Rings: The Fellowship of the Ring	J.R.R. Tolkien	2012	Harpercollins	Classic Fiction
978-0007488346	Lord of the Rings: The Return of the King	J.R.R. Tolkien	2012	Harpercollins	Classic Fiction
978-0007488322	Lord of the Rings: The Two Towers	J.R.R. Tolkien	2013	Harpercollins	Classic Fiction
978-1977066053	The Adventures of Tom Sawyer	Mark Twain	2012	Amazon	Drama & Plays
978-1544712048	The Mysterious Stranger	Mark Twain	2012	Amazon	Drama & Plays
978-8175992832	War and Peace	Leo Tolstoy	2014	Fingerprint Publishing	Classic Fiction



How the choice of partition key impacts data storage



How the choice of partition key impacts data storage



- partition key = isbn
 - each partition will have only 1 row, since isbn is unique
 - querying books will always require you to specify isbn as the primary query condition
- partition key = publisher
 - if the publisher is a quite popular one and publishes thousands of books every year, then with time this partition will keep on growing and become very huge.
- A simple solution would be to add published year to the partition key.



Partitions and tokens in a real cluster

- In the examples that we saw above, we used RF of 1 for simplicity and also showed token ranges in very low ranges (less than 100).
- In real production clusters
 - Replication Factor would generally be 3.
 - Tokens are generally in range of -2^{63} to $+2^{63} - 1$.
 - Each node will be assigned multiple token ranges via Vnodes

Partitions and tokens in a real cluster

$$\text{token}(n_1) = t_1$$

$$\text{token}(n_2) = t_2$$

$$\text{token}(n_3) = t_3$$

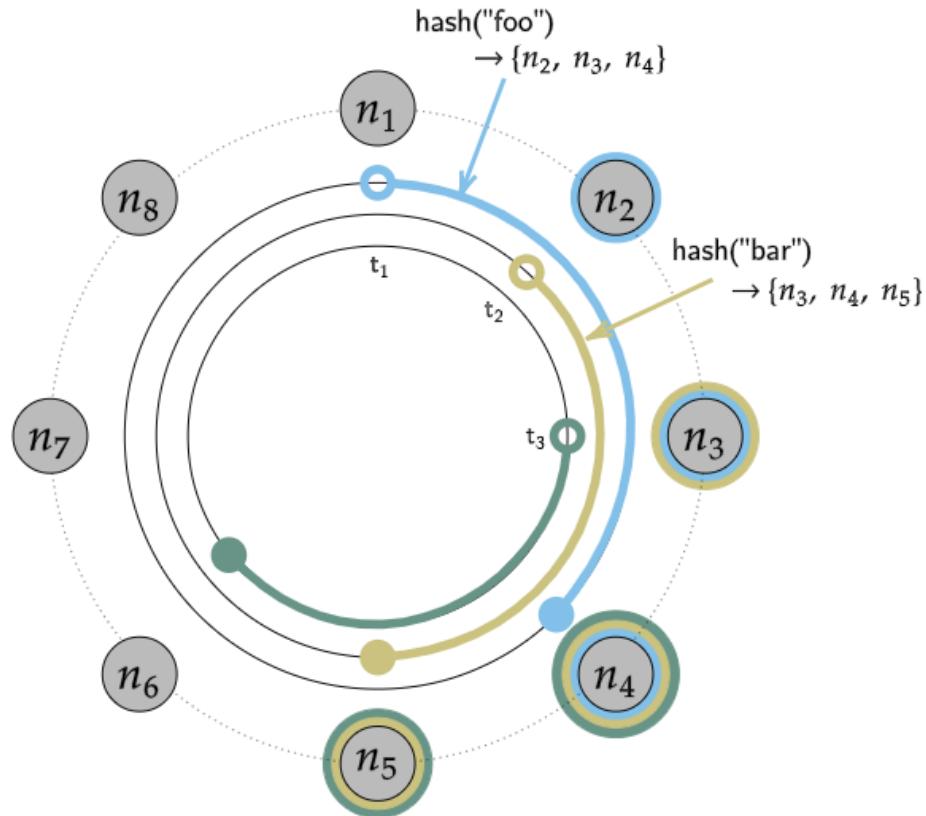
...

$$\text{range}(t_1, t_2] \rightarrow \{n_2, n_3, n_4\}$$

$$\text{range}(t_2, t_3] \rightarrow \{n_3, n_4, n_5\}$$

$$\text{range}(t_3, t_4] \rightarrow \{n_4, n_5, n_6\}$$

...





Partitions and tokens in a real cluster

- In above example we have a 8 node cluster, with a RF = 3.
- We first hash the key to generate the token.
- We “walk” the ring in a clockwise fashion until we encounter three distinct nodes, at which point we have found all the replicas of that key.
- e.g.
- For Key = foo
 - Hash is between t1 and t2
 - So replicas for this key, are next three nodes n2, n3, n4
- For Key = bar
 - Hash is between t2 and t3
 - So replicas for this key, are next three nodes n3, n4, n5



Cassandra.yaml File

```
cassandra.yaml
/usr/lib/apache-cassandra-3.11.0/conf
Save - x

# one logical cluster from joining another.
cluster_name: 'Test Cluster'

# This defines the number of tokens randomly assigned to this node on the ring
# The more tokens, relative to other nodes, the larger the proportion of data
# that this node will store. You probably want all nodes to have the same number
# of tokens assuming they have equal hardware capability.
#
# If you leave this unspecified, Cassandra will use the default of 1 token for
# legacy compatibility,
# and will use the initial_token as described below.
#
# Specifying initial_token will override this setting on the node's initial start,
# on subsequent starts, this setting will apply even if initial token is set.
#
# If you already have a cluster with 1 token per node, and wish to migrate to
# multiple tokens per node, see http://wiki.apache.org/cassandra/Operations
num_tokens: 256
```

YAML ▾ Tab Width: 8 ▾

Ln 31, Col 2 ▾

INS

How are Virtual Nodes Helpful

01

Token ranges are distributed. Hence, bootstrapping is faster

02

Token range calculation and assignment will be automated

03

Rebalancing a cluster is no longer necessary while adding or removing nodes

04

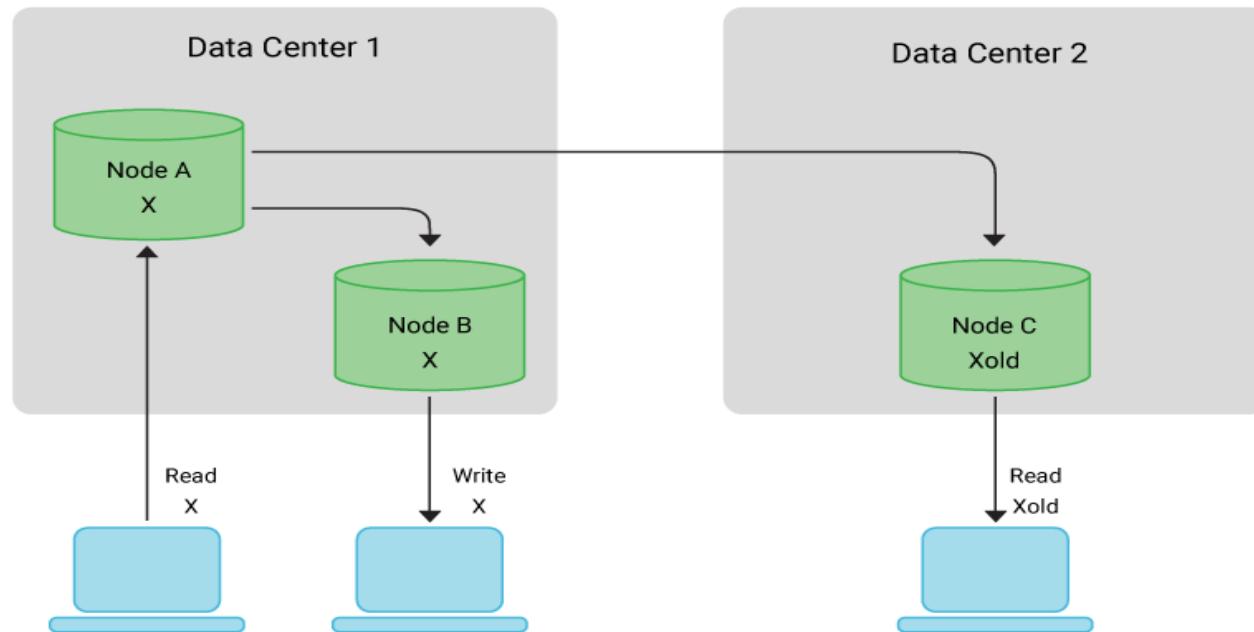
Rebuilding a dead node is faster as it involves every other node in the cluster

05

Improves the use of heterogeneous machines in a cluster

Eventual Consistency

- Eventual Consistency is a guarantee that when an update is made in a distributed database, that update will eventually be reflected in all nodes that store the data, resulting in the same response every time the data is queried.





Eventual Consistency

- Consistency refers to a database query returning the same data each time the same request is made.
- Strong consistency means the latest data is returned, but, due to internal consistency methods, it may result with higher latency or delay.
- With eventual consistency, results are less consistent early on, but they are provided much faster with low latency.
- Early results of eventual consistency data queries may not have the most recent updates because it takes time for updates to reach replicas across a database cluster.



Eventual Consistency

- A key benefit of an eventually consistent database is that it supports the high availability model of NoSQL.
- Eventually consistent databases prioritize availability over strong consistency.
- Eventual consistency in microservices can support an always-available API that must be responsive, even if the query results may occasionally be missing the latest commit.



What Are Eventual Consistency Examples?

- In AWS eventual consistency can refer to more than one eventual consistency database.
- DynamoDB eventual consistency and Cassandra eventual consistency both depend on a ring topology where data is spread to more than one peer database node.
- A good example is DynamoDB eventual consistency vs strong consistency with results written to majority of nodes, and eventually updating all nodes.
- If majority of nodes are consulted on the read process, the read will get either conflicting values, or the latest value.



What Are Eventual Consistency Examples?

- Redis eventual consistency offers another way how eventual consistency is achieved.
- Redis uses a master-slave topology for how to handle eventual consistency, relying on replication from the master node to the slave nodes.



What Are Eventual Consistency Examples?

- Elastic Search eventual consistency is handled similarly to Redis with a master node and plus a primary shard and replica shards.
- The Elastic Search eventual consistency pattern is also a hybrid with the methods of eventual consistency Cassandra and DynamoDB utilize, because Elastic Search also offers consistency parameters to indicate how many shards must be written or read for a successful operation.



Tuneable Consistency

- Tunable Consistency means that you can set the CL for each read and write request.
- So, Cassandra gives you a lot of control over how consistent your data is.
- We can allow some queries to be immediately consistent and other queries to be eventually consistent.
- That means, in our application, the data that requires **immediate consistency**, we can create queries accordingly and the data for which immediate consistency is not required, we can optimize for performance and choose eventual consistency.

Immediate Consistency

- Immediate Consistency is the ability of the database to always return the most recent.
- Immediate Consistency Formula
- $R + W > RF \Rightarrow$ immediate consistency
- where R = number of nodes your data is read from.
- W = number of nodes your data is written to.
- RF = replication factor, how many nodes have exact same copy of your data.

Immediate Consistency

- Example: if we have replication factor as 3, i.e. 3 nodes are going to have exact same copy of data, write CL as QUORUM, i.e. we are going to write to majority or 2 nodes.
- Read CL as QUORUM, i.e. we are going to read from majority or 2 nodes, then we can see from the formula that we are going to have immediate consistency.

Immediate Consistency

- Achieving Immediate Consistency across local Data Center
- Read optimized, set Write CL = ALL, Read CL = LOCAL_ONE
- Write optimized, set Write CL = LOCAL_ONE, Read CL = ALL
- Balanced, set Write CL = LOCAL_QUORUM, Read CL = LOCAL_QUORUM

Eventual Consistency

- Entire Cluster, set Write CL = ONE, Read CL = ONE
- Local Data Center, set Write CL = LOCAL_ONE, Read CL = LOCAL_ONE
- This configuration is performance optimized.
- As we know, data sync across all nodes is few milliseconds away, so we get great performance using this configuration.
- So, we give up few milliseconds of consistency here and get highest throughput, highest performance and highest availability.

Write Consistency Levels

The write consistency level determines “ *how many replica nodes must respond with a success acknowledgment in order for the write to be considered successful* ”

Consistency Level	Implication
ZERO	The write operation will return immediately to the client before the write is recorded; the write will happen asynchronously in a background thread, and there are no guarantees of success.
ANY	Ensure that the value is written to a minimum of one node, allowing hints to count as a write.
ONE	Ensure that the value is written to the commit log and memtable of at least one node before returning to the client.
QUORUM	Ensure that the write was received by at least a majority of replicas ((replication factor / 2)+ 1).
ALL	Ensure that the number of nodes specified by replication factor received the write before returning to the client. If even one replica is unresponsive to the write operation, fail the operation.

Key Elements: Write Path

1

Commit log

The *Commit log* is a crash-recovery mechanism that supports Cassandra's durability goals

2

MemTable

MemTable is an in-memory data structure that corresponds to a CQL table

3

SSTable

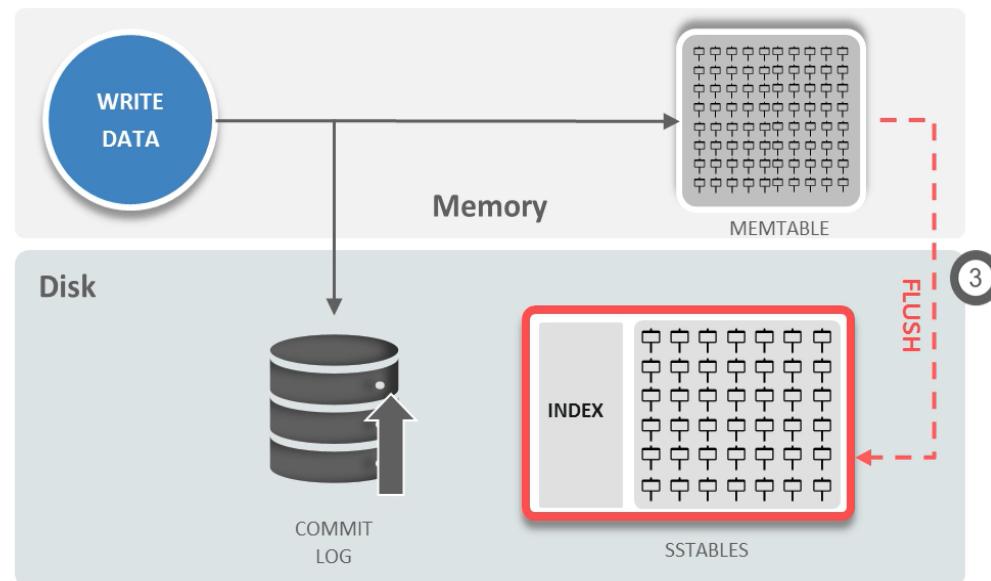
The contents of the memtable are *flushed to disk* in a *file called an SSTable*

Write Path Process

When *write request* comes to the node:

- 1 Firstly, it logs in the *Commit Log*
- 2 Data will be captured and stored in the *Mem-Table*
- 3 When mem-table is full, data is flushed to the *SSTable* data file

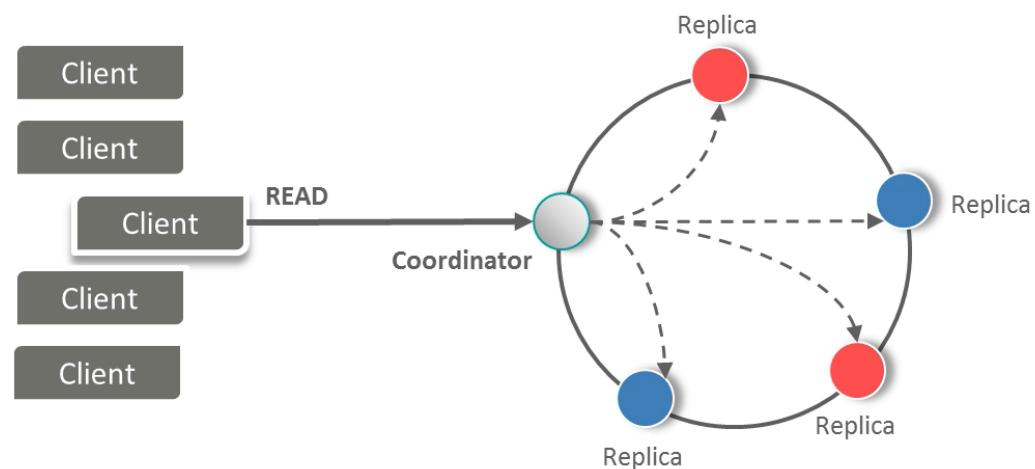
All writes are automatically partitioned and replicated throughout the cluster
 Cassandra periodically consolidates the *SSTables*, discarding unnecessary data



Read Data: Coordinator Node

Read Operation is easy, because clients can connect to any node in the cluster to perform reads

- If a client connects to a node *that doesn't have the data it's trying to read*, the node it's connected to will act as *coordinator node*
- When a client requests data from a coordinator node, it sends *the request to all replica nodes* responsible for owning the data
- The coordinator node compares the data and responds to the client with the most recent data returned by the replicas



Read Consistency Level

The number of *replicas contacted by a client read* request is determined by *the consistency level* specified by the client.

Consistency Level	Implication
ZERO	Unsupported. You cannot specify CL.ZERO for read operations because it doesn't make sense.
ANY	Unsupported. Use CL.ONE instead
ONE	Immediately return the record held by the first node that responds to the query. A background thread is created to check that record against the same record on other replicas. If any are out of date, a read repair is then performed to sync them all to the most recent value.
QUORUM	Query all nodes. Once a majority of replicas ((replication factor / 2) + 1) respond, return to the client the value with the most recent timestamp. Then, if necessary, perform a read repair in the background on all remaining replicas.
ALL	Query all nodes. Wait for all nodes to respond, and return to the client the record with the most recent timestamp. Then, if necessary, perform a read repair in the background. If any nodes fail to respond, fail the read operation.

Read Requests Types

There are three types of read requests that a Coordinator can send to a replica:

Direct Request



The coordinator node contacts one replica node

Digest Request



The coordinator sends the digest request to the number of replicas specified by the consistency level

The request checks whether the returned data is an updated data, then the coordinator sends a digest request to all remaining replicas

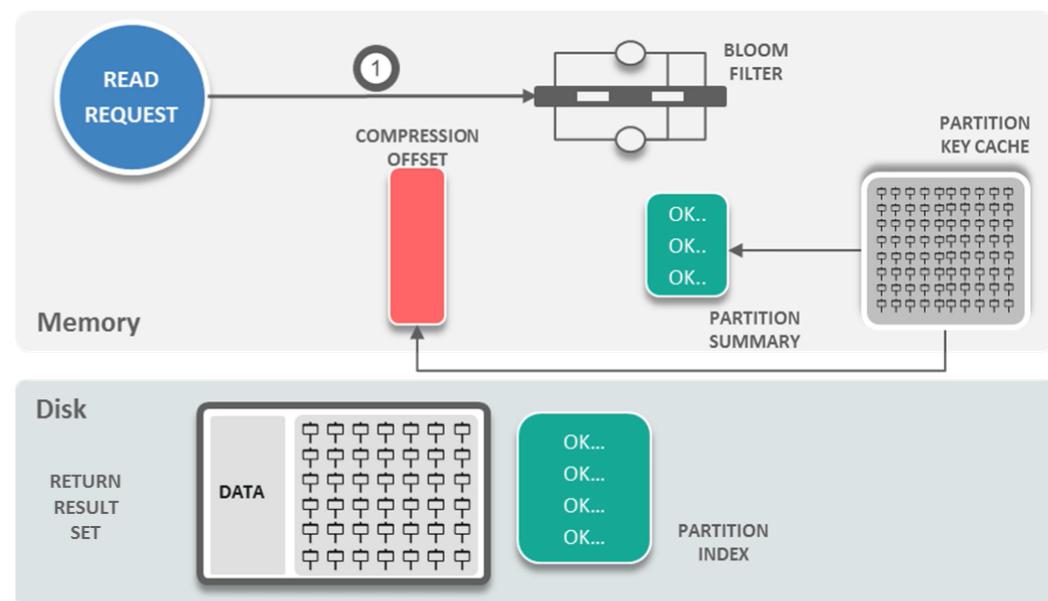
Read Repair Request



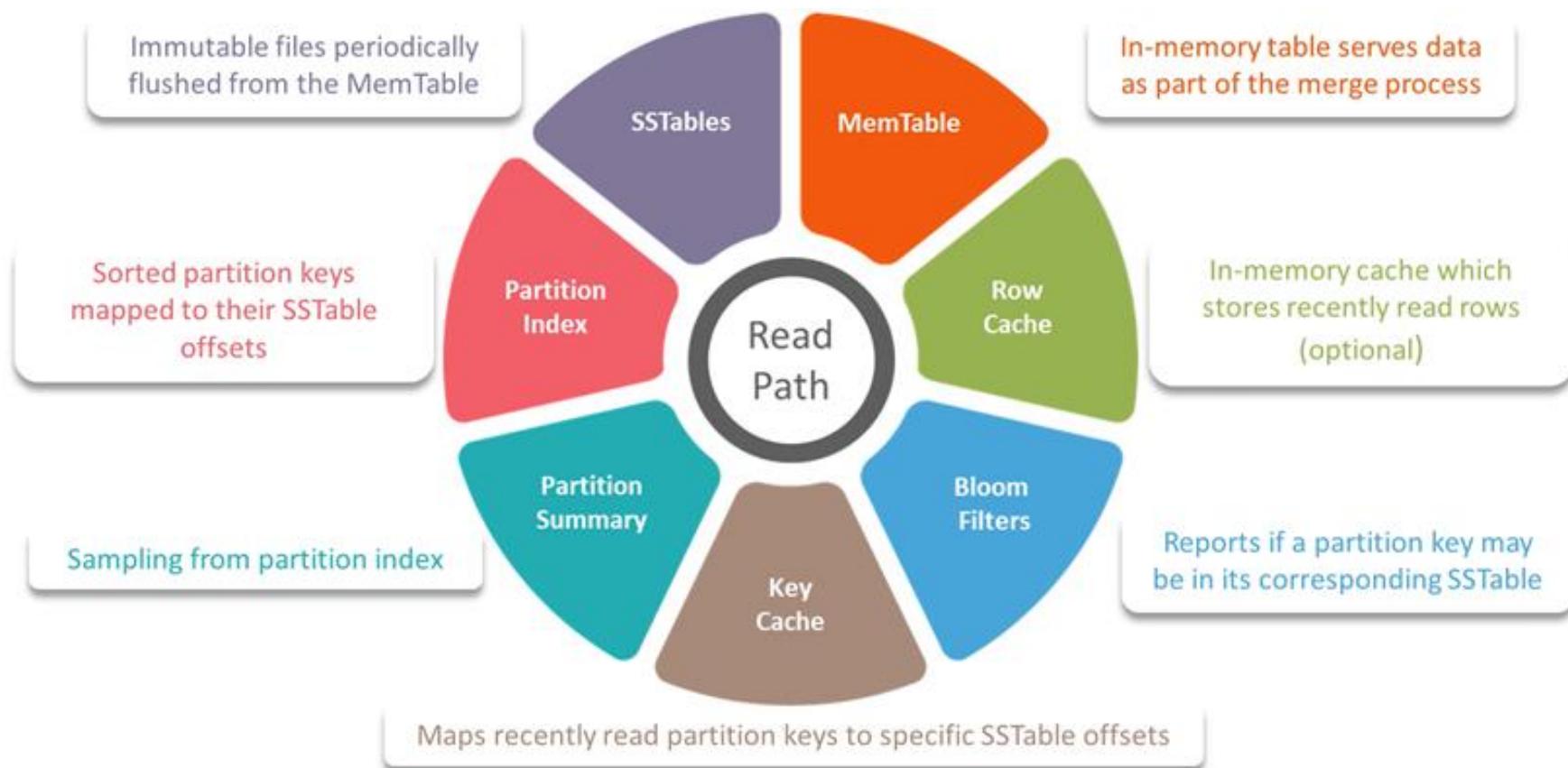
If any node gives out of date value, a background read repair request will update that data

Cassandra - Read

- Cassandra returns the most recent record among the nodes read for a given request
- Reading data from Cassandra involves a number of processes that can include various memory caches and other mechanisms designed to produce fast read response times

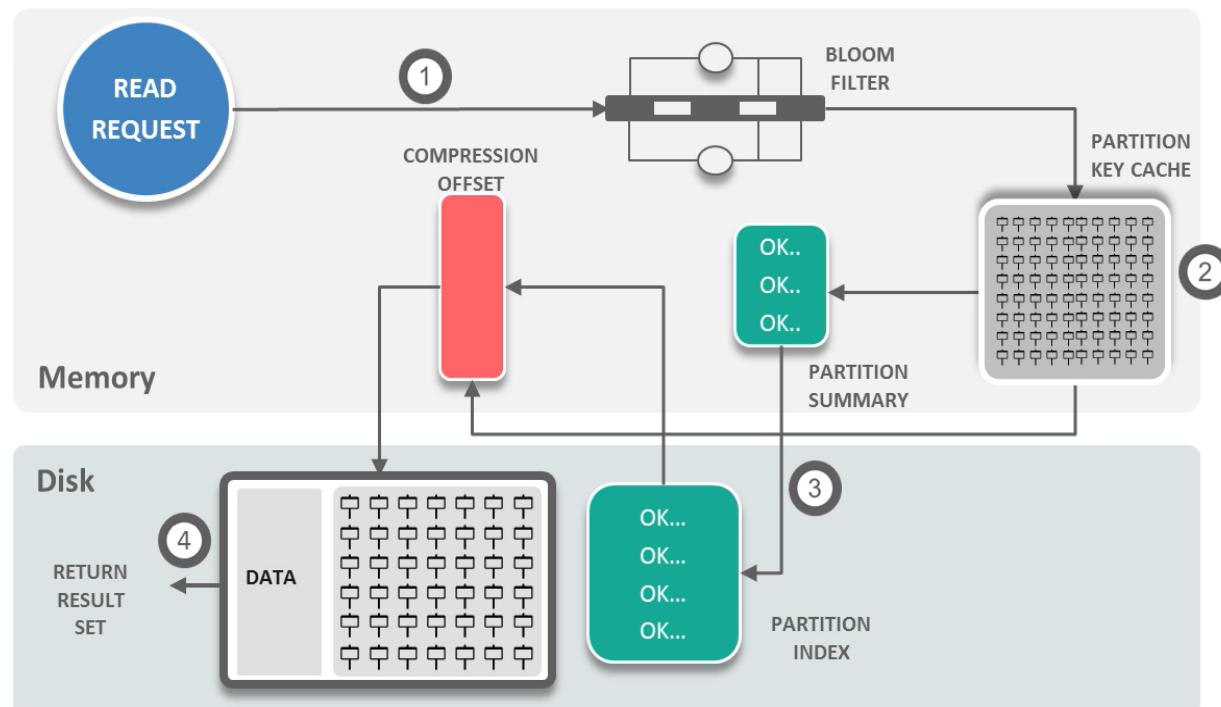


Key Elements – Read Path



Read Path - Process

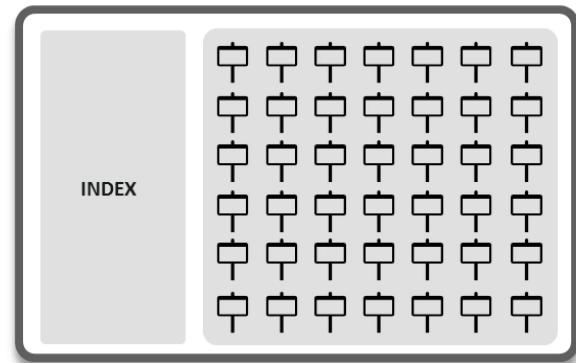
- 1 Cassandra consults a bloom filter that checks the probability of a table having the needed data.
- 2 If the probability is good, Cassandra checks a memory cache that contains row keys and finds the needed key in the cache
- 3 Locates the needed key in Partition Index and finds corresponding data on disk
- 4 Then returns the required result set.



Indexing

An **Index** provides *means to access data* in Cassandra *using attributes* other than the partition key

- The index, *indexes column values in a hidden table*
- This table is *separate* from the one that contains the values being indexed
- The data of an index is local i.e. it will not be replicated to other nodes
- The benefit is fast, efficient lookup of data matching a given condition



Secondary Index

- Secondary index allows query on column in a Cassandra table that is not part of the primary key
- Consider the following scenario

```
cqlsh> use abc;
cqlsh:abc> CREATE TABLE movies
    ... (title text PRIMARY KEY,
    ... also_viewed_title text,
    ... count int);
cqlsh:abc> INSERT INTO movies(title,also_viewed_title,count)VALUES ('Titanic','A
cqlsh:abc> select * from movies where also_viewed_title = 'Avatar';
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING"
cqlsh:abc>
```

- Create a secondary index or use ALLOW FILTERING to resolve the error

```
cqlsh:abc> select * from movies where also_viewed_title='Avatar' ALLOW FILTERING
;

      title   |  also_viewed_title  |  count
-----+-----+-----
    Titanic |          Avatar |      2
```



Regular Secondary Index

- The regular secondary index is the most basic index we can define for executing queries on non-primary key columns.
- Let's define a secondary index on the employee_age column:
- CREATE INDEX IF NOT EXISTS ON company (employee_age);
- SELECT * FROM company WHERE employee_age = 30;

Regular Secondary Index

- When we set up the index, Cassandra creates a hidden table for storing the index data in the background:

```
CREATE TABLE company_by_employee_age_idx (
    employee_age int,
    company_name text,
    employee_email text,
    PRIMARY KEY ((employee_age), company_name,
    employee_email)
);
```



Regular Secondary Index

- Unlike regular tables, Cassandra doesn't distribute the hidden index table using the cluster-wide partitioner.
- The index data is co-located with the source data on the same nodes.
- Therefore, when executing a search query using the secondary index, Cassandra reads the indexed data from every node and collects all the results.
- If our cluster has many nodes, this can lead to increased data transfer and high latency.



Regular Secondary Index

- When we read data based on the secondary index, Cassandra first retrieves the primary keys for all matching rows in the index, and after that, it uses them to fetch all the data from the source table.



SSTable-Attached Secondary Index (SASI)

- SASI introduces the new idea of binding the SSTable lifecycle to the index.
- Performing in-memory indexing followed by flushing the index with the SSTable to disk reduces disk usage and saves CPU cycles.
- CREATE CUSTOM INDEX IF NOT EXISTS
company_by_employee_age ON company
(employee_age) USING
'org.apache.cassandra.index.sasi.SASIIndex';



SSTable-Attached Secondary Index (SASI)

- The advantages of SASI are the tokenized text search, fast range scans, and in-memory indexing.
- On the other hand, a disadvantage is that it generates big index files, especially when enabling text tokenization.



Storage-Attached Indexing (SAI)

- Storage-Attached Indexing is a highly-scalable data indexing mechanism available for DataStax Astra and DataStax Enterprise databases.
- We can define one or more SAI indexes on any column and then use range queries (numeric only), CONTAINS semantics, and filter queries.
- SAI stores individual index files for each column and contains a pointer to the offset of the source data in the SSTable.
- Once we insert data into an indexed column, it will be written first to memory.
- Whenever Cassandra flushes data from memory to disk, it writes the index along with the data table.



Storage-Attached Indexing (SAI)

- This approach improves throughput by 43% and latency by 230% over 2i by reducing the overhead for writing.
- Compared to SASI and 2i, it uses significantly less disk space for indexing, has fewer failure points, and comes with a more simplified architecture.
- `CREATE CUSTOM INDEX ON company
(employee_age) USING 'StorageAttachedIndex' WITH
OPTIONS = {'case_sensitive': false, 'normalize': false};`



Best Practices

- Firstly, when we use secondary indexes in our queries, a recommendation is to add the partition key as a condition.
- As a result, we can reduce the read operation to a single node (plus replicas depending on the consistency level):
- `SELECT * FROM company WHERE employee_age = 30 AND company_name = "company_A";`



Best Practices

- Secondly, we can restrict the query to a list of partition keys and bound the number of nodes involved in fetching the results:
- `SELECT * FROM company WHERE company_name IN ("company_A", "company_B", "company_C") AND employee_age = 30;`



Best Practices

- Thirdly, if we need just a subset of the results, we can add a limit to the query. This also reduces the number of nodes involved in the read path:
- `SELECT * FROM company WHERE employee_age = 30 LIMIT 10;`

Best Practices

- Additionally, we must avoid defining secondary indexes on columns with very low cardinality (gender, true/false columns, etc.) because they produce very wide partitions that impact performance.
- Similarly, columns with high cardinality (social security number, email, etc.) will result in indexes with very granular partitions, which in the worst case will force the cluster coordinator to hit all the primary replicas.
- Lastly, we must avoid using secondary indexes on frequently updated columns.

Indexing – When to use or not



When to use

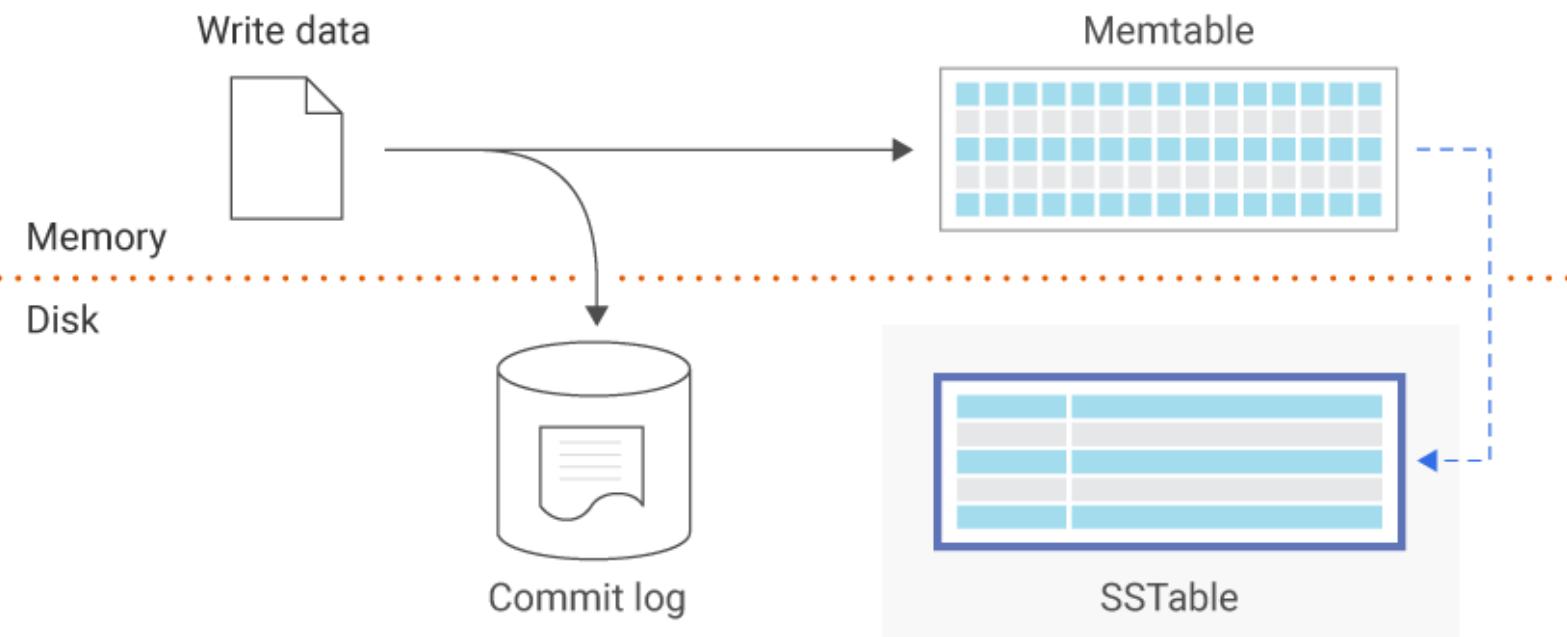
- Built-in Indexes are best on a table having many rows that contain the indexed value.
- The more unique values that exist in a particular column, the more overhead you will have for querying and maintaining the index



When NOT to use

- On high-cardinality columns
- In tables that use a counter column
- On a frequently updated or deleted column
- To look for a row in a large partition unless narrowly queried

SSTable





When Does SSTable Compaction Occur?

- As data continues to be written and updated, more immutable SSTable files are created.
- So, the same record, with different versions of the data, may be found across the different SSTable files on disk.
- The system understands which of these records are the most current, and only responds to query requests with the latest version.
- The SSTable count and data volume stored would get very high, and the disks would fill up.



When Does SSTable Compaction Occur?

- Compaction is a process that writes a whole new file using data found across the extant SSTables.
- This process deduplicates obsolete records and only writes the most current changes for the same key on different SSTables, writing a new SSTable file.
- Deleted rows indicated by a marker called a tombstone or entire deleted columns are also cleaned up, and the process creates a new index for the compacted SSTable file.



When Does SSTable Compaction Occur?

- In size tiered compaction strategy, SSTables are grouped into different buckets based on their size.
- SSTables with similar sizes are grouped in to same bucket.
- For each bucket an average size is calculated using the sizes of SSTables in it.
- There are two options for a bucket, `bucketLow` and `bucketHigh` which define the bucket width.
- If a SSTable size falls between (`bucketLow * average bucket size`) and (`bucketHigh * average bucket size`) then it will be includes in this bucket.
- The average size of the bucket is updated by the newly added SSTable size.
- The default value for `bucketLow` is 0.5 and for `bucketHigh` is 1.5.



When Does SSTable Compaction Occur?

- Next, the buckets with SSTable count greater than or equal to the minimum compaction threshold are selected.
- The thresholds are defined as part of Compaction settings in Column Family schema.
- `compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}`
- Finally, Cassandra selects the bucket that contains SSTables with highest read activity.



When Does SSTable Compaction Occur?

- For each bucket, a read hotness score is assigned and the bucket with highest read hotness score is selected for compaction.
- Read Hotness score for a SSTable is the number of reads per second per key.
- A two-hour read rate is used in the calculation.
- For a bucket, the read hotness score is the combined read hotness score of all SSTable in it.

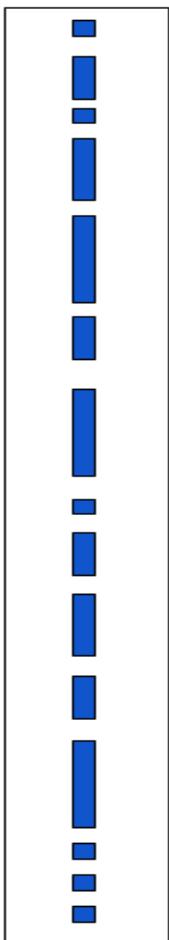


When Does SSTable Compaction Occur?

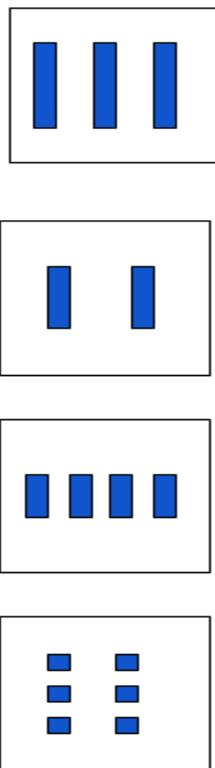
- If two buckets have same read hotness score, then the bucket with smaller SSTables is selected.
- If the selected bucket has SSTable count more than maximum compaction threshold, then the coldest SSTables are dropped from the bucket to meet the max threshold.
- The SSTables in a bucket are sorted in descending order on their read hotness score and the coldest SSTables will be at the end of sorted order.

When Does SSTable Compaction Occur?

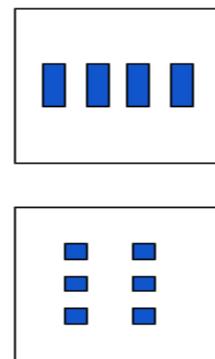
Input SSTables



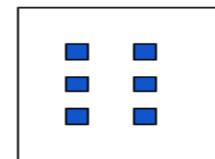
Group SSTables with similar size into Buckets



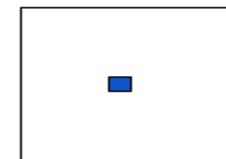
Filter buckets with $\#SSTables \geq \text{minThreshold}$ (4 SSTables)



Select Highest read hotness score bucket



Compact Input SSTables to single Output SSTable



Compaction

- Before performing compaction, the available disk space is checked against the estimated compaction output size.
- If the available disk space is not enough then largest SSTables are dropped from the input list of SSTables to reduce the storage space needed.
- During compaction, the input SSTables are scanned and the same rows from multiple SSTables are merged.
- Compaction can cause heavy disk I/O.
- To limit the disk I/O, there is a configuration setting to throttle the compaction activity `compaction_throughput_mb_per_sec`.
- The setting `concurrent_compactors` limits the number of simultaneous compactations at a time.



How the actual compaction is performed

- A scanner object is created for each input SSTable which can sequentially iterate the partitions in it.
- The input SSTable scanners added to a min heap-based Priority queue to merge the partitions.
- The partitions are consumed from the priority queue.
- The scanner at the root of the priority queue contains the least element in the sorted order of partitions.
- If multiple SSTables contain the same partition which matches with the root node of priority queue, matching partition from all those SSTables are consumes and the SSTables are advanced to point to the next partitions.



How the actual compaction is performed

- Once a scanner is consumed from Priority queue, it is removed from the top and added back at the end and the data structure will re-heapify to bubble up the smallest element to the root.
- During compaction, deleted rows with delete timestamp older than the GC grace period (`gc_before`) are also deleted.



How the actual compaction is performed

```
rps@rps-virtual-machine:/etc/cassandra$ nodetool compactionhistory
Compaction History:
id          keyspace_name columnfamily_name compacted_at      bytes_in bytes_out rows_merged
b11cce60-5eb5-11ed-8438-57a8f77c1a66 system    table_estimates 2022-11-07T21:33:17.190 23784   5437   {4:4}
b10ec4a0-5eb5-11ed-8438-57a8f77c1a66 system    size_estimates  2022-11-07T21:33:17.098 10760   2458   {4:4}
4ec2dd20-5ead-11ed-8438-57a8f77c1a66 system    sstable_activity 2022-11-07T20:33:16.210 772    123    {1:16, 4:2}
8bd366f0-5e9c-11ed-8438-57a8f77c1a66 system    table_estimates 2022-11-07T18:33:17.215 23806   5433   {4:4}
8bc02d10-5e9c-11ed-8438-57a8f77c1a66 system    size_estimates  2022-11-07T18:33:17.089 10783   2461   {4:4}
29764160-5e94-11ed-8438-57a8f77c1a66 system    sstable_activity 2022-11-07T17:33:16.214 773    123    {1:16, 4:2}
6682ac80-5e83-11ed-8438-57a8f77c1a66 system    table_estimates 2022-11-07T15:33:17.192 23837   5432   {4:4}
66773ad0-5e83-11ed-8438-57a8f77c1a66 system    size_estimates  2022-11-07T15:33:17.117 10768   2466   {4:4}
042279b0-5e7b-11ed-8438-57a8f77c1a66 system    sstable_activity 2022-11-07T14:33:16.171 694    123    {1:12, 4:2}
4139e150-5e6a-11ed-8438-57a8f77c1a66 system    table_estimates 2022-11-07T12:33:17.221 23799   5446   {4:4}
4123c140-5e6a-11ed-8438-57a8f77c1a66 system    size_estimates  2022-11-07T12:33:17.076 10788   2456   {4:4}
ded568c0-5e61-11ed-8438-57a8f77c1a66 system    sstable_activity 2022-11-07T11:33:16.172 772    123    {1:16, 4:2}
1bef6870-5e51-11ed-8438-57a8f77c1a66 system    table_estimates 2022-11-07T09:33:17.239 23764   5437   {4:4}
1bd836f0-5e51-11ed-8438-57a8f77c1a66 system    size_estimates  2022-11-07T09:33:17.087 10753   2469   {4:4}
b99c2df0-5e48-11ed-8438-57a8f77c1a66 system    sstable_activity 2022-11-07T08:33:16.303 772    123    {1:16, 4:2}
f69d7580-5e37-11ed-8438-57a8f77c1a66 system    table_estimates 2022-11-07T06:33:17.208 23829   5438   {4:4}
f692ee30-5e37-11ed-8438-57a8f77c1a66 system    size_estimates  2022-11-07T06:33:17.139 10769   2472   {4:4}
943d69c0-5e2f-11ed-8438-57a8f77c1a66 system    sstable_activity 2022-11-07T05:33:16.188 695    123    {1:12, 4:2}
d157dea0-5e1e-11ed-8438-57a8f77c1a66 system    table_estimates 2022-11-07T03:33:17.258 23817   5444   {4:4}
d14519f0-5e1e-11ed-8438-57a8f77c1a66 system    size_estimates  2022-11-07T03:33:17.135 10792   2464   {4:4}
6eeb76d0-5e16-11ed-8438-57a8f77c1a66 system    sstable_activity 2022-11-07T02:33:16.156 776    123    {1:16, 4:2}
ac03efe0-5e05-11ed-8438-57a8f77c1a66 system    table_estimates 2022-11-07T00:33:17.214 24574   5445   {4:4}
```

Major Compaction

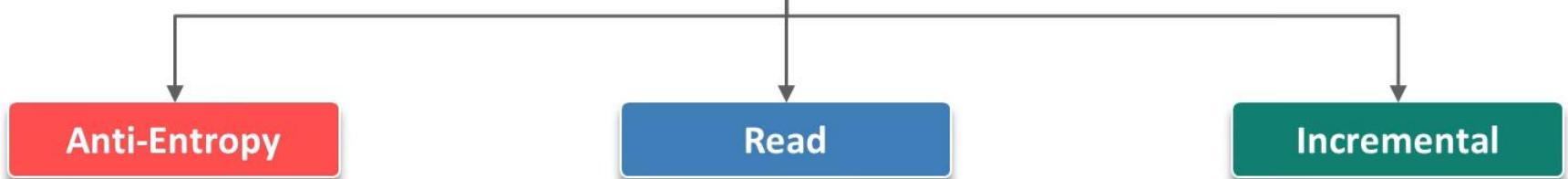
- By default, auto compaction selects the input SSTables for compaction based on the threshold defined as part of Column family definition.
- To run compaction on all the SSTables for a given column family, nodetool compact command can be used to force a major compaction.
- Major compaction can result in two compaction runs one of repaired SSTables and other one is for not repaired SSTables.
- If tombstones and live columns spread across these two different sets of SSTables, major compaction may not be able to drop tombstones.
- The tool `/opt/cassandra/tools/bin/sstablerepairedset` can be used to reset the repairAt to 0 as to include all SSTables in single set.

Node for Repair

- Nodes can become unreachable as they experience:
 - hardware failures
 - get cut off the network
 - will shut down for system maintenance
 - fail to respond under heavy load
 - run out of disk space
- It causes data between the replicas to have diverged and ***repairing is needed***
- A cluster can be considered fully repaired, in case all replicas are identical



Types of Repair





When to run anti-entropy repair

- Run repair in these situations:
 - To routinely maintain node health.
 - Note: Even if deletions never occur, schedule regular repairs. Setting a column to null is a delete.
 - To recover a node after a failure while bringing it back into the cluster.
 - To update data on a node containing data that is not read frequently, and therefore does not get read repair.
 - To update data on a node that has been down.
 - To recover missing data or corrupted SSTables. To do this, you must run non-incremental repair.

When to run anti-entropy repair

- Run repair in these situations:
 - To routinely maintain node health.
 - Note: Even if deletions never occur, schedule regular repairs. Setting a column to null is a delete.
 - To recover a node after a failure while bringing it back into the cluster.
 - To update data on a node containing data that is not read frequently, and therefore does not get read repair.
 - To update data on a node that has been down.
 - To recover missing data or corrupted SSTables. To do this, you must run non-incremental repair.

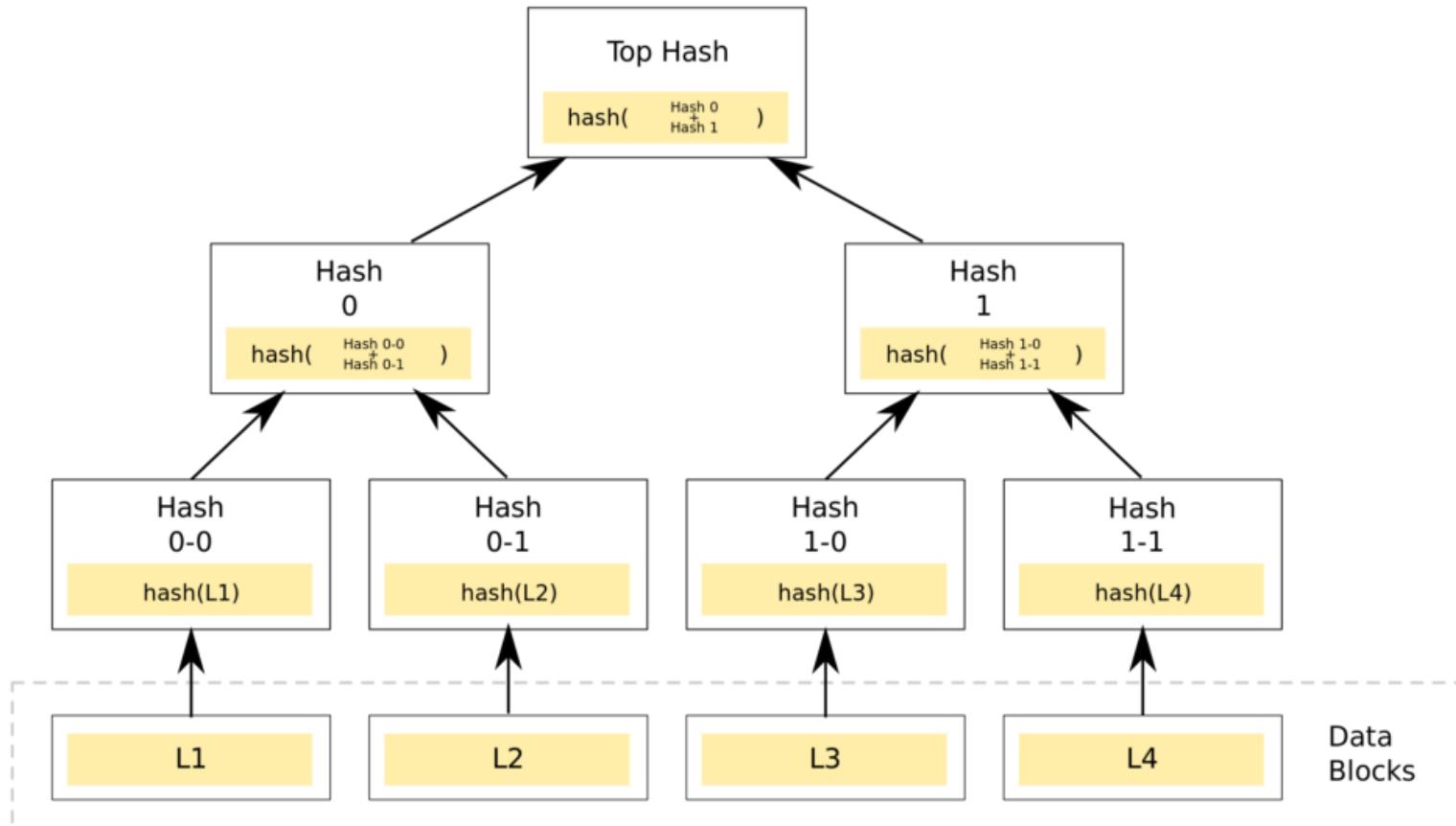
Merkle Tree

- Merkle tree is a tree-based data structure in which each leaf node is a hash of a block of data, and each non-leaf node is hash of its children.
- Merkle trees are used in distributed systems and peer-to-peer networks for efficient data verification.
- Same data exists in multiple places.
- Data changes/Data corruption can be quickly detected by the Merkle trees.
- It is inefficient to check the whole file for differences.
- Data verification using the Merkle trees assume that network I/O takes longer than local I/O to perform hashes.

Anti-Entropy Merkle Tree

- Cassandra accomplishes anti-entropy repair using Merkle trees, similar to Dynamo and Riak.
- Anti-entropy is a process of comparing the data of all replicas and updating each replica to the newest version.
- Cassandra has two phases to the process:
 - Build a Merkle tree for each replica
 - Compare the Merkle trees to discover differences

Anti-Entropy – Merkle Tree



Anti-Entropy – Merkle Tree

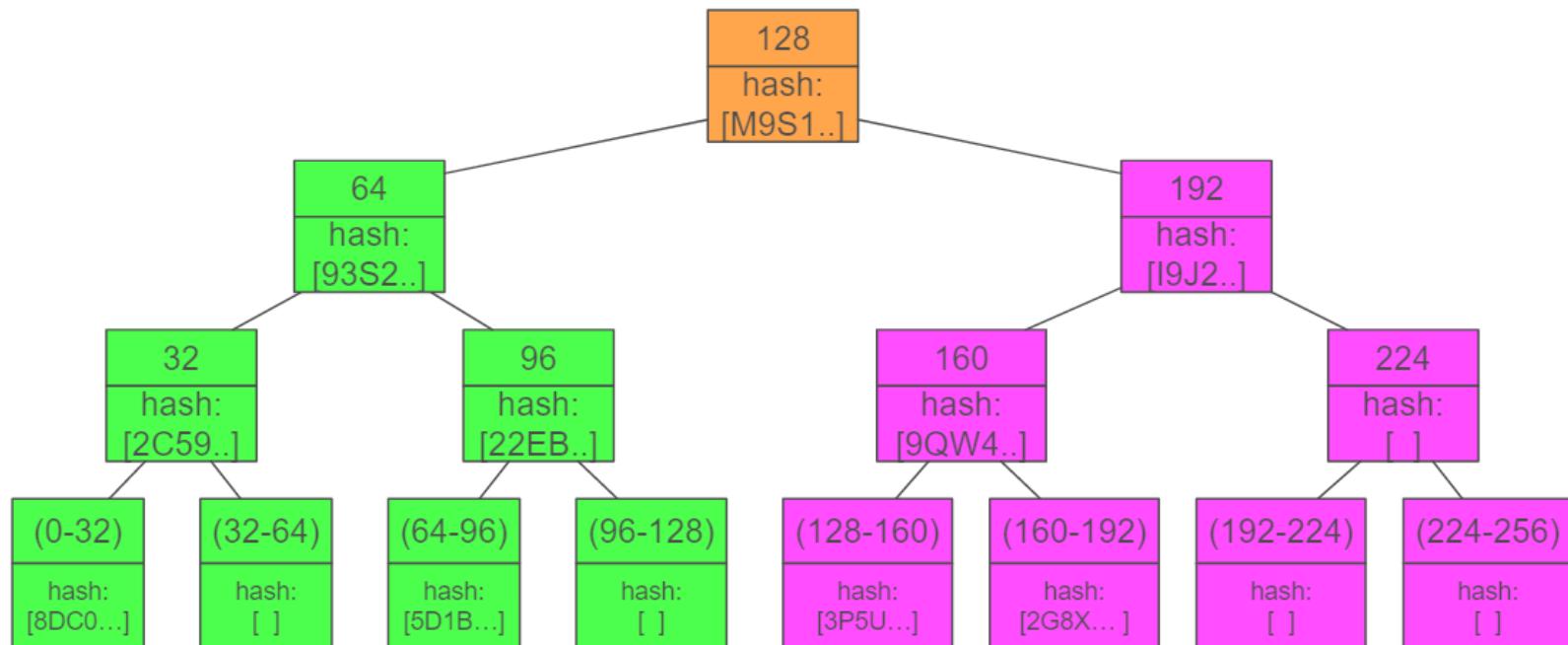
- Merkle trees are binary hash trees whose leaves are hashes of the individual key values.
- The leaf of a Cassandra Merkle tree is the hash of a row value.
- Each Parent node higher in the tree is a hash of its respective children.
- Because higher nodes in the Merkle tree represent data further down the tree, Casandra can check each branch independently without requiring the coordinator node to download the entire data set.
- For anti-entropy repair Cassandra uses a compact tree version with a depth of 15 ($2^{15} = 32K$ leaf nodes).



Anti-Entropy – Merkle Tree

- For example, a node containing a million partitions with one damaged partition, about 30 partitions are streamed, which is the number that fall into each of the leaves of the tree.
- Cassandra works with smaller Merkle trees because they require less storage memory and can be transferred more quickly to other nodes during the comparison process.

Anti-Entropy – Merkle Tree



Row key: jack	Row key: jill	Row key: terry	Row key: misty
Row token: 5	Row token: 7	Row token: 10	Row token: 20
hash: 8DC0....	hash: 5D1B...	hash: 3P5U...	hash: @G8X...



Anti-Entropy – Merkle Tree

- After the initiating node receives the Merkle trees from the participating peer nodes, the initiating node compares every tree to every other tree.
- If a difference is detected, the differing nodes exchange data for the conflicting range(s), and the new data is written to SSTables.
- The comparison begins with the top node of the Merkle tree.
- If no difference is detected, the process proceeds to the left child node and compares and then the right child node.

Anti-Entropy – Merkle Tree

- When a node is found to differ, inconsistent data exists for the range that pertains to that node.
- All data that corresponds to the leaves below that Merkle tree node will be replaced with new data.
- For any given replica set, Cassandra performs validation compaction on only one replica at a time.

Tombstone

- Tombstones are created in Cassandra whenever you delete some data.
- In simple terms, tombstones are pretty much the same as the row of data you just deleted with the addition of a special “delete” marker that tells the database that the data has been deleted.



Tombstone

- Assume that row A is in the table.
- When we delete the row like we delete any rows in relational or non-relational databases.
- Instead of removing this data immediately, Cassandra just adds a “delete marker” to this data.
- It tells that the data has been removed.

Tombstone

- The “delete marker” includes some information about the delete operation itself such as what has been deleted and when the delete happened.
- If a row of data that has 2MB in size deleted, we would expect to free up that 2MB space.
- Instead, Cassandra creates some new metadata with a delete marker which increases the space that the “deleted” data occupies in disk now.



Tombstone

- Tombstone is specific data stored with the data that just deleted.
- It contains information about the delete operation, such as when the delete was performed.
- When we delete some data, all Cassandra does is insert a tombstone.
- It then tells that the delete was successful.



How Long Are Tombstones Retained

- Cassandra carries out a scheduled operation called compaction which you can look at like a maintenance operation.
- The operation consolidates multiple copies of the same data in a host and makes sure future read operations are more performant.
- During this maintenance/repair process, Cassandra also deals with removing the tombstones.
- How often this is carried out is configurable and you will have control over it.

Read Repair

- Read repair improves consistency in a Cassandra cluster with every read request.
- In a read, the coordinator node sends a data request to one replica node and digest requests to others for consistency level (CL) greater than ONE.
- If all nodes return consistent data, the coordinator returns it to the client.

Read Repair

- In read repair, Cassandra sends a digest request to each replica not directly involved in the read.
- Cassandra compares all replicas and writes the most recent version to any replica node that does not have it.
- If the query's consistency level is above ONE, Cassandra performs this process on all replica nodes in the foreground before the data is returned to the client.
- Read repair repairs any node queried by the read.
- This means that for a consistency level of ONE, no data is repaired because no comparison takes place.
- For QUORUM, only the nodes that the query touches are repaired, not all nodes.

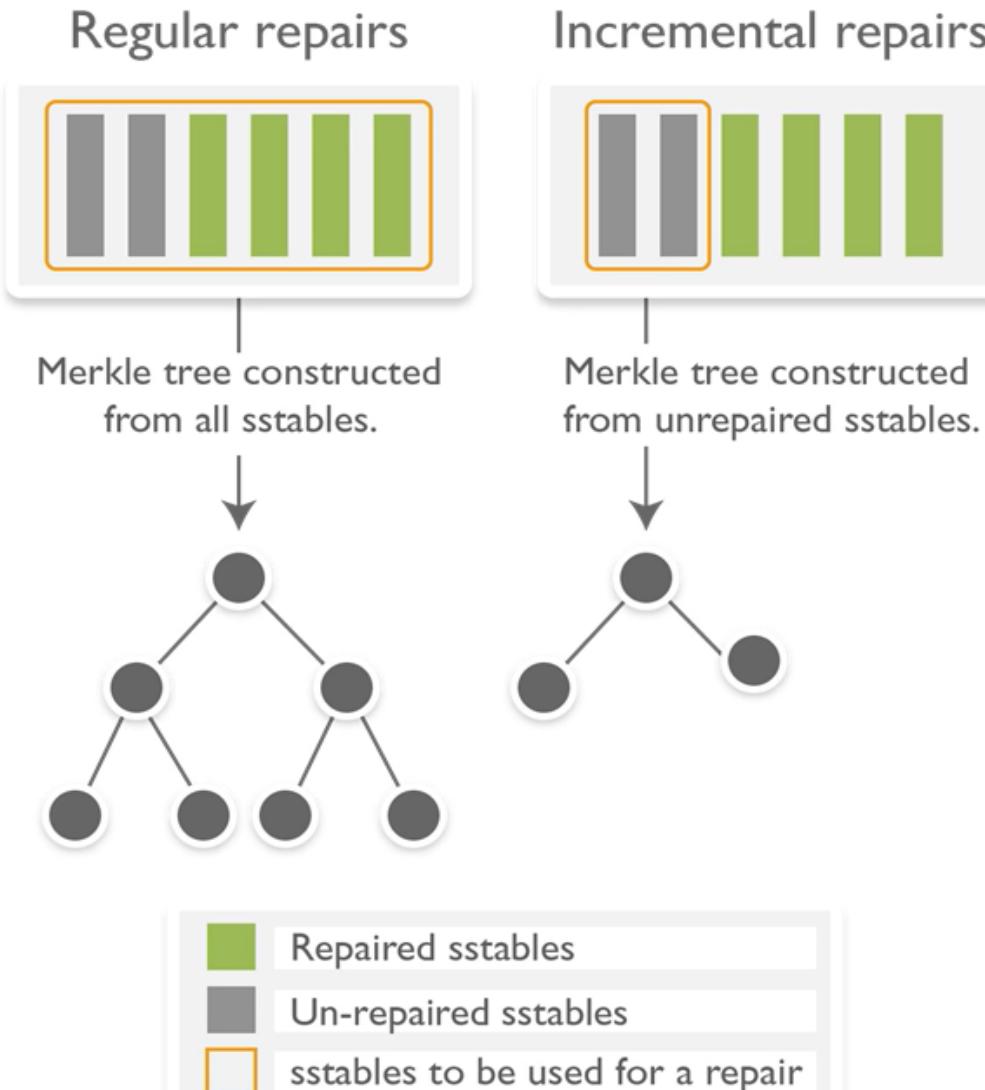
Read Repair

- Cassandra can also perform read repair randomly on a table, independent of any read.
- The `read_repair_chance` property set for a table configures the frequency of this operation.
- Read repair cannot be performed on tables that use `DateTieredCompactionStrategy`, due to the method of checking timestamps used in DTCS compaction.
- If the table uses `DateTieredCompactionStrategy`, set `read_repair_chance` to zero.
- For other compaction strategies, `read_repair_chance` is typically set to a value of 0.2.

Incremental Repair

- Incremental repair consumes less time and resources because it skips SSTables that are already marked as repaired.
- Incremental repair works equally well with any compaction scheme — Size-Tiered Compaction (STCS), Date-Tiered Compaction(DTCS), Time-Window Compaction(TWCS), or Leveled Compaction (LCS).

Incremental Repair



Incremental Repair

- Overview of the procedure
- To migrate one Cassandra node to incremental repair:
 - Disable autocompaction on the node.
 - Run a full, sequential repair.
 - Stop the node.
 - Set the repairedAt metadata value to each SSTable that existed before you disabled compaction.
 - Restart Cassandra on the node.
 - Re-enable autocompaction on the node

Incremental Repair

- Disable autocompaction on the node
 - From the install_directory:
 - \$ bin/nodetool disableautocompaction
 - Running this command without parameters disables autocompaction for all keyspaces.
- Run the default full, sequential repair
 - From the install_directory:
 - \$ bin/nodetool repair
 - Running this command without parameters starts a full sequential repair of all SSTables on the node. This may take a substantial amount of time.

Incremental Repair

- Stop the node.
- Set the repairedAt metadata value to each SSTable that existed before you disabled compaction.
 - Use sstablerepairedset. To mark a single SSTable SSTable-example-Data.db:
 - sudo bin/sstablerepairedset --really-set --is-repaired SSTable-example-Data.db
 - To do this as a batch process using a text file of SSTable names:
 - sudo bin/sstablerepairedset --really-set --is-repaired -f SSTable-names.txt
 - Note: The value of the repairedAt metadata is the timestamp of the last repair. The sstablerepairedset command applies the current date/time. To check the value of the repairedAt metadata for an SSTable, use:
 - \$ bin/sstablemetadata example-keyspace-SSTable-example-Data.db | grep "Repaired at"
- Restart the node.



Size Tiered Compaction Strategy (STCS)

- Recommended for write-intensive workloads.
- The SizeTieredCompactionStrategy (STCS) initiates compaction when Cassandra has accumulated a set number (default: 4) of similar-sized SSTables.
- STCS merges these SSTables into one larger SSTable.
- As these larger SSTables accumulate, STCS merges these into even larger SSTables.
- At any given time, several SSTables of varying sizes are present.

Leveled Compaction Strategy (LCS)

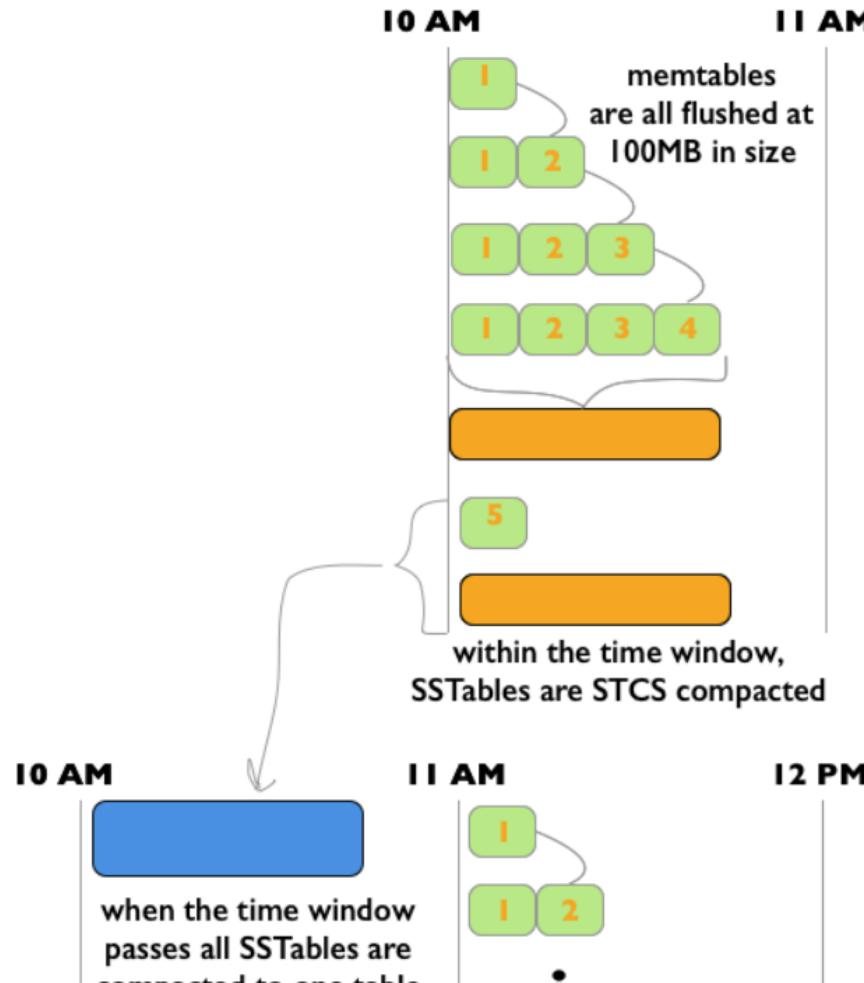
- Recommended for read-intensive workloads.
 - The Leveled Compaction Strategy (LCS) alleviates some of the read operation issues with STCS.
 - This strategy works with a series of levels.
 - First, data in mem tables is flushed to SS Tables in the first level (L0).
 - LCS compaction merges these first SS Tables with larger SS Tables in level L1.
 - The SS Tables in levels greater than L1 are merged into SS Tables with a size greater than or equal to sstable_size_in_mb (default: 160 MB).
 - If a L1 SS Table stores data of a partition that is larger than L2, LCS moves the SS Table past L2 to the next level up.



Time Window Compaction Strategy (TWCS)

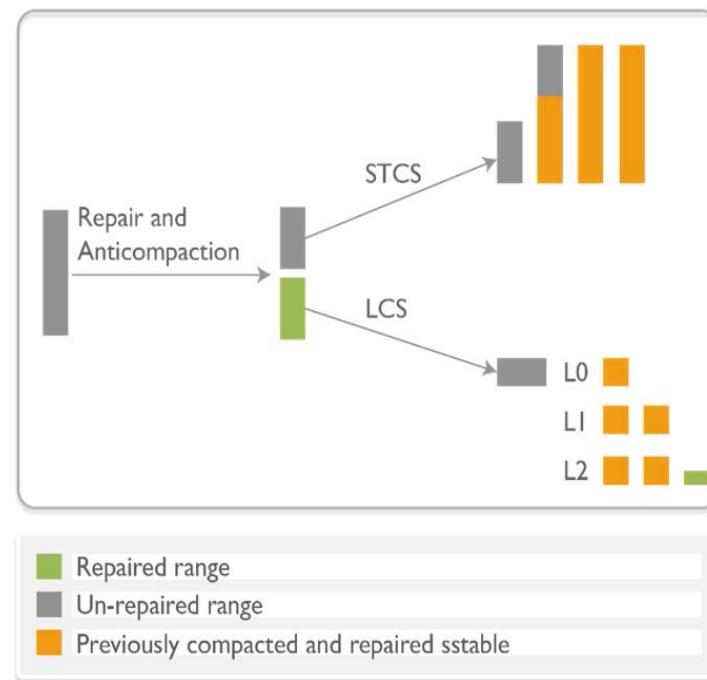
- Recommended for time series and expiring TTL workloads.
 - The Time Window Compaction Strategy (TWCS) is DTCS with simpler settings.
 - TWCS groups SSTables using a series of time windows. During compaction, TWCS applies STCS to uncompacted SSTables in the most recent time window.
 - At the end of a time window, TWCS compacts all SSTables that fall into that time window into a single SSTable based on the SSTable maximum timestamp.
 - Once the major compaction for a time window is completed, no further compaction of the data will ever occur.
 - The process starts over with the SSTables written in the next time window.

Time Window Compaction Strategy (TWCS)



Incremental Repair

Incremental repair works equally well with any compaction scheme — Size-Tiered Compaction (STCS), Date-Tiered Compaction(DTCS) or Leveled Compaction (LCS)



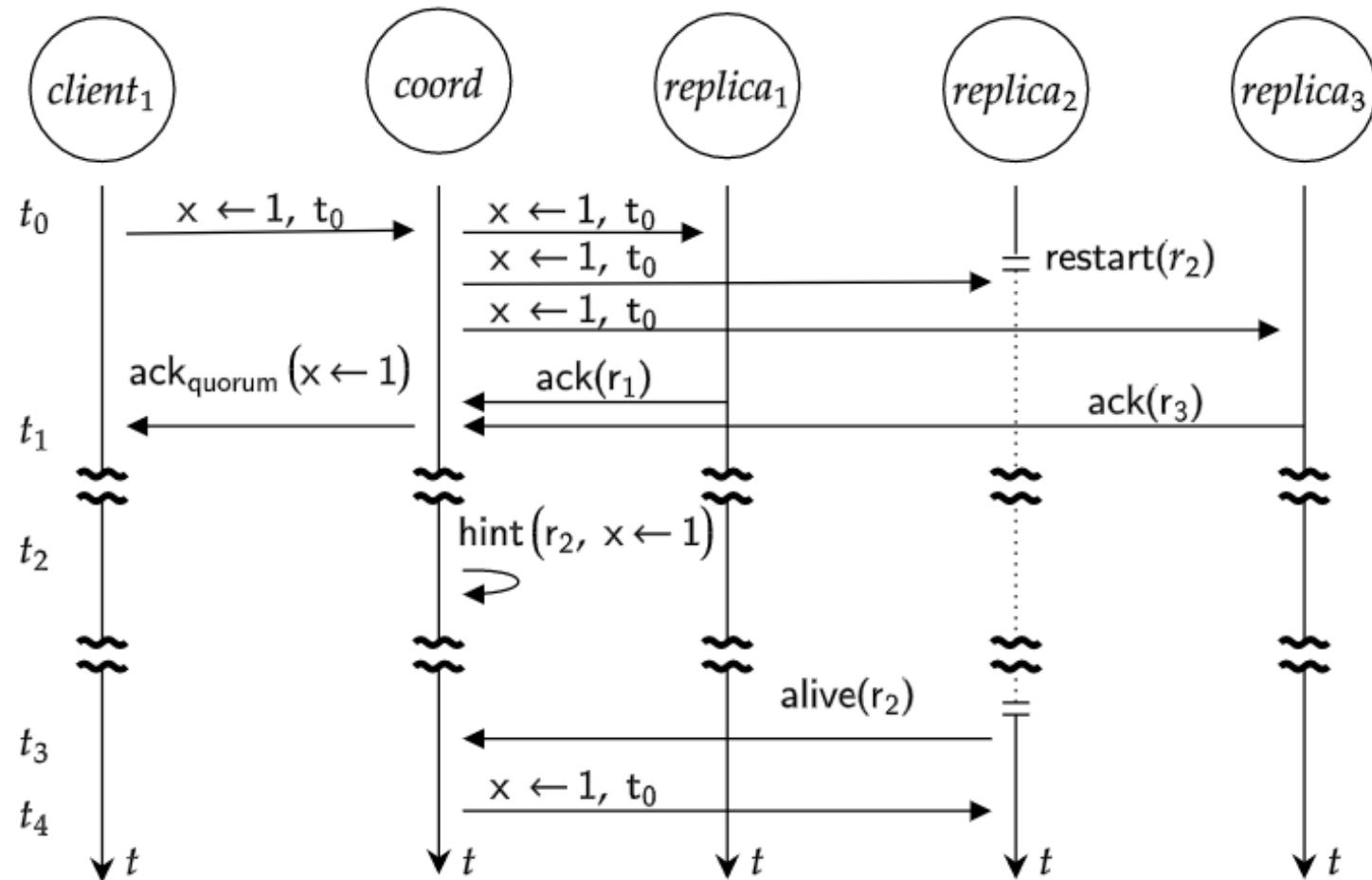


After all these Repairs, What if a Node is Down During a Write Process?

Solution – Hinted Handoff

- Hinted handoff is the process by which Cassandra applies hints to unavailable nodes.
- For example, consider a mutation is to be made at Consistency Level LOCAL_QUORUM against a keyspace with Replication Factor of 3.
- Normally the client sends the mutation to a single coordinator.
- It then sends the mutation to all three replicas, and when two of the three replicas acknowledge the mutation, the coordinator responds successfully to the client.
- If a replica node is unavailable, however, the coordinator stores a hint locally to the filesystem for later application.
- New hints will be retained for up to max_hint_window_in_ms of downtime (defaults to 3 h).
- If the unavailable replica does return to the cluster before the window expires, the coordinator applies any pending hinted mutations against the replica to ensure that eventual consistency is maintained.

Hinted Handoff



Hinted Handoff

- (t0): The write is sent by the client, and the coordinator sends it to the three replicas.
- Unfortunately, replica_2 is restarting and cannot receive the mutation.
- (t1): The client receives a quorum acknowledgement from the coordinator. At this point the client believe the write to be durable and visible to reads (which it is).
- (t2): After the write timeout (default 2s), the coordinator decides that replica_2 is unavailable and stores a hint to its local disk.
- (t3): Later, when replica_2 starts back up it sends a gossip message to all nodes, including the coordinator.
- (t4): The coordinator replays hints including the missed mutation against replica_2.



Hinted Handoff

- If the node does not return in time, the destination replica will be permanently out of sync until either read-repair or full/incremental anti-entropy repair propagates the mutation.
- Hints are enabled by default as they are critical for data consistency.
- The `cassandra.yaml` configuration file provides several settings for configuring hints:

Hinted Handoff

Setting	Description	Default Value
<code>hinted_handoff_enabled</code>	Enables/Disables hinted handoffs	<code>true</code>
<code>hinted_handoff_disabled_datacenters</code>	A list of data centers that do not perform hinted handoffs even when handoff is otherwise enabled. Example:	<code>unset</code>

```
YAML  
hinted_handoff_disabled_datacenters:  
  - DC1  
  - DC2
```



Hinted Handoff

<code>max_hint_window</code>	Defines the maximum amount of time a node shall have hints generated after it has failed.	3h
<code>hinted_handoff_throttle</code>	Maximum throttle in KiBs per second, per delivery thread. This will be reduced proportionally to the number of nodes in the cluster. (If there are two nodes in the cluster, each delivery thread will use the maximum rate; if there are 3, each will throttle to half of the maximum, since it is expected for two nodes to be delivering hints simultaneously.)	1024KiB

Hinted Handoff

<code>max_hints_delivery_threads</code>	Number of threads with which to deliver hints; Consider increasing this number when you have multi-dc deployments, since cross-dc handoff tends to be slower	2
<code>hints_directory</code>	Directory where Cassandra stores hints.	<code>\$CASSANDRA_HOME/data/hints</code>
<code>hints_flush_period</code>	How often hints should be flushed from the internal buffers to disk. Will <i>not</i> trigger fsync.	10000ms
<code>max_hints_file_size</code>	Maximum size for a single hints file, in megabytes.	128MiB



Hinted Handoff

hints_compression	Compression to apply to the hint files. If omitted, hints files will be written uncompressed. LZ4, Snappy, and Deflate compressors are supported.	LZ4Compressor
-------------------	---------------------------------------------------------------------------------------------------------------------------------------------------	---------------



Nodetool Hints

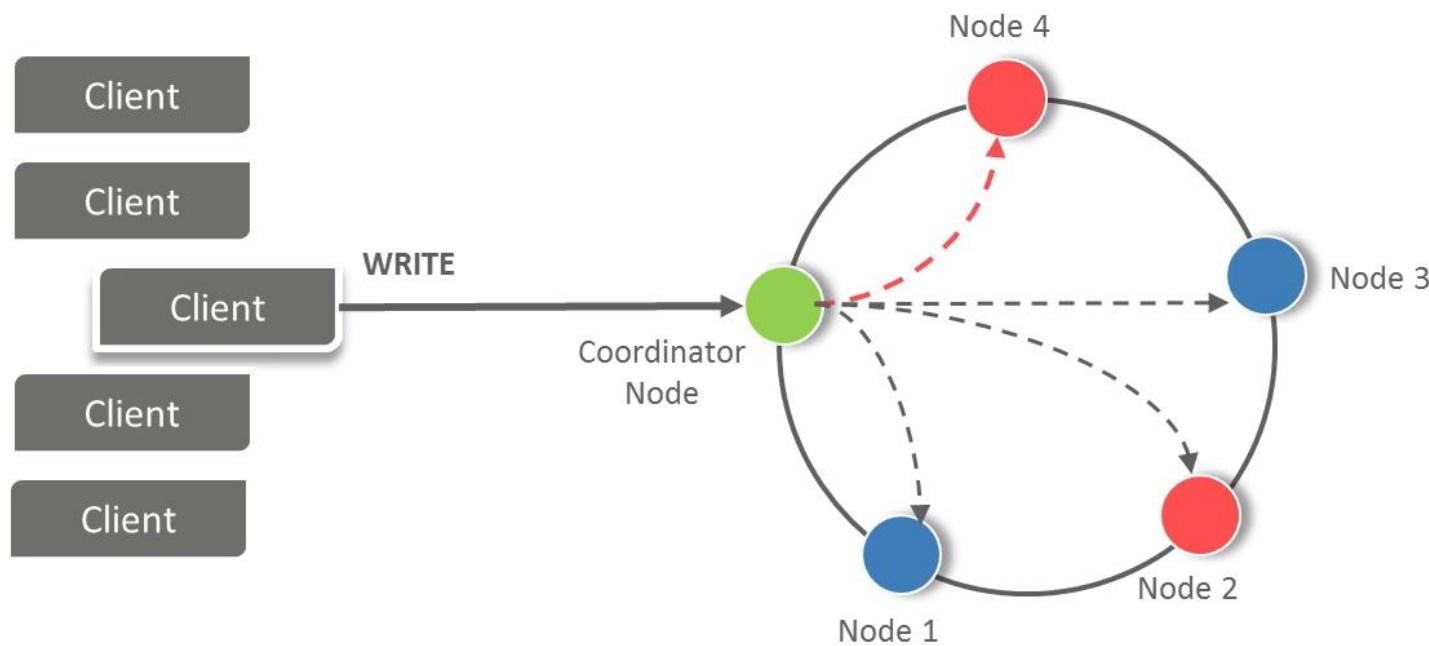
Command	Description
<code>nodetool disablehandoff</code>	Disables storing and delivering hints
<code>nodetool disablehintsfordc</code>	Disables storing and delivering hints to a data center
<code>nodetool enablehandoff</code>	Re-enables future hints storing and delivery on the current node
<code>nodetool enablehintsfordc</code>	Enables hints for a data center that was previously disabled
<code>nodetool getmaxhintwindow</code>	Prints the max hint window in ms. New in Cassandra 4.0.
<code>nodetool handoffwindow</code>	Prints current hinted handoff window
<code>nodetool pausehandoff</code>	Pauses hints delivery process



Let's see how Hinted Handoff solves this Problem?

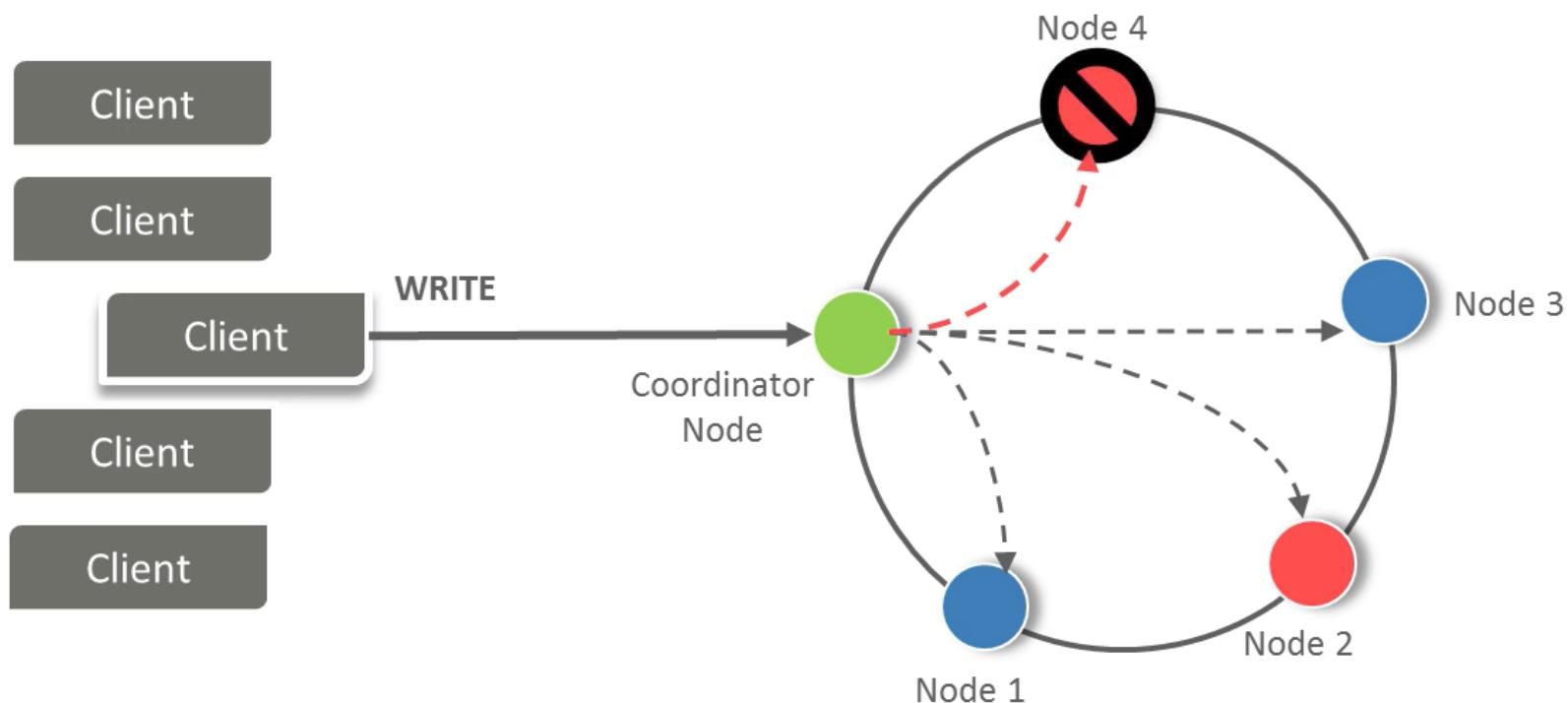
Hinted Handoff : Process

Step 1: Data has to be written on Node 4, for that Request has been given to Coordinator Node



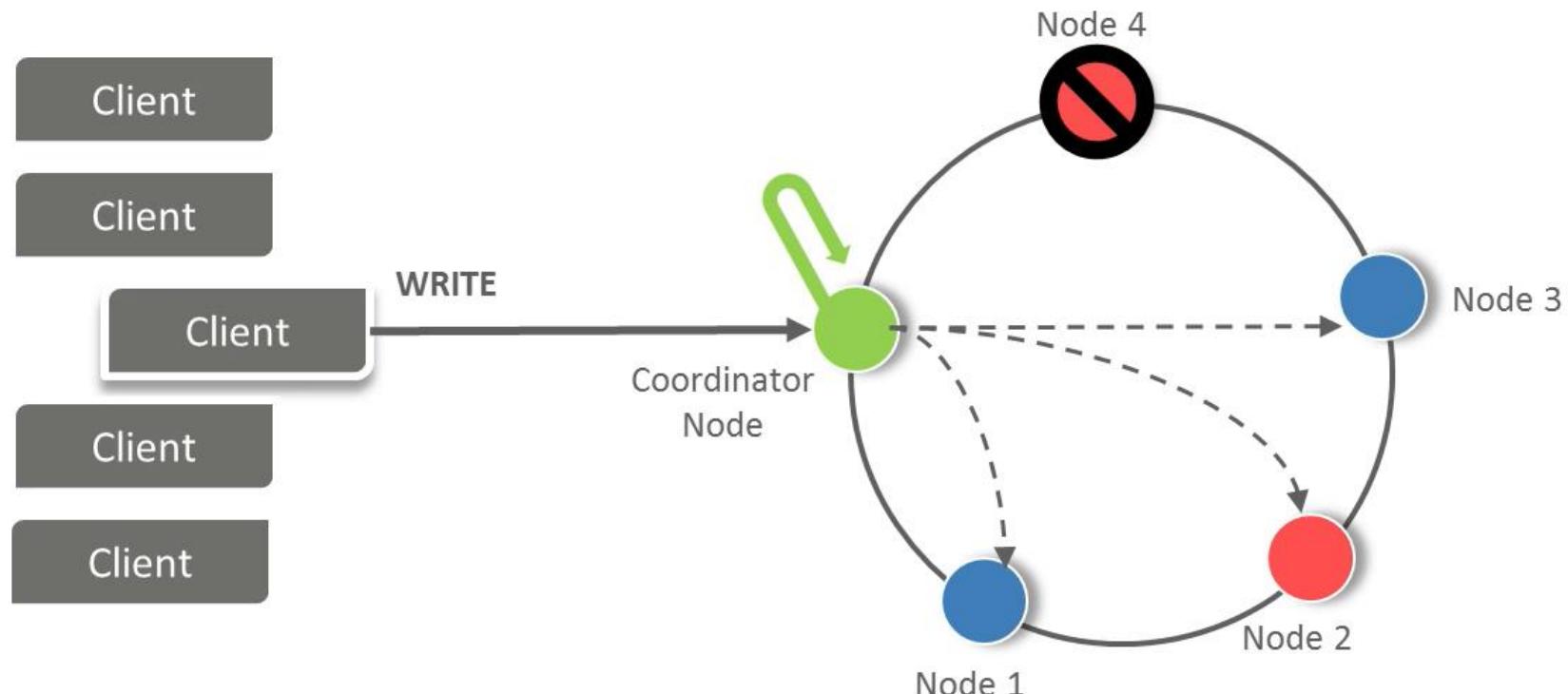
Hinted Handoff : Process

Step 2: For Some Reason Node 4 is not Available



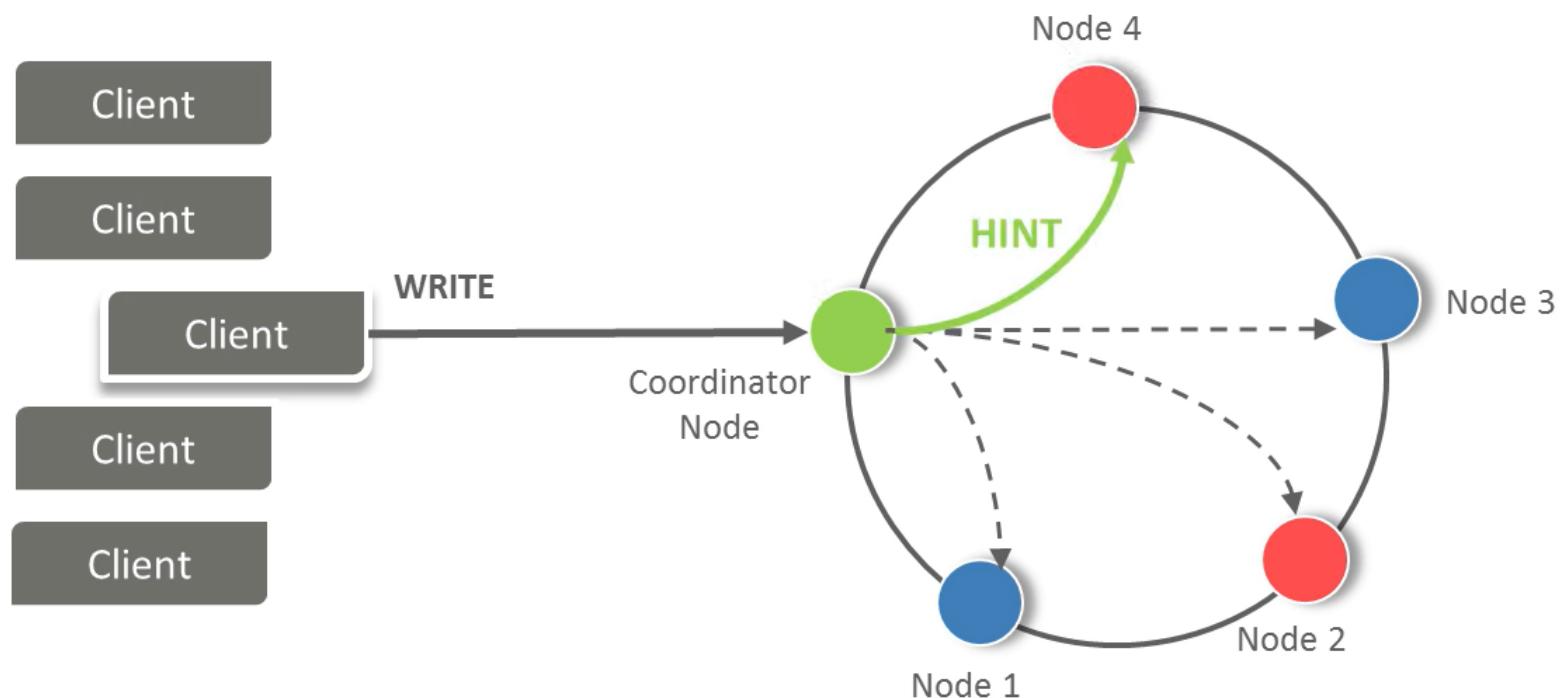
Hinted Handoff : Process

Step 3: Coordinator Node *temporarily stores* the write to itself on behalf of Node 4



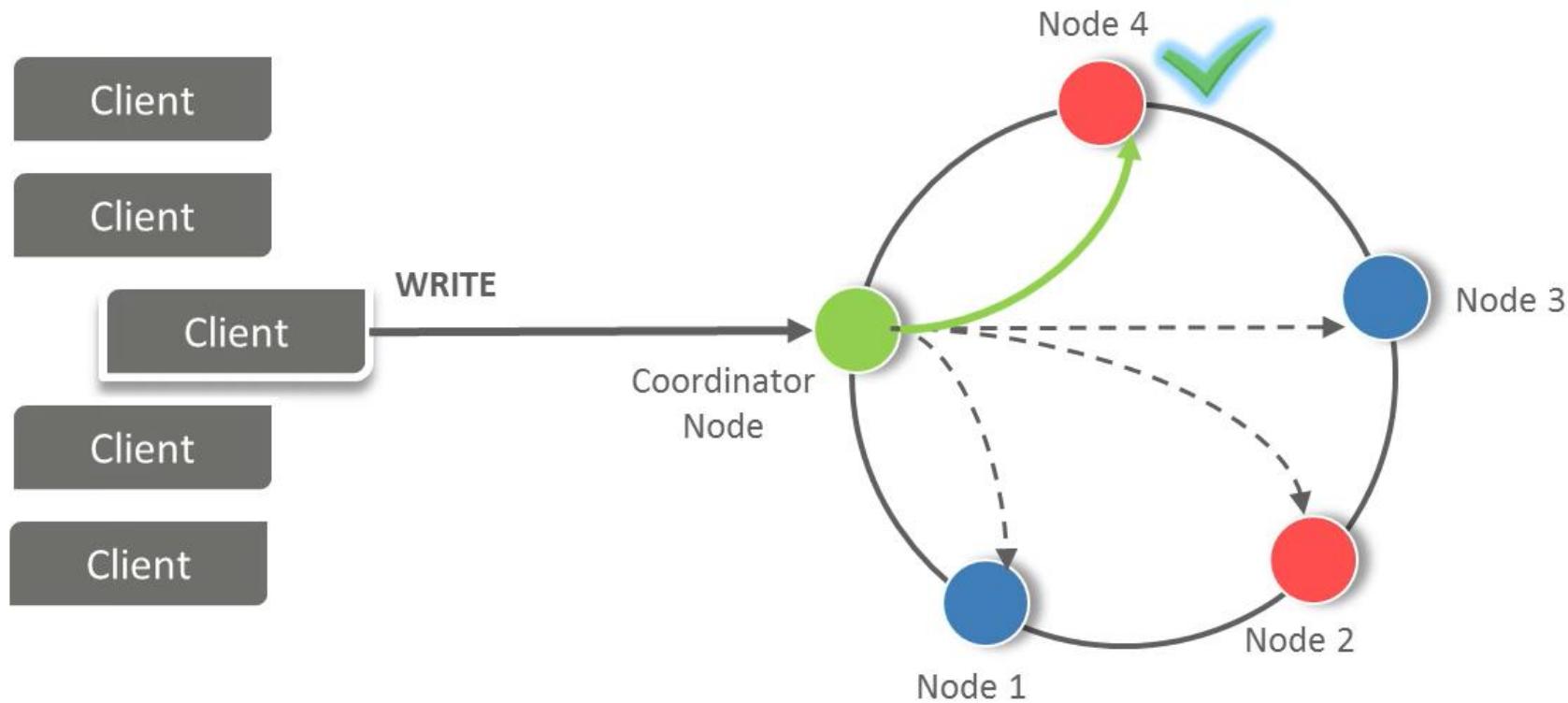
Hinted Handoff : Process

Step 4: Once Node 4 is back up the Coordinator Node forwards the request to Node 4 as a **HINT**



Hinted Handoff : Process

Step 5: The Write is Executed on Node 4

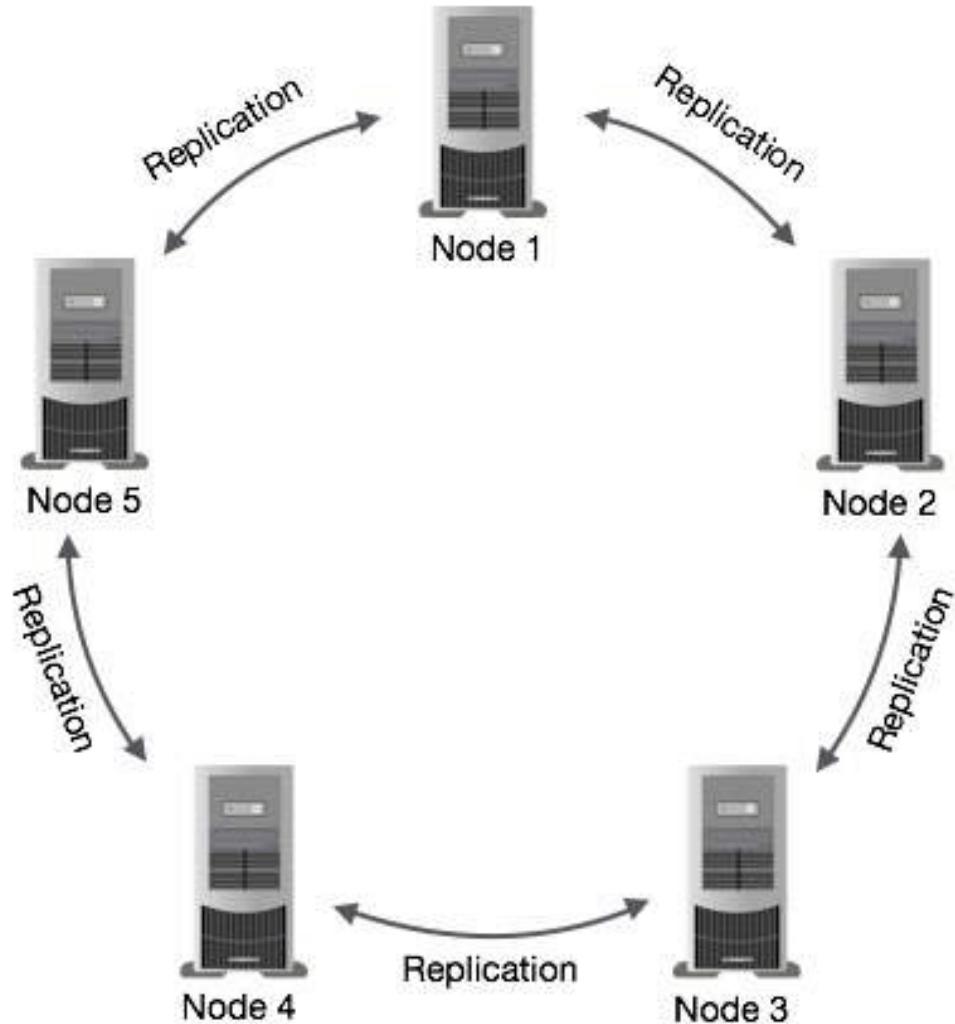




Data Replication in Cassandra

- In Cassandra, nodes in a cluster act as replicas for a given piece of data.
- If some of the nodes are responded with an out-of-date value, Cassandra will return the most recent value to the client.
- After returning the most recent value, Cassandra performs a read repair in the background to update the stale values.

Data Replication in Cassandra

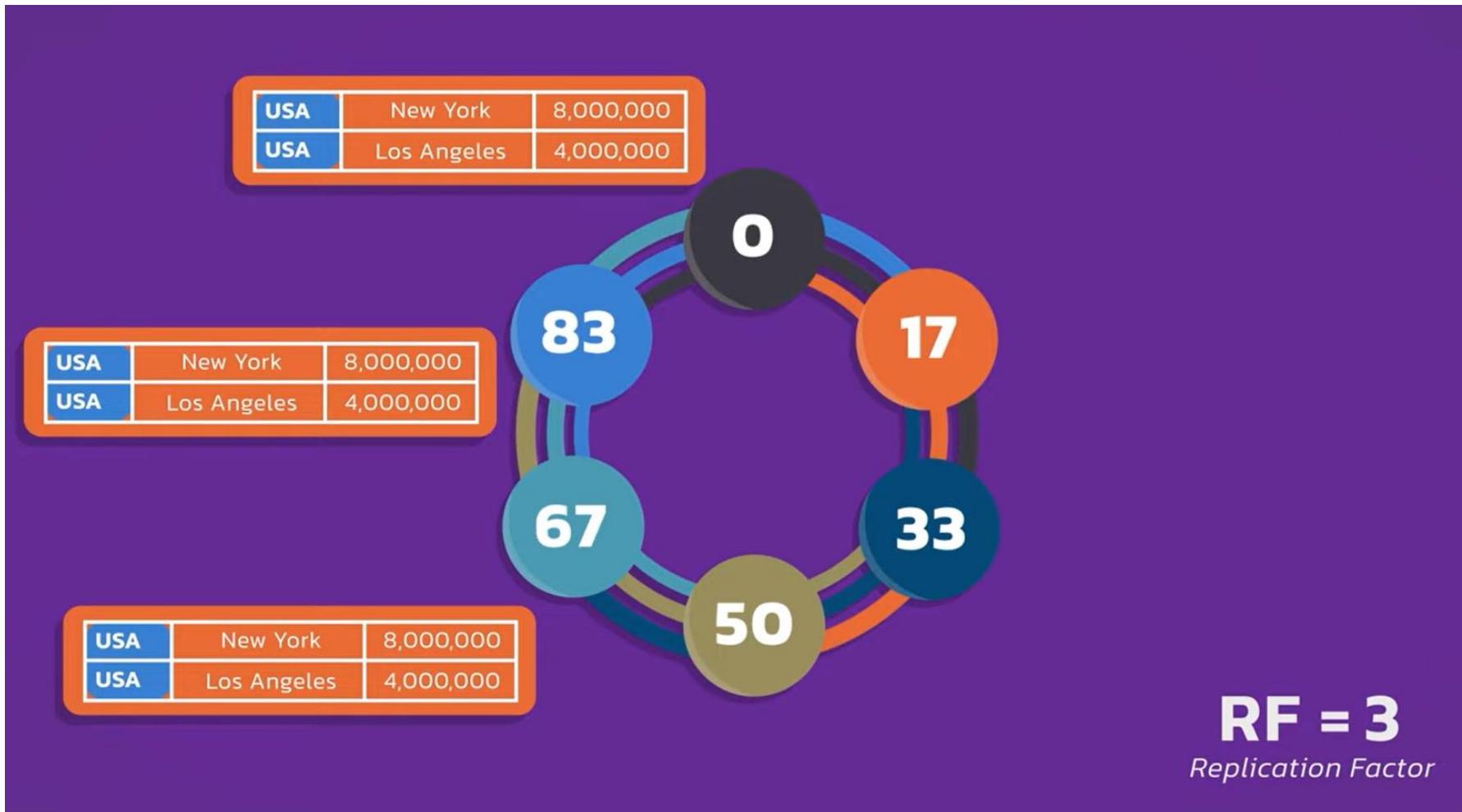




Data Replication in Cassandra

- To avoid a single point of failure, Cassandra makes replicas of data on several nodes.
- Here, there are two things that are important to understanding the process correctly:
- Replication Factor: Replication means the no. of copies maintained on different nodes.
- Replication Factor of 3 means, 3 copies of data maintained on 3 different nodes.
- So, if 2 of the nodes go down, we still have one copy of data safe.

Data Replication in Cassandra





Data Replication Strategies in Cassandra

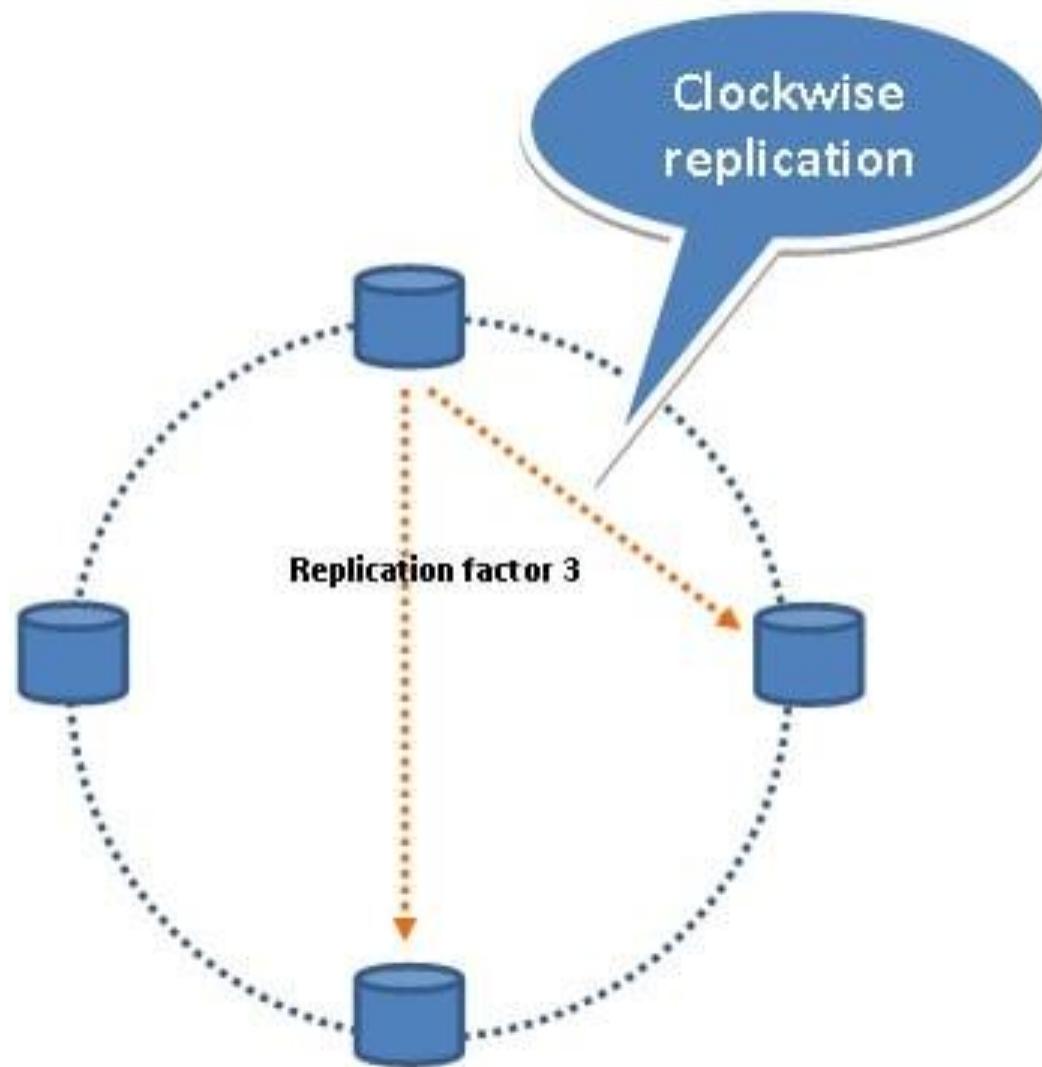
- 1. SimpleStrategy
- 2. LocalStrategy
- 3. NetworkTopologyStrategy



Data Replication Strategies in Cassandra

- SimpleStrategy:
 - It is a simple strategy that is recommended for multiple nodes over multiple racks in a single data center.
 - Let's consider taking an example, strategy_demo is a keyspace name in which class is SimpleStrategy and replication_factor is 2 which simply means there are two redundant copies of each row in a single data center.

Data Replication Strategies in Cassandra





Data Replication Strategies in Cassandra

- CREATE KEYSPACE cluster1 WITH
- replication = {'class': 'SimpleStrategy',
 'replication_factor' : 2};
- describe keyspace cluster1;



Data Replication Strategies in Cassandra

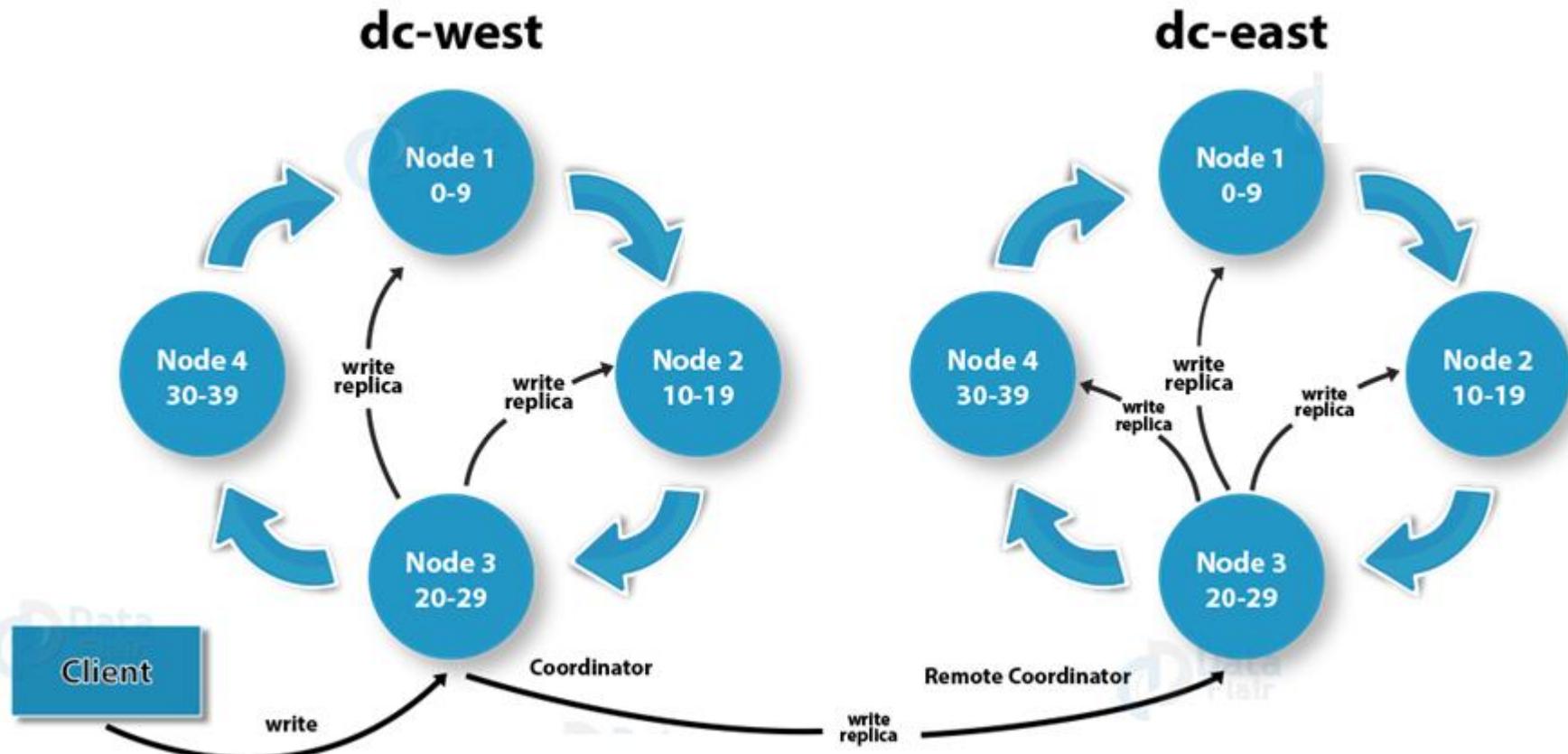
- LocalStrategy:
 - It is the strategy in which we will use a replication strategy for internal purposes such that is used for system and sys_auth keyspaces are internal keyspaces.
 - In Cassandra internal keyspaces implicitly handled by Cassandra's storage architecture for managing authorization and authentication.
 - It is not permissible to creating keyspace with LocalStrategy class if we will try to create such keyspace then it would give an error like “LocalStrategy is for Cassandra's internal purpose only”.



Data Replication Strategies in Cassandra

- Network Topology Strategy:
 - It is the strategy in which we can store multiple copies of data on different data centers as per need.
 - This is one important reason to use `NetworkTopologyStrategy` when multiple replica nodes need to be placed on different data centers.
 - Let's consider an example, `cluster1` is a keyspace name in which `NetworkTopologyStrategy` is a replication strategy and there are two data centers one is east with `RF(Replication Factor) = 2` and second is west with `RF(Replication Fact`

Data Replication Strategies in Cassandra





Data Replication Strategies in Cassandra

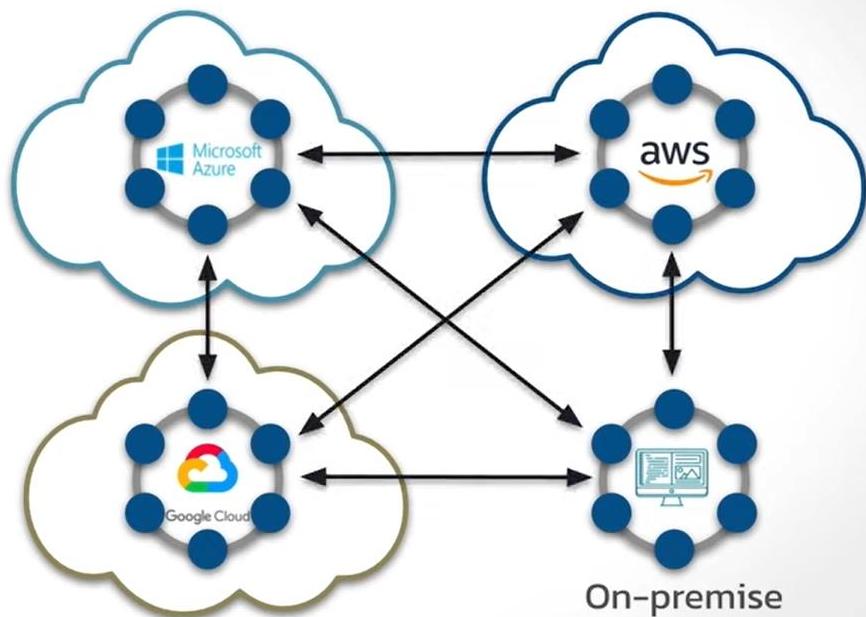
- CREATE KEYSPACE cluster1 WITH
- replication = {'class': 'NetworkTopologyStrategy',
 'east' : 2, 'west' : 3};
- select *
- from system_schema.keyspaces;

Distribution

GEOGRAPHIC DISTRIBUTION



HYBRID CLOUD & MULTI-CLOUD





Cassandra Foreground and Background Mode

- To start Cassandra in the background:
 - cd install_location
 - bin/cassandra #Starts Cassandra
- To start Cassandra in the foreground:
 - cd install_location
 - bin/cassandra -f #Starts Cassandra

Log Structure Access

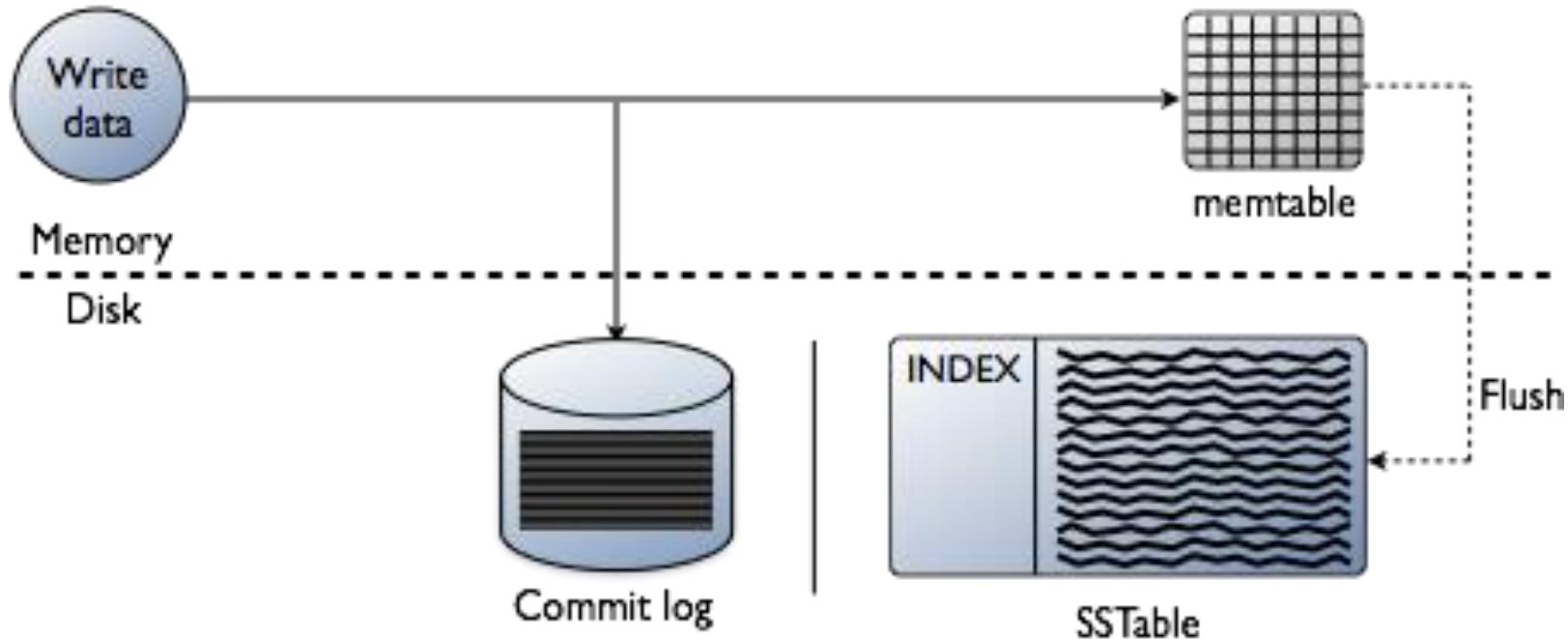
- Cassandra processes data at several stages on the write path, starting with the immediate logging of a write and ending in with a write of data to disk:
 - Logging data in the commit log
 - Writing data to the memtable
 - Flushing data from the memtable
 - Storing data on disk in SSTables



Storing data on disk in SSTables

- Memtables and SSTables are maintained per table.
- The commit log is shared among tables.
- SSTables are immutable, not written to again after the memtable is flushed.
- Consequently, a partition is typically stored across multiple SSTable files.
- A number of other SSTable structures exist to assist read operations:

Storing data on disk in SSTables





Storing data on disk in SSTables

- For each SSTable, Cassandra creates these structures:
- Data (Data.db)
 - The SSTable data
- Primary Index (Index.db)
 - Index of the row keys with pointers to their positions in the data file
- Bloom filter (Filter.db)
 - A structure stored in memory that checks if row data exists in the memtable before accessing SSTables on disk
- Compression Information (CompressionInfo.db)
 - A file holding information about uncompressed data length, chunk offsets and other compression information
- Statistics (Statistics.db)
 - Statistical metadata about the content of the SSTable



Storing data on disk in SSTables

- Digest (Digest.crc32, Digest.adler32, Digest.sha1)
 - A file holding adler32 checksum of the data file
- CRC (CRC.db)
 - A file holding the CRC32 for chunks in an uncompressed file.
- SSTable Index Summary (SUMMARY.db)
 - A sample of the partition index stored in memory
- SSTable Table of Contents (TOC.txt)
 - A file that stores the list of all components for the SSTable TOC
- Secondary Index (SI_.*.db)
 - Built-in secondary index. Multiple SIs may exist per SSTable



Log Files

- Cassandra has three main logs, the system.log, debug.log and gc.log which hold general logging messages, debugging logging messages, and java garbage collection logs respectively.
- These logs by default live in \${CASSANDRA_HOME}/logs, but most Linux distributions relocate logs to /var/log/cassandra.
- Operators can tune this location as well as what levels are logged using the provided logback.xml file



Log Files

- Cassandra has three main logs, the system.log, debug.log and gc.log which hold general logging messages, debugging logging messages, and java garbage collection logs respectively.
- These logs by default live in \${CASSANDRA_HOME}/logs, but most Linux distributions relocate logs to /var/log/cassandra.
- Operators can tune this location as well as what levels are logged using the provided logback.xml file



Log Files

- system.log
 - This log is the default Cassandra log and is a good place to start any investigation. Some examples of activities logged to this log:
 - Uncaught exceptions. These can be very useful for debugging errors.
 - GCInspector messages indicating long garbage collector pauses.
 - When long pauses happen Cassandra will print how long and also what was the state of the system (thread state) at the time of that pause.
 - This can help narrow down a capacity issue (either not enough heap or not enough spare CPU).

Log Files

- system.log
 - Information about nodes joining and leaving the cluster as well as token metadata (data ownership) changes. This is useful for debugging network partitions, data movements, and more.
 - Keyspace/Table creation, modification, deletion.
 - StartupChecks that ensure optimal configuration of the operating system to run Cassandra
 - Information about some background operational tasks (e.g. Index Redistribution).

Log Files

- debug.log
 - This log contains additional debugging information that may be useful when troubleshooting but may be much noisier than the normal system.log.
- Some examples of activities logged to this log:
 - Information about compactions, including when they start, which sstables they contain, and when they finish.
 - Information about memtable flushes to disk, including when they happened, how large the flushes were, and which commitlog segments the flush impacted.
 - This log can be very noisy, so it is highly recommended to use grep and other log analysis tools to dive deep. For example:

Log Files

- gc.log
 - The gc log is a standard Java GC log.
 - With the default jvm.options settings you get a lot of valuable information in this log such as application pause times, and why pauses happened.
 - This may help narrow down throughput or latency issues to a mistuned JVM. For example you can view the last few pauses:



Getting more logging information

- If the default logging levels are insufficient, nodetool can set higher or lower logging levels for various packages and classes using the nodetool setlogginglevel command.
- Start by viewing the current levels:
- \$ nodetool getloginglevels
- nodetool setlogginglevel org.apache.cassandra.gms.Gossiper TRACE
- nodetool getloginglevels



Components of Cassandra

- The main components of Cassandra are:
 - Node: A Cassandra node is a place where data is stored.
 - Data center: Data center is a collection of related nodes.
 - Cluster: A cluster is a component which contains one or more data centers.
 - Commit log: In Cassandra, the commit log is a crash-recovery mechanism. Every write operation is written to the commit log.
 - Mem-table: A mem-table is a memory-resident data structure. After commit log, the data will be written to the mem-table. Sometimes, for a single-column family, there will be multiple mem-tables.



Components of Cassandra

- SSTable: It is a disk file to which the data is flushed from the mem-table when its contents reach a threshold value.
- Bloom filter: These are nothing but quick, nondeterministic, algorithms for testing whether an element is a member of a set. It is a special kind of cache. Bloom filters are accessed after every query.



Cassandra Query Language

- Cassandra Query Language (CQL) is used to access Cassandra through its nodes.
- CQL treats the database (Keyspace) as a container of tables.
- Programmers use cqlsh: a prompt to work with CQL or separate application language drivers.
- The client can approach any of the nodes for their read-write operations.
- That node (coordinator) plays a proxy between the client and the nodes holding the data.



Use Cases/ Applications of Cassandra

- Messaging
 - Cassandra is a great database which can handle a big amount of data. So it is preferred for the companies that provide Mobile phones and messaging services. These companies have a huge amount of data, so Cassandra is best for them.
- Handle high speed Applications
 - Cassandra can handle the high speed data so it is a great database for the applications where data is coming at very high speed from different devices or sensors.



Use Cases/ Applications of Cassandra

- Product Catalogs and retail apps
 - Cassandra is used by many retailers for durable shopping cart protection and fast product catalog input and output.
- Social Media Analytics and recommendation engine
 - Cassandra is a great database for many online companies and social media providers for analysis and recommendation to their customers.



Cassandra Data Types

- Generally, Cassandra supports a rich set of data types.
- These include native types, collection types, user-defined types, and tuples, together with custom types.

Cassandra Data Types

- Native Types
 - The native types are the built-in types and provide support to a range of constants in Cassandra.
 - To begin with, a string is a very popular datatype in the programming world.
 - CQL offers four different datatypes for strings:

Data Type	Constants Supported	Description
ascii	<i>string</i>	ASCII character string
inet	<i>string</i>	IPv4 or IPv6 address string
text	<i>string</i>	UTF8 encoded string
varchar	<i>string</i>	UTF8 encoded string

Cassandra Data Types

Data Type	Constants Supported	Description
boolean	<i>boolean</i>	<i>true</i> or <i>false</i>

Using the blob data type, we can store images or multimedia data as a binary stream in a database:

Data Type	Constants Supported	Description
blob	<i>blob</i>	Arbitrary bytes

Duration is a three-signed integer that represents months, days, and nanoseconds:

Data Type	Constants Supported	Description
duration	<i>duration</i>	A duration value



Cassandra Data Types

Cassandra offers a wide range of data types for integer data:

Data Type	Constants Supported	Description
tinyint	<i>integer</i>	8-bit signed int
smallint	<i>integer</i>	16-bit signed int
int	<i>integer</i>	32-bit signed int
bigint	<i>integer</i>	64-bit signed long
variant	<i>integer</i>	Arbitrary-precision integer
counter	<i>integer</i>	Counter column (64-bit signed)

For integer and float, we have three data types:

Data Type	Constants Supported	Description
decimal	<i>integer, float</i>	Variable precision decimal
double	<i>integer, float</i>	64-bit floating-point
float	<i>integer, float</i>	32-bit floating-point

For date- and time-related needs, Cassandra provides three data types:



Cassandra Data Types

Data Type	Constants Supported	Description
date	<i>integer, string</i>	A date value (without time)
time	<i>integer, string</i>	A time value (without date)
timestamp	<i>integer, string</i>	A timestamp (with date & time)

Generally, we have to avoid collision while using the INSERT or UPDATE commands:

Data Type	Constants Supported	Description
uuid	<i>uuid</i>	A UUID (any version)
timeuuid	<i>uuid</i>	A version 1 UUID



Cassandra Data Types

- When a user has multiple values against one field in a relational database, it's common to store them in a separate table.
- For example, a user has numerous bank accounts, contact information, or email addresses.
- Therefore, we need to apply joins between two tables to retrieve all the data in this case.
- Cassandra provides a way to group and store data together in a column using collection types.
- Let's quickly look at those types:
- set – unique values; stored as unordered
- list – can contain duplicate values; order matters
- map – data stores in the form of key-value pairs



Cassandra Data Types

- User-Defined Types
- User-defined types give us the liberty to attach multiple data fields in a single column:

```
CREATE TYPE student.basic_info (
    birthday timestamp,
    race text,
    weight text,
    height text
);
```



Cassandra Data Types

- Tuple Type
- A tuple is an alternative to a user-defined type. It's created using angle brackets and a comma delimiter to separate the types of elements it contains.
- Here are the commands for a simple tuple:
- -- create a tuple

```
CREATE TABLE subjects (
```

```
    k int PRIMARY KEY,  
    v tuple<int, text, float>  
);
```

- -- insert values
- INSERT INTO subjects (k, v) VALUES(0, (3, 'cs', 2.1));
- -- retrieve values
- SELECT * FROM subjects;



Keyspace Commands

- The first thing to remember is that a keyspace in Cassandra is much like a database in RDBMS.
- It is an outermost container of data that defines the replication strategy and other options, particularly for all the keyspace tables.
- A good general rule is one keyspace per application.



Keyspace Commands

Command	Example	Description
CREATE keyspace	<code>CREATE KEYSPACE <i>keyspace_name</i> WITH replication = {'class':'SimpleStrategy', 'replication_factor' : 2};</code>	To create a keyspace.
DESCRIBE keyspace	<code>DESCRIBE KEYSPACES;</code>	It will list all the key spaces.
USE keyspace	<code>USE <i>keyspace_name</i>;</code>	This command connects the client session to a keyspace.
ALTER keyspace	<code>ALTER KEYSPACE <i>keyspace_name</i> WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 3 } AND DURABLE_WRITES = false;</code>	To alter a keyspace.
DROP keyspace	<code>DROP KEYSPACE <i>keyspace_name</i>;</code>	To drop a keyspace



Table Commands

- In Cassandra, a table is also referred to as a column family.
- We already know the importance of a primary key.
- It is mandatory to define the primary key while creating the table.



Table Commands

Command	Example	Description
CREATE table	<pre>CREATE TABLE <i>table_name</i> (<i>column_name</i> UUID PRIMARY KEY, <i>column_name</i> text, <i>column_name</i> text, <i>col umn_name</i> timestamp);</pre>	To create a table.
ALTER table	<pre>ALTER TABLE <i>table_name</i> ADD <i>column_name</i> int;</pre>	It will add a new column to a table.
ALTER table	<pre>ALTER TABLE <i>table_name</i> ALTER <i>column_name</i> TYP E <i>datatype</i>;</pre>	We can change the data type of an existing column.
ALTER table	<pre>ALTER TABLE <i>table_name</i> WITH caching = {'keys' : 'NONE', 'rows_per_partition' : '1'};</pre>	This command helps to alter the properties of a table.
DROP table	<pre>DROP TABLE <i>table_name</i>;</pre>	To drop a table.
TRUNCATE table	<pre>TRUNCATE <i>table_name</i>;</pre>	Using this, we can remove all the data permanently.



Index Commands

- Instead of scanning a whole table and waiting for results, we can use indexes to speed up queries.
- Primary key in Cassandra is already indexed.
- Therefore, it cannot be used for the same purpose again.

Command	Example	Description
CREATE index	CREATE INDEX <i>index_name</i> on <i>table_name</i> (<i>column_name</i>);	To create an index.
DELETE index	DROP INDEX IF EXISTS <i>index_name</i> ;	To drop an index.



Basic Commands

Command	Example	Description
INSERT	<code>INSERT INTO <i>table_name</i> (<i>column_name1</i>, <i>column_name2</i>) VALUES(<i>value1</i>, <i>value2</i>);</code>	To insert a record in a table.
SELECT	<code>SELECT * FROM <i>table_name</i>;</code>	The command is used to fetch data from a specific table.
WHERE	<code>SELECT * FROM <i>table_name</i> WHERE <i>column_name</i>= <i>value</i>;</code>	It filters out records on a predicate.
UPDATE	<code>UPDATE <i>table_name</i> SET <i>column_name2</i>= <i>value2</i> WHERE <i>column_name1</i>=<i>value1</i>;</code>	It is used to edit records.
DELETE	<code>DELETE <i>identifier</i> FROM <i>table_name</i> WHERE <i>condition</i>;</code>	This statement deletes the value from a table.



Other Commands

Command	Example	Description
ORDER BY	<pre>SELECT * FROM <i>table_name</i> WHERE <i>column_name1</i> = <i>value</i> ORDER BY <i>column_name2</i> ASC;</pre>	For this, the partition key must be defined in the WHERE clause. Also, the ORDER BY clause represents the clustering column to use for ordering.
GROUP BY	<pre>SELECT <i>column_name</i> FROM <i>table_name</i> GROUP BY <i>condition1, condition2</i>;</pre>	This clause only supports with Partition Key or Partition Key and Clustering Key.
LIMIT	<pre>SELECT * FROM <i>table_name</i> LIMIT 3;</pre>	For a large table, limit the number of rows retrieved.



Cassandra Data Model

- Cluster
 - Cassandra database is distributed over several machines that are operated together.
 - The outermost container is known as the Cluster which contains different nodes.
 - Every node contains a replica, and in case of a failure, the replica takes charge.
 - Cassandra arranges the nodes in a cluster, in a ring format, and assigns data to them.



Cassandra Data Model

- Keyspace
 - Keyspace is the outermost container for data in Cassandra.
- Following are the basic attributes of Keyspace in Cassandra:
- Replication factor: It specifies the number of machine in the cluster that will receive copies of the same data.
- Replica placement Strategy: It is a strategy which specifies how to place replicas in the ring.
- There are three types of strategies such as:
 - Simple strategy (rack-aware strategy)
 - old network topology strategy (rack-aware strategy)
 - network topology strategy (datacenter-shared strategy).



Cassandra Data Model

- Column families: column families are placed under key space.
- A key space is a container for a list of one or more column families while a column family is a container of a collection of rows.
- Each row contains ordered columns.
- Column families represent the structure of your data.
- Each key space has at least one and often many column families.
- In Cassandra, a well data model is very important because a bad data model can degrade performance, especially when you try to implement the RDBMS concepts on Cassandra.



Cassandra data Model Rules

- Cassandra doesn't support JOINS, GROUP BY, OR clause, aggregation etc.
- So, we must store data in a way that it should be retrieved whenever you want.
- Cassandra is optimized for high write performances so you should maximize your writes for better read performance and data availability.
- There is a tradeoff between data write and data read.
- So, optimize data read performance by maximizing the number of data writes.
- Maximize data duplication because Cassandra is a distributed database and data duplication provides instant availability without a single point of failure.



Data Modeling Goals

- Spread Data Evenly Around the Cluster:
- To spread equal amount of data on each node of Cassandra cluster, we must choose integers as a primary key.
- Data is spread to different nodes based on partition keys that are the first part of the primary key.
- Minimize number of partitions read while querying data:
- Partition is used to bind a group of records with the same partition key.
- When the read query is issued, it collects data from different nodes from different partitions.



Data Modeling Goals

- In the case of many partitions, all these partitions need to be visited for collecting the query data.
- It does not mean that partitions should not be created. If your data is very large, you can't keep that huge amount of data on the single partition.
- The single partition will be slowed down. So, we must have a balanced number of partitions



When You Should Think About Using Cassandra

- Cassandra's design criteria are the following:
 - Distributed: Runs on more than one server node.
 - Scale linearly: By adding nodes, not more hardware on existing nodes.
 - Work globally: A cluster may be geographically distributed.
 - Favor writes over reads: Writes are an order of magnitude faster than reads.
 - Democratic peer to peer architecture: No master/slave.
 - Favor partition tolerance and availability over consistency: Eventually.
 - Support fast targeted reads by primary key: Focus on primary key reads alternative paths are very sub-optimal.
 - Support data with a defined lifetime: All data in a Cassandra database has a defined lifetime no need to delete it after the lifetime expires the data goes away.



Wrong usecases for cassandra

- Tables have multiple access paths. Example: lots of secondary indexes.
- The application depends on identifying rows with sequential values.
- MySQL autoincrement or Oracle sequences.
- Cassandra does not do ACID.
- Aggregates: Cassandra does not support aggregates, if you need to do a lot of them, think another database.



Wrong usecases for cassandra

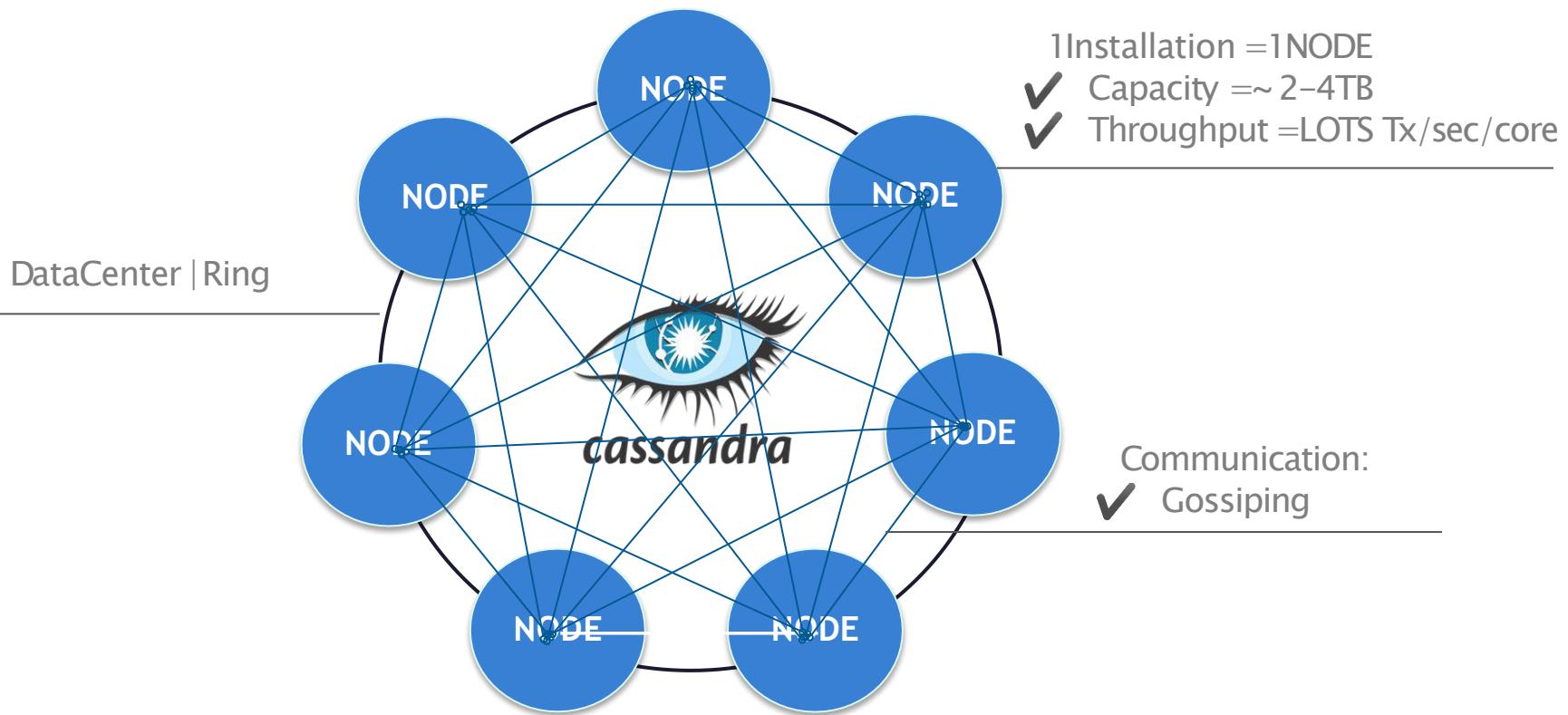
- Joins: You many be able to data model yourself out of this one, but take care.
- Locks: Honestly, Cassandra does not support locking. There is a good reason for this. Don't try to implement them yourself. I have seen the end result of people trying to do locks using Cassandra and the results were not pretty.



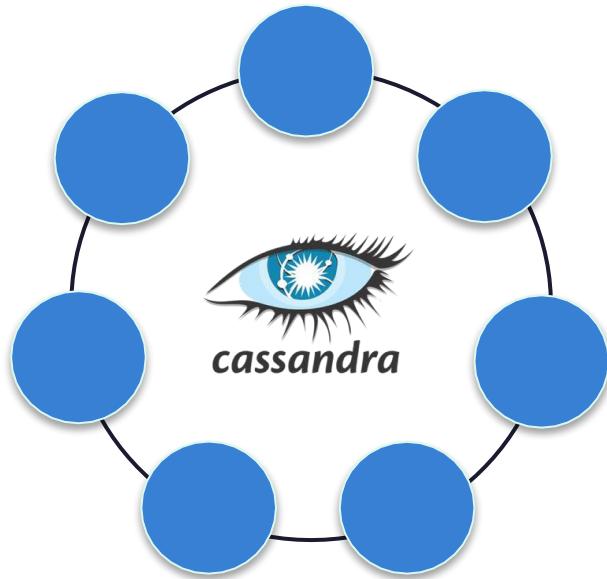
Wrong usecases for cassandra

- Updates: Cassandra is very good at writes, okay with reads. Updates and deletes are implemented as special cases of writes and that has consequences that are not immediately obvious.
- Transactions: CQL has no begin/commit transaction syntax. If you think you need it then Cassandra is a poor choice for you. Don't try to simulate it. The results won't be pretty.

Apache Cassandra™ = NoSQL Distributed Database



Apache Cassandra™ = NoSQL Distributed Database



- Big Data Ready
- Highest Availability
- Geographical Distribution
- Read/Write Performance
- Vendor Independent

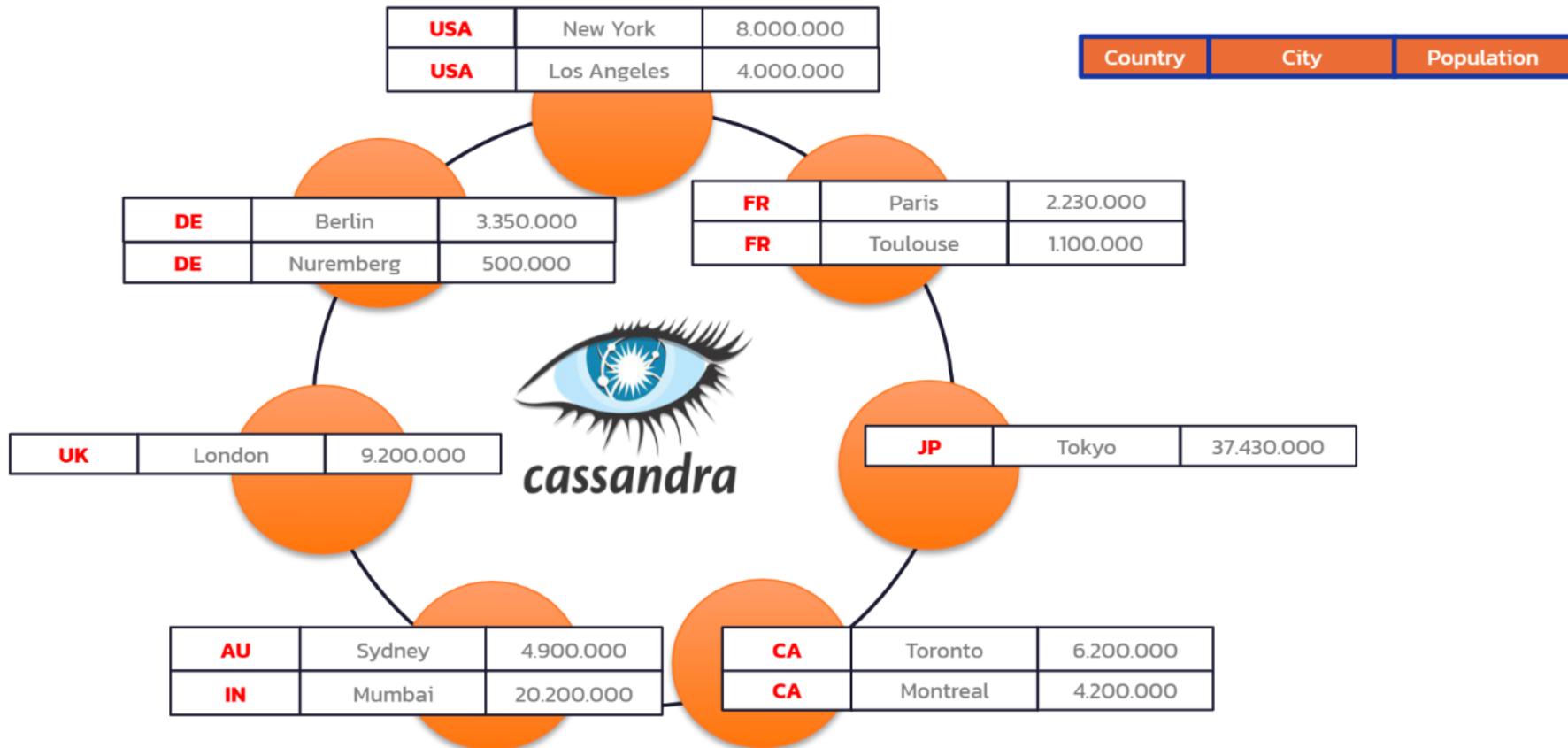
Data Distributed



Country	City	Population
USA	New York	8.000.000
USA	Los Angeles	4.000.000
FR	Paris	2.230.000
DE	Berlin	3.350.000
UK	London	9.200.000
AU	Sydney	4.900.000
DE	Nuremberg	500.000
CA	Toronto	6.200.000
CA	Montreal	4.200.000
FR	Toulouse	1.100.000
JP	Tokyo	37.430.000
IN	Mumbai	20.200.000


Partition Key

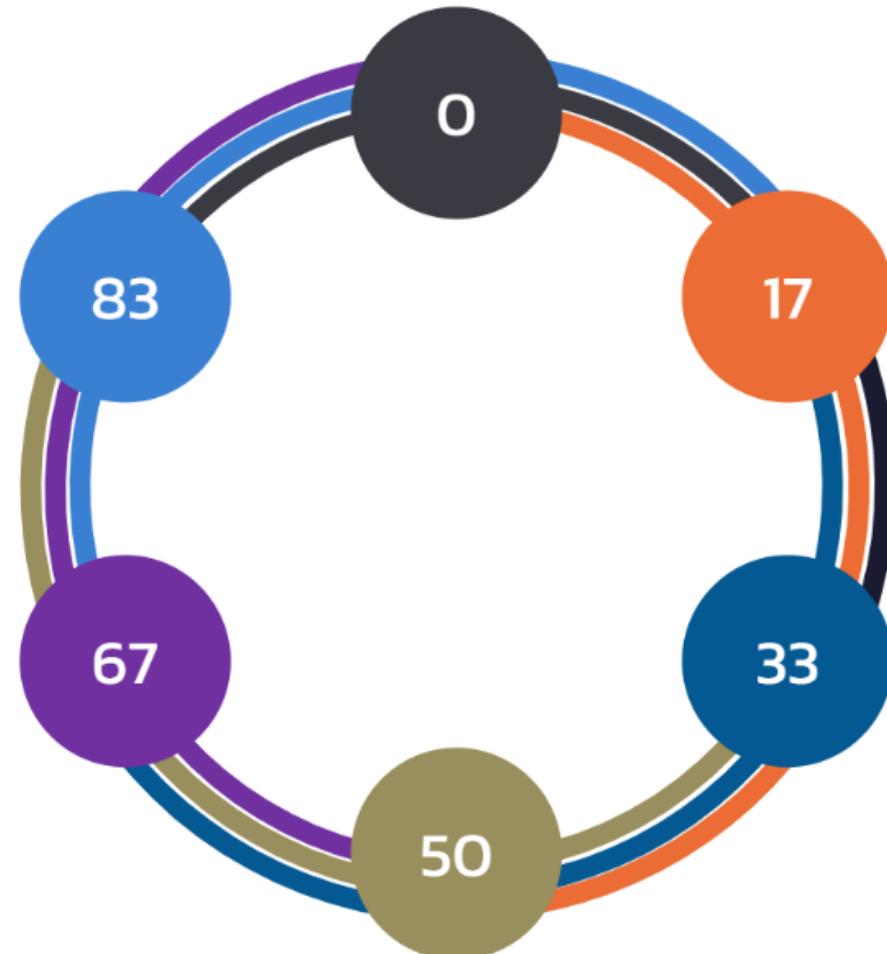
Data Distributed



Data is Replicated

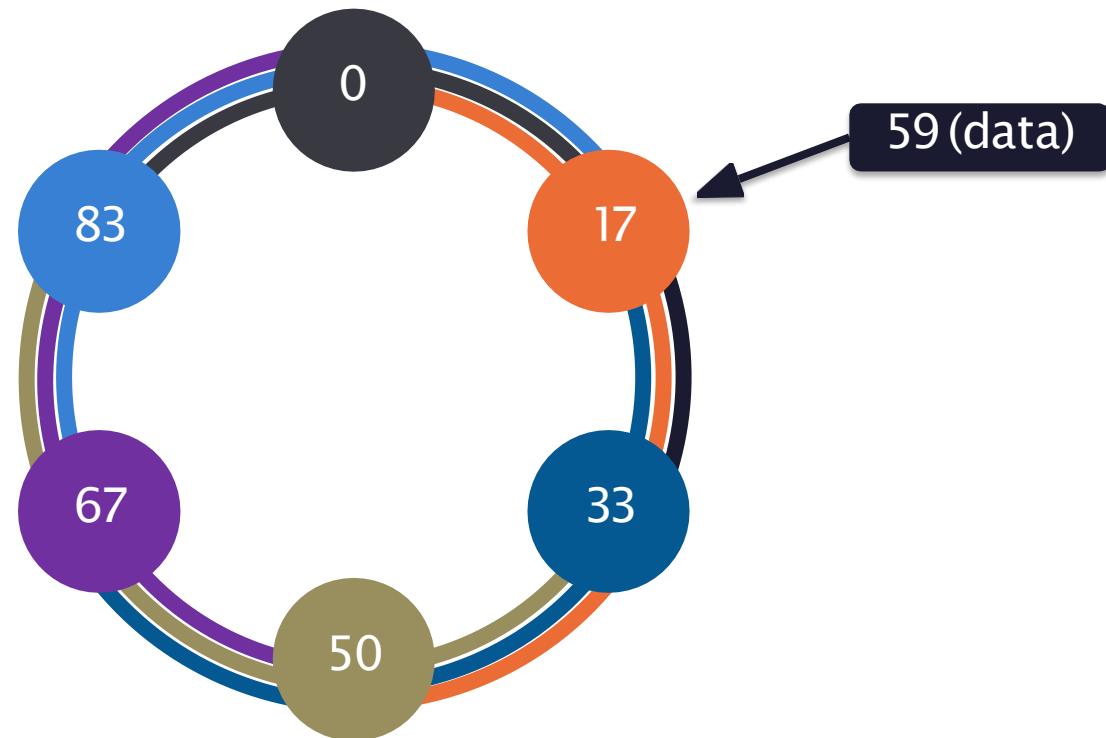
RF = 3

Replication Factor 3
means that every
row is stored on 3
different nodes



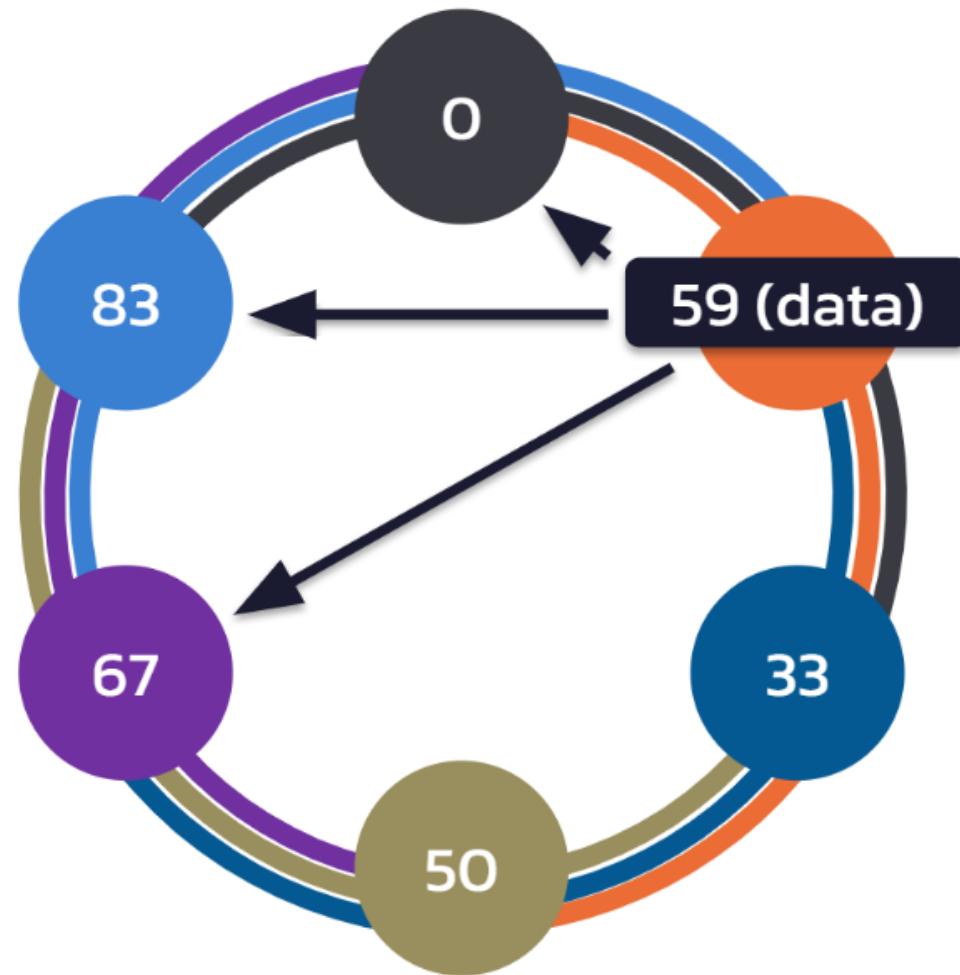
Replication within the Ring

RF = 3



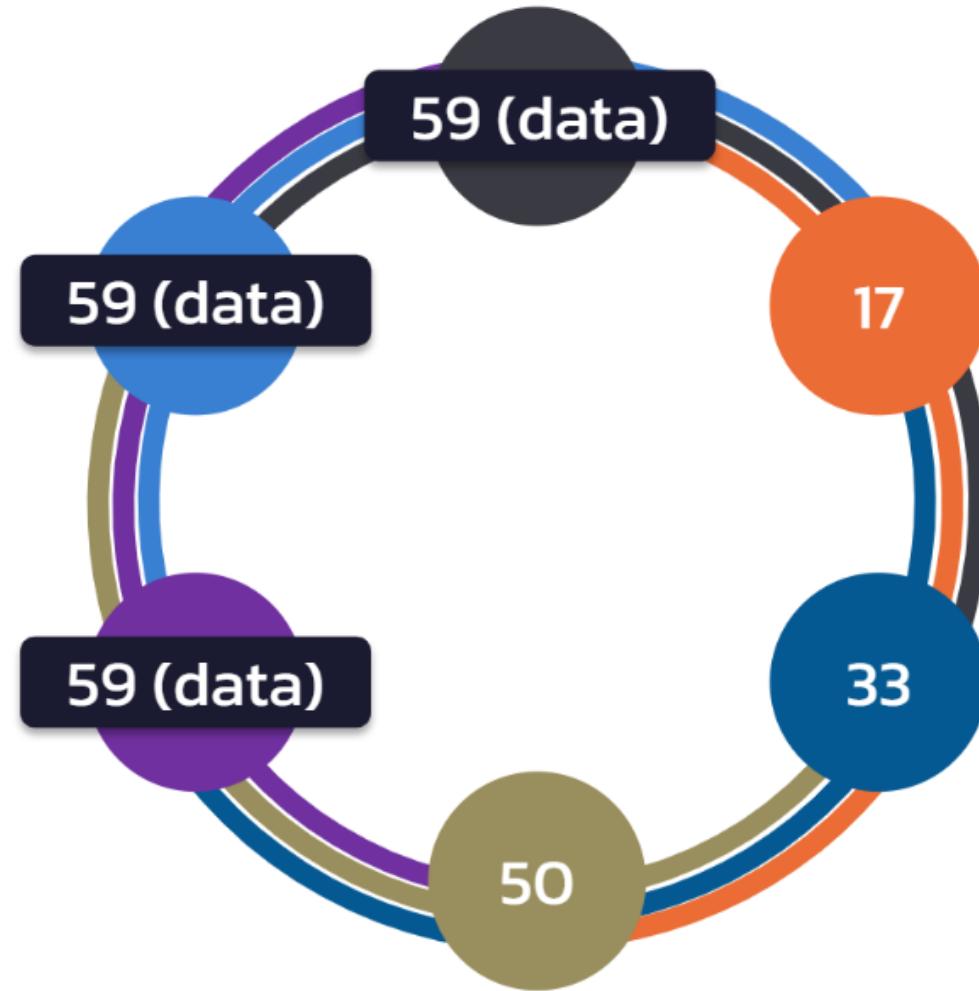
Replication within the Ring

RF = 3



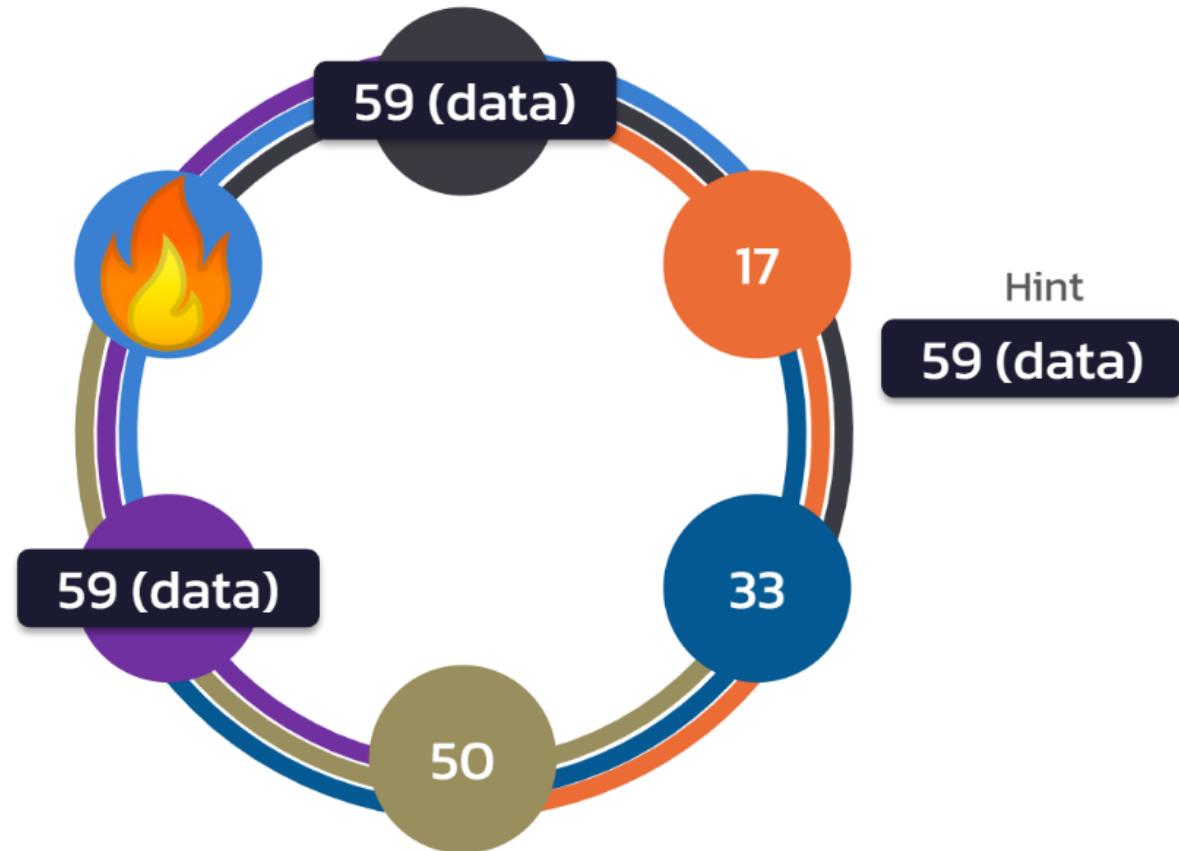
Replication within the Ring

RF = 3



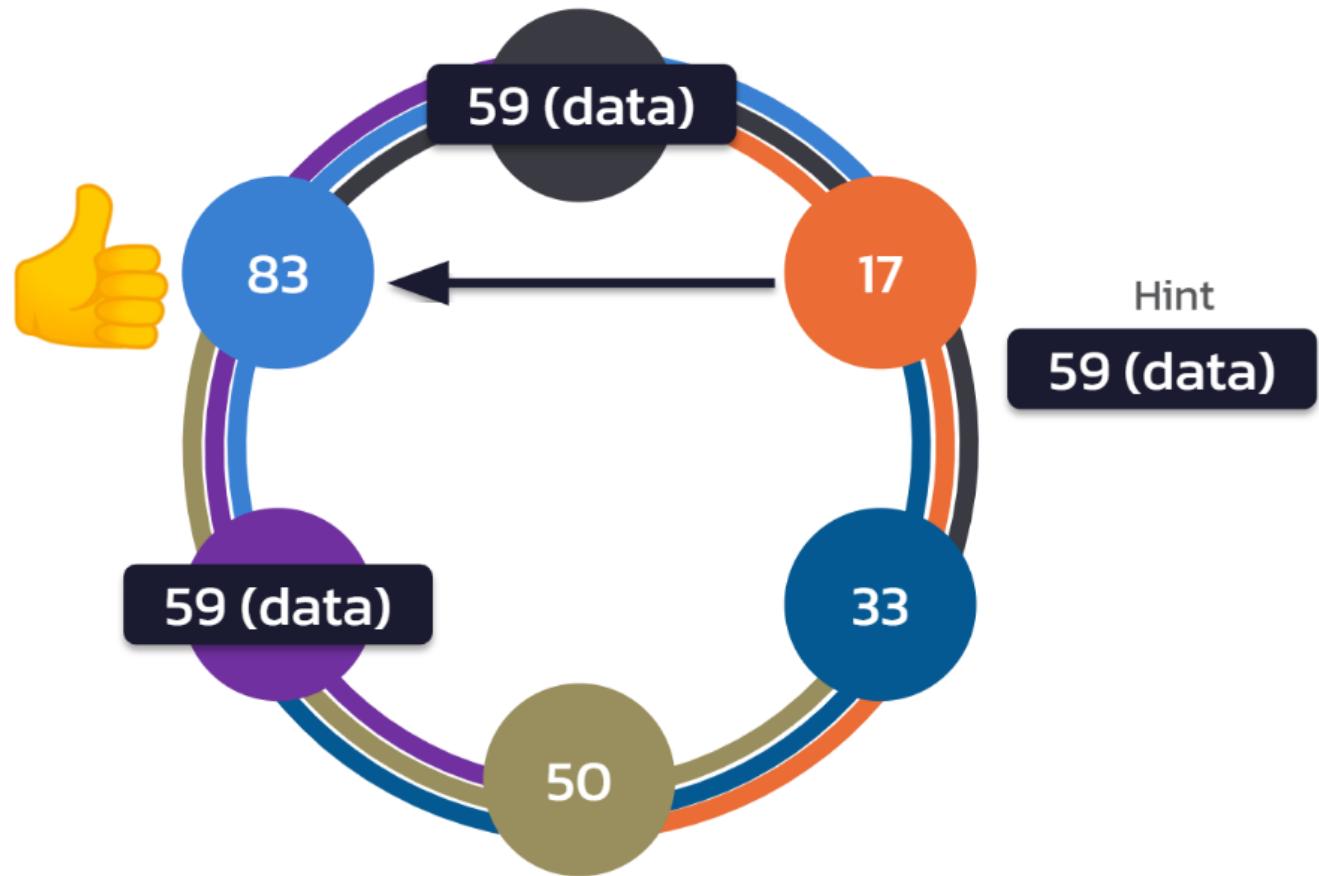
Node Failure

RF = 3

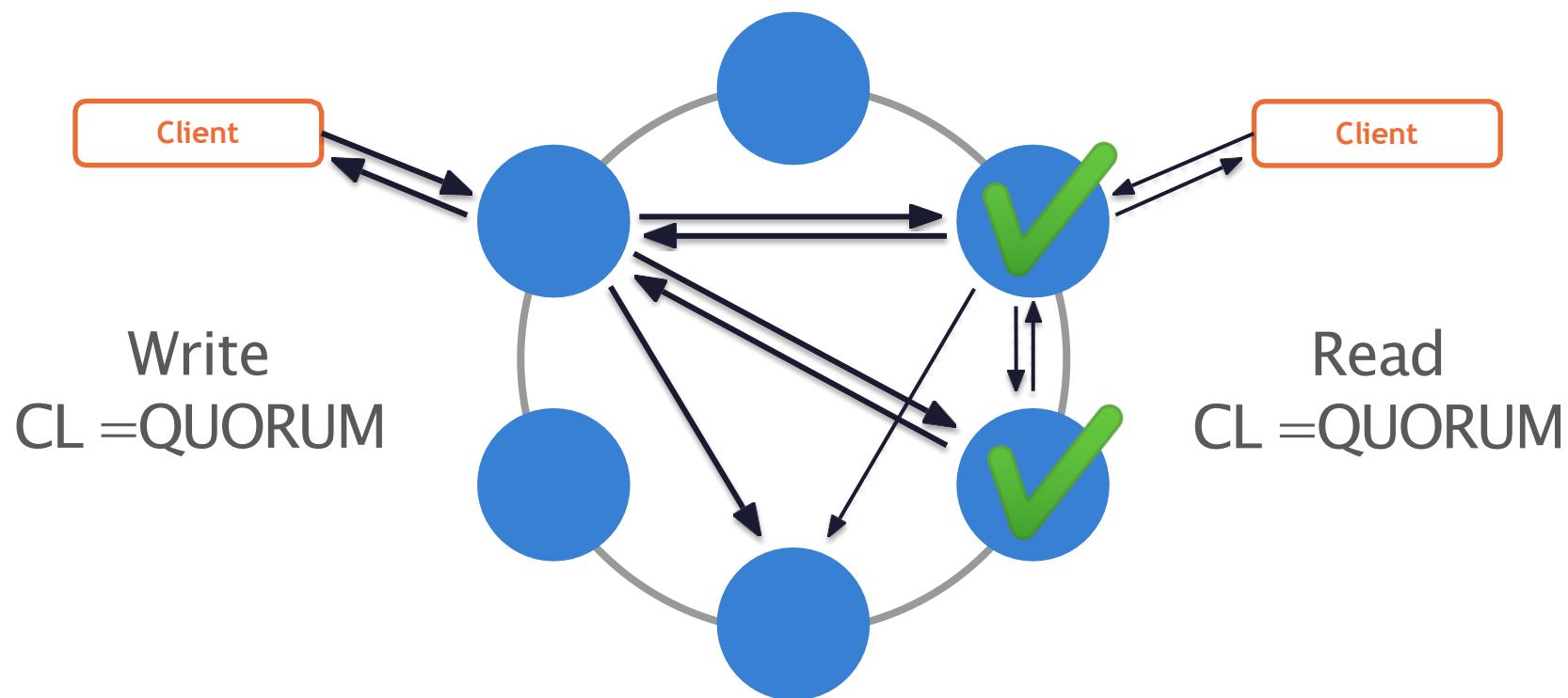


Node Failure Recovered

RF = 3



Immediate Consistency – A Better Way



Data Structure: a Cell

An intersection of a row and a column, stores data.

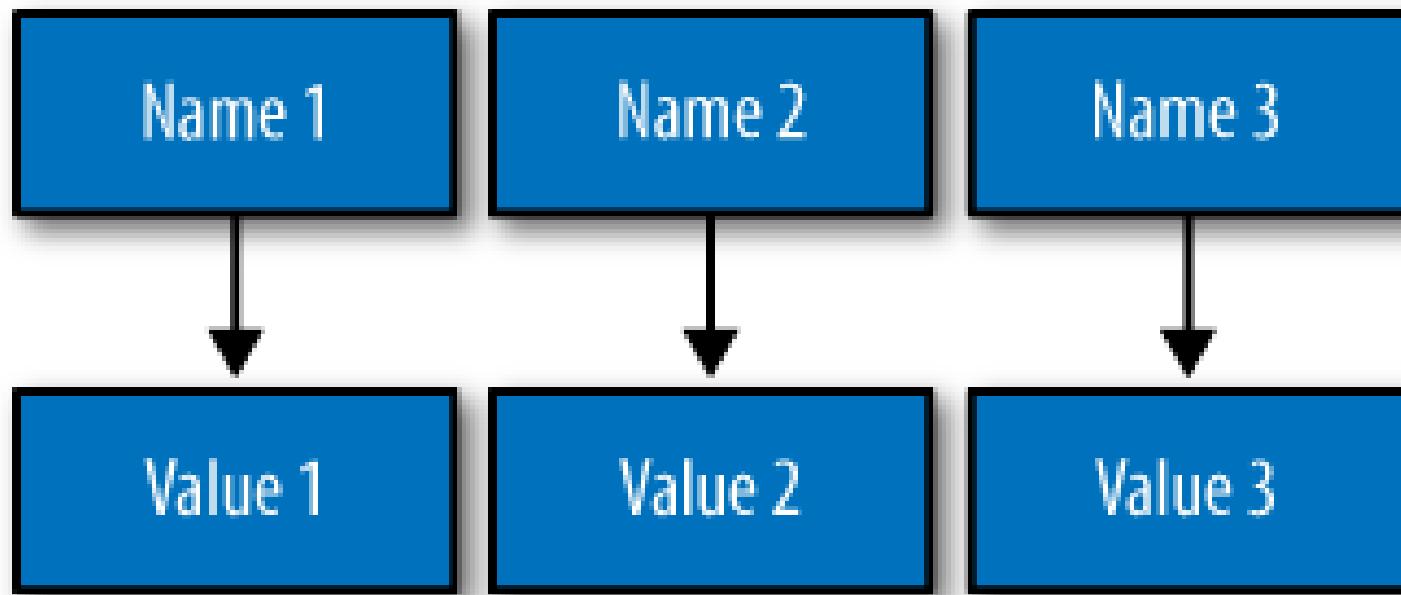


A list of values

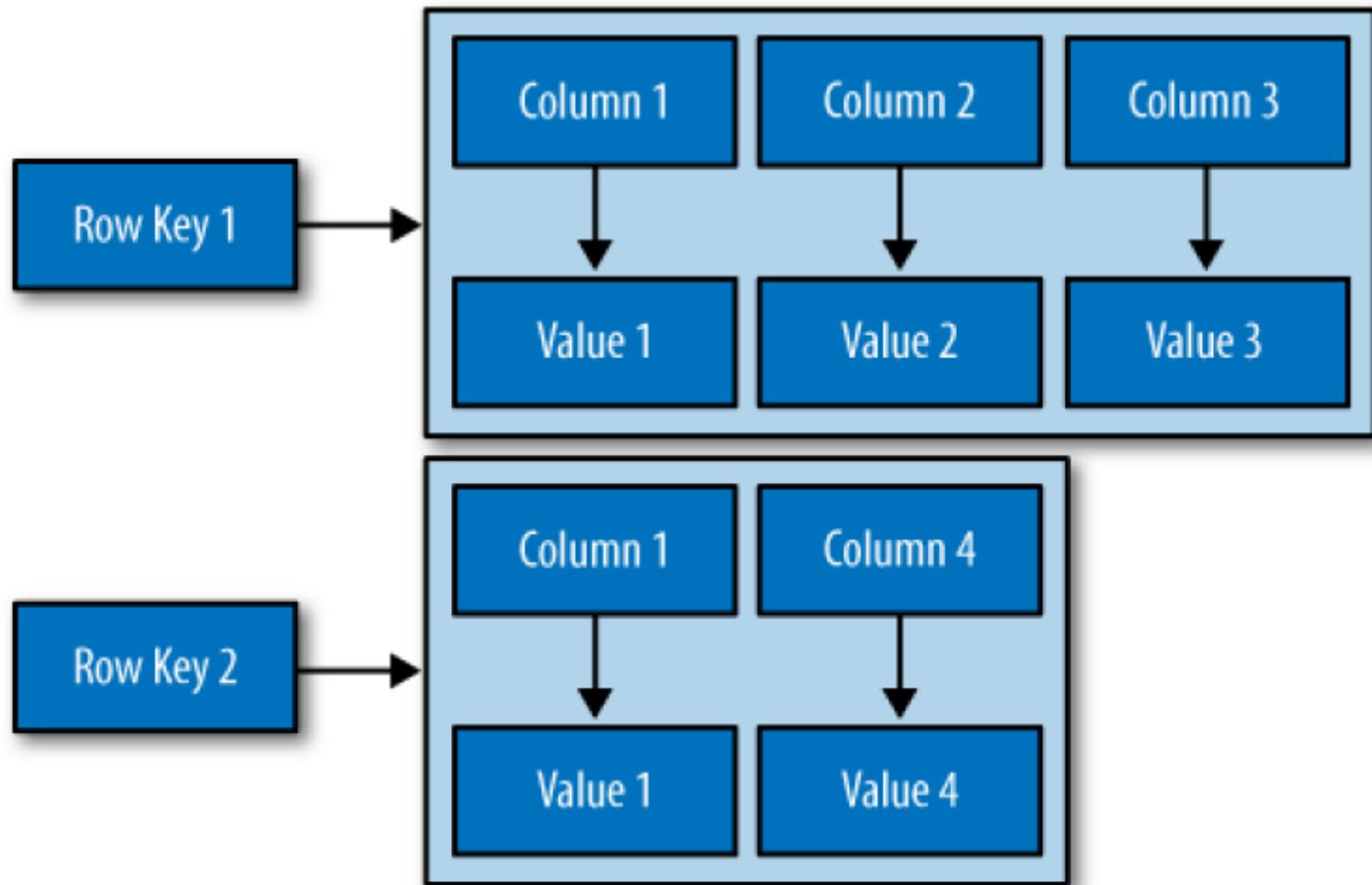


Data Structure: a Cell

A map of name/value pairs



Column Family

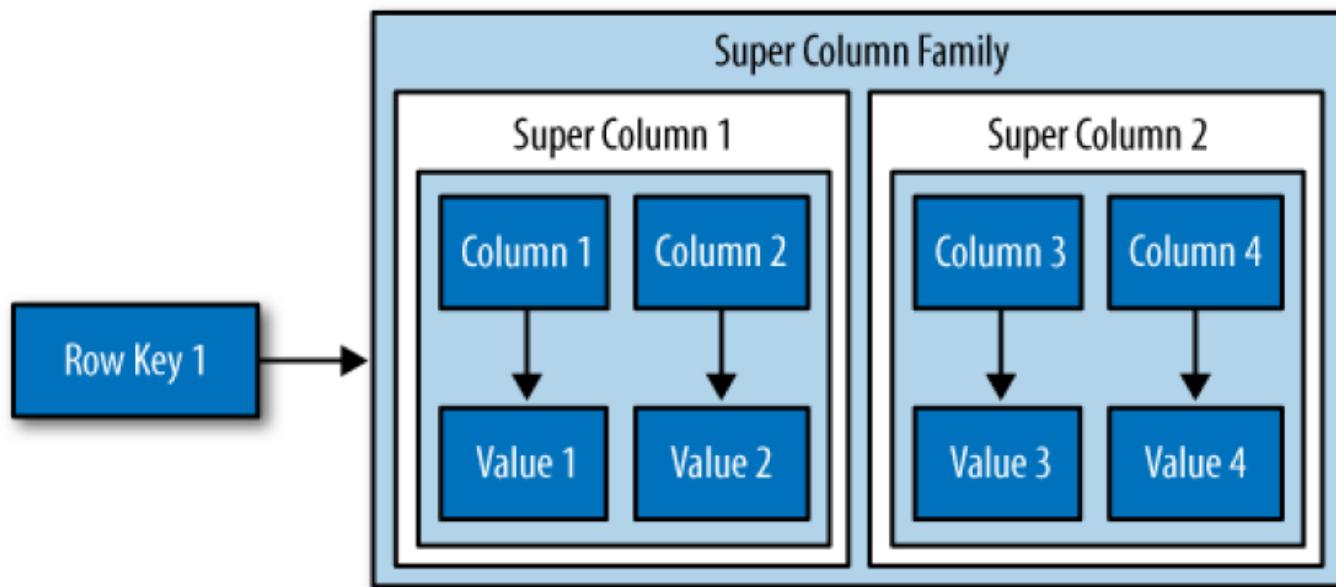


Column Family

It may help to think of it in terms of JavaScript Object Notation (JSON) instead of a picture:

Musician:	ColumnFamily 1
bootsy:	RowKey
email: bootsy@pfunk.com,	ColumnName:Value
instrument: bass	ColumnName:Value
george:	RowKey
email: george@pfunk.com	ColumnName:Value
Band:	ColumnFamily 2
george:	RowKey
pfunk: 1968-2010	ColumnName:Value

Column Family



Column Family

- A column family is a container for an ordered collection of rows, each of which is itself an ordered collection of columns.
- In the relational world, when we are physically creating database from a model, we specify the name of the database (key space), the names of the tables (remotely like column families, but don't get stuck on the idea that column families equal tables—they don't).
- Then you define the names of the columns that will be in each table.

Column Family

- First, Cassandra is considered schema-free because although the column families are defined, the columns are not.
- We can freely add any column to any column family at any time, depending on our needs.
- Second, a column family has two attributes: a name and a comparator.
- The comparator value indicates how columns will be sorted when they are returned to us in a query—according to long, byte, UTF8, or other ordering.

Column Family

- When you write data to a column family in Cassandra, we specify values for one or more columns.
- That collection of values together with a unique identifier is called a row.
- That row has a unique key, called the row key, which acts like the primary key unique identifier for that row.
- So, while it's not incorrect to call it column-oriented, or columnar, it might be easier to understand the model if we think of rows as containers for columns.

Column Family

- [Keyspace][ColumnFamily][Key][Column]
- We can use a JSON-like notation to represent a Hotel column family, as shown here:

```
Hotel {
```

```
key: AZC_043 { name: Cambria Suites Hayden, phone: 480-444-4444,  
address: 400 N. Hayden Rd., city: Scottsdale, state: AZ, zip: 85255}
```

```
key: AZS_011 { name: Clarion Scottsdale Peak, phone: 480-333-3333,  
address: 3000 N. Scottsdale Rd, city: Scottsdale, state: AZ, zip: 85255}
```

```
key: CAS_021 { name: W Hotel, phone: 415-222-2222,  
address: 181 3rd Street, city: San Francisco, state: CA, zip: 94103}
```

```
key: NYN_042 { name: Waldorf Hotel, phone: 212-555-5555,  
address: 301 Park Ave, city: New York, state: NY, zip: 10019}
```

```
}
```

Column Family Options

- **keys_cached**
 - The number of locations to keep cached per SSTable. This doesn't refer to column name/values at all, but to the number of keys, as locations of rows per column family, to keep in memory in least-recently-used order.
- **rows_cached**
 - The number of rows whose entire contents (the complete list of name/value pairs for that unique row key) will be cached in memory.
- **comment**
 - This is just a standard comment that helps you remember important things about your column family definitions.

Column Family Options

- **read_repair_chance**
 - This is a value between 0 and 1 that represents the probability that read repair operations will be performed when a query is performed without a specified quorum, and it returns the same row from two or more replicas and at least one of the replicas appears to be out of date.
 - You may want to lower this value if you are performing a much larger number of reads than writes.
- **preload_row_cache**
 - Specifies whether you want to prepopulate the row cache on server startup.

Column

- A column is the most basic unit of data structure in the Cassandra data model.
- A column is a triplet of a name, a value, and a clock, which you can think of as a timestamp for now.
- Here's an example of a column you might define, represented with JSON notation just for clarity of structure:

```
{  
  "name": "email",  
  "value": "me@example.com",  
  "timestamp": 1274654183103300  
}
```



Wide Rows, Skinny Rows

- When designing a table in a traditional relational database, we're typically dealing with “entities” or the set of attributes that describe a particular noun (Hotel, User, Product, etc.).
- A wide row means a row that has lots and lots (perhaps tens of thousands or even millions) of columns.
- Typically, there is a small number of rows that go along with so many columns.
- Conversely, we could have something closer to a relational model, where we define a smaller number of columns and use many different rows—that's the skinny model.

Wide Rows, Skinny Rows

- Wide rows typically contain automatically generated names (like UUIDs or timestamps) and are used to store lists of things.
- Consider a monitoring application as an example: we might have a row that represents a time slice of an hour by using a modified timestamp as a row key, and then store columns representing IP addresses that accessed your application within that interval.
- We can then create a new row key after an hour elapses.

Column Sorting

- Columns have another aspect to their definition.
- In Cassandra, we specify how column names will be compared for sort order when results are returned to the client.
- Columns are sorted by the “Compare With” type defined on their enclosing column family, and we can choose from the following: AsciiType, BytesType, LexicalUUIDType, Integer Type, LongType, TimeUUIDType, or UTF8Type.

Column Sorting

- AsciiType
 - This sorts by directly comparing the bytes, validating that the input can be parsed as US-ASCII. US-ASCII is a character encoding mechanism based on the lexical order of the English alphabet. It defines 128 characters, 94 of which are printable.
- BytesType
 - This is the default, and sorts by directly comparing the bytes, skipping the validation step.
 - BytesType is the default for a reason: it provides the correct sorting for most types of data (UTF-8 and ASCII included).

Column Sorting

- LexicalUUIDType
 - A 16-byte (128-bit) Universally Unique Identifier (UUID), compared lexically (by byte value).
- LongType
 - This sorts by an 8-byte (64-bit) long numeric type.
- IntegerType
 - Introduced in 0.7, this is faster than LongType and allows integers of both fewer and more bits than the 64 bits provided by LongType.

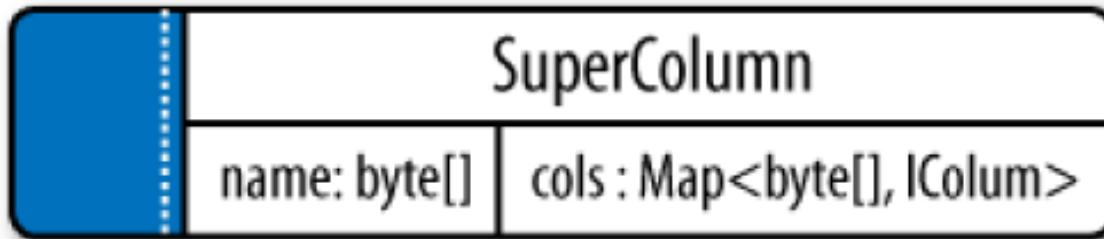
Column Sorting

- TimeUUIDType
 - This sorts by a 16-byte (128-bit) timestamp.
 - There are five common versions of generating timestamp UUIDs.
 - The scheme Cassandra uses is a version one UUID, which means that it is generated based on conflating the computer's MAC address and the number of 100-nanosecond intervals since the beginning of the Gregorian calendar.
- UTF8Type
 - A string using UTF-8 as the character encoder.
 - Although this may seem like a good default type, that's probably because it's comfortable to programmers who are used to using XML or other data exchange mechanism that requires common encoding.
 - In Cassandra, however, you should use UTF8Type only if you want your data validated.

Super Columns

- A super column is a special kind of column. Both kinds of columns are name/value pairs, but a regular column stores a byte array value, and the value of a super column is a map of subcolumns (which store byte array values).
- Note that they store only a map of columns; we cannot define a super column that stores a map of other super columns.
- So, the super column idea goes only one level deep, but it can have an unbounded number of columns.
- The basic structure of a super column is its name, which is a byte array (just as with a regular column), and the columns it stores (see Figure 3-6).
- Its columns are held as a map whose keys are the column names and whose values are the columns.

Super Columns



- Each column family is stored on disk in its own separate file.
- So, to optimize performance, it's important to keep columns that you are likely to query together in the same column family, and a super column can be helpful for this.

Super Columns

- The SuperColumn class implements both the IColumn and the IColumnContainer classes, both from the org.apache.cassandra.db package.
- The Thrift API is the underlying RPC serialization mechanism for performing remote operations on Cassandra.
- Because the Thrift API has no notion of inheritance, we will sometimes see the API refer to a Column Or Supercolumn type; when data structures use this type, you are expected to know whether your underlying column family is of type Super or Standard.



Super Columns

- Cassandra looks like a four-dimensional hashtable.
- But for super columns, it becomes more like a five-dimensional hash:
- [Keyspace][ColumnFamily][Key][SuperColumn][SubColumn]
- To use a super column, you define your column family as type Super.
- Then, you still have row keys as you do in a regular column family, but you also reference the super column, which is simply a name that points to a list or map of regular columns (sometimes called the subcolumns).

Super Columns

PointOfInterest (SCF)

SCkey: Cambria Suites Hayden

{

key: Phoenix Zoo

{

phone: 480-555-9999,

desc: They have animals here.

},

Super Columns

```
key: Spring Training
{
  phone: 623-333-3333,
  desc: Fun for baseball fans.
},
}, //end of Cambria row
SCkey: (UTF8) Waldorf=Astoria
{
  key: Central Park
  desc: Walk around. It's pretty.
},
```

Super Columns

key: Empire State Building

{

phone: 212-777-7777,

desc: Great view from the 102nd floor.

54 | Chapter 3: The Cassandra Data Model

}

}

}

Cluster

- A cluster is a container for key spaces—typically a single key space.
- A key space is the outermost container for data in Cassandra, corresponding closely to a relational database.
- Like a relational database, a key space has a name and a set of attributes that define keyspace-wide behavior.

Cluster

- Even though it's a good idea to create a single key space per application, this doesn't appear to have much practical basis.
- It's certainly an acceptable practice, but it's perfectly fine to create as many key spaces as your application needs.
- Note, we will probably run into trouble creating thousands of key spaces per application



Data Structure: a Row

A single, structured data item in a table.

1	John	Doe	Wizardry
---	------	-----	----------

Data Structure: a Partition

A group of rows having the same partition token, a base unit of access in Cassandra.

IMPORTANT: stored together, all the rows are guaranteed to be neighbors.

ID	First Name	Last Name	Department
1	John	Doe	Wizardry
399	Marisha	Chavez	Wizardry
415	Maximus	Flavius	Wizardry

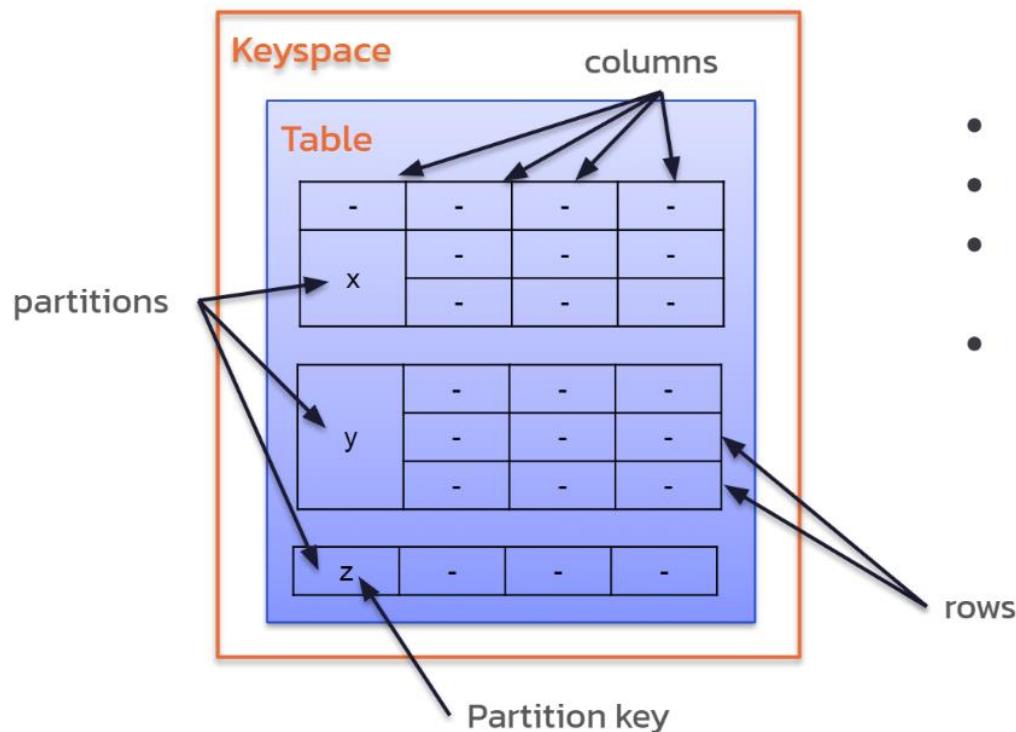


Data Structure: a Table

A group of columns and rows storing partitions.

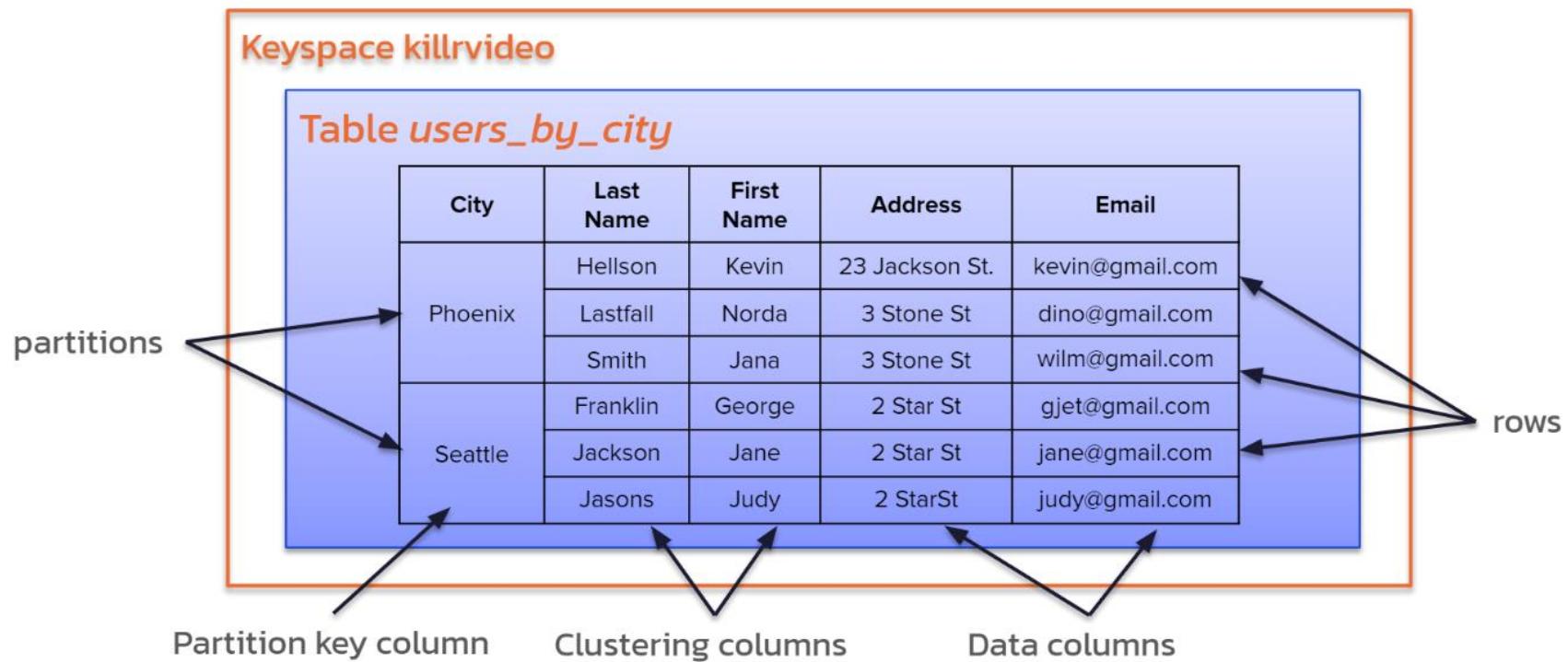
ID	First Name	Last Name	Department
1	John	Doe	Wizardry
2	Mary	Smith	Dark Magic
3	Patrick	McFadin	DevRel

Data Structure: Overall

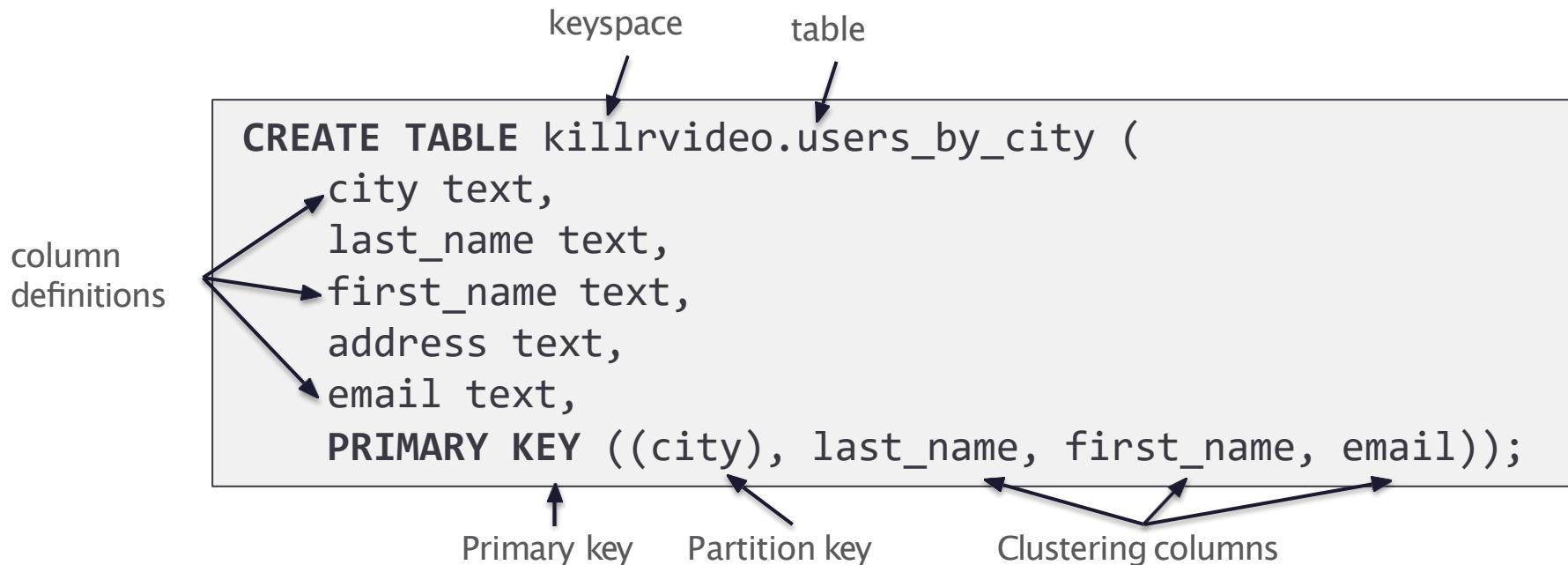


- Tabular data model, with one twist
- *Tables* are organized in *rows* and *columns*
- Groups of related rows called *partitions* are stored together on the same node (or nodes)
- Each row contains a *partition key*
 - One or more columns that are hashed to determine which node(s) store that data

Example Data: Users Organized By City



Creating a Table in CQL



Primary Key

An identifier for a row. Consists of at least one Partition Key and zero or more Clustering Columns.

MUST ENSURE UNIQUENESS.
MAY DEFINE SORTING.

```
CREATE TABLE killrvideo.users_by_city (
    city text,
    last_name text,
    first_name text,
    address text,
    email text,
    PRIMARY KEY ((city), last_name, first_name, email));
```

Partition key

Clustering columns

Good Examples:

```
PRIMARY KEY ((city), last_name, first_name, email);
```

```
PRIMARY KEY (user_id);
```

Bad Example:

```
PRIMARY KEY ((city), last_name, first_name);
```

Partition Key

An identifier for a partition.
Consists of at least one column,
may have more if needed

PARTITIONS ROWS.

```
CREATE TABLE killrvideo.users_by_city (
    city text,
    last_name text,
    first_name text,
    address text,
    email text,
    PRIMARY KEY ((city), last_name, first_name, email));
```



Good Examples:

```
PRIMARY KEY (user_id);
```

```
PRIMARY KEY ((video_id), comment_id);
```

Bad Example:

```
PRIMARY KEY ((sensor_id), logged_at);
```

Clustering Column(s)

Used to ensure uniqueness and sorting order. Optional.

```
CREATE TABLE killrvideo.users_by_city (
    city text,
    last_name text,
    first_name text,
    address text,
    email text,
    PRIMARY KEY ((city), last_name, first_name, email));
```

Partition key

Clustering columns

`PRIMARY KEY ((city), last_name, first_name);`



Not Unique

`PRIMARY KEY ((city), last_name, first_name, email);`



`PRIMARY KEY ((video_id), comment_id);`



Not Sorted

`PRIMARY KEY ((video_id), created_at, comment_id);`



Rules of a Good Partition

- **Store together what you retrieve together**
- Avoid big partitions
- Avoid hot partitions

Example: open a video? Get the comments in a single query!

```
PRIMARY KEY ((video_id), created_at, comment_id);
```



```
PRIMARY KEY ((comment_id), created_at);
```



Rules of a Good Partition

- Store together what you retrieve together
- **Avoid big partitions**
- Avoid hot partitions

```
PRIMARY KEY ((video_id), created_at, comment_id);
```



```
PRIMARY KEY ((country), user_id);
```



- Up to 2 billion cells per partition
- Up to ~100k rows in a partition
- Up to ~100MB in a Partition

Rules of a Good Partition

- Store together what you retrieve together
- **Avoid big and constantly growing partitions!**
- Avoid hot partitions

Example: a huge IoT infrastructure, hardware all over the world, different sensors reporting their state every 10 seconds. Every sensor reports its UUID, timestamp of the report, sensor's value.

```
PRIMARY KEY ((sensor_id), reported_at);
```



- Sensor ID: UUID
- Timestamp: Timestamp
- Value: float

Rules of a Good Partition

- Store together what you retrieve together
- **Avoid big and constantly growing partitions!**
- Avoid hot partitions

Example: a huge IoT infrastructure, hardware all over the world, different sensors reporting their state every 10 seconds. Every sensor reports its UUID, timestamp of the report, sensor's value.

```
PRIMARY KEY ((sensor_id), reported_at);
```



```
PRIMARY KEY ((sensor_id, month_year), reported_at);
```



BUCKETING

- Sensor ID: UUID
- **MonthYear:** Integer or String
- Timestamp: Timestamp
- Value: float

Rules of a Good Partition

- Store together what you retrieve together
- Avoid big partitions
- **Avoid hot partitions**

```
PRIMARY KEY (user_id);
```



```
PRIMARY KEY ((video_id), created_at, comment_id);
```



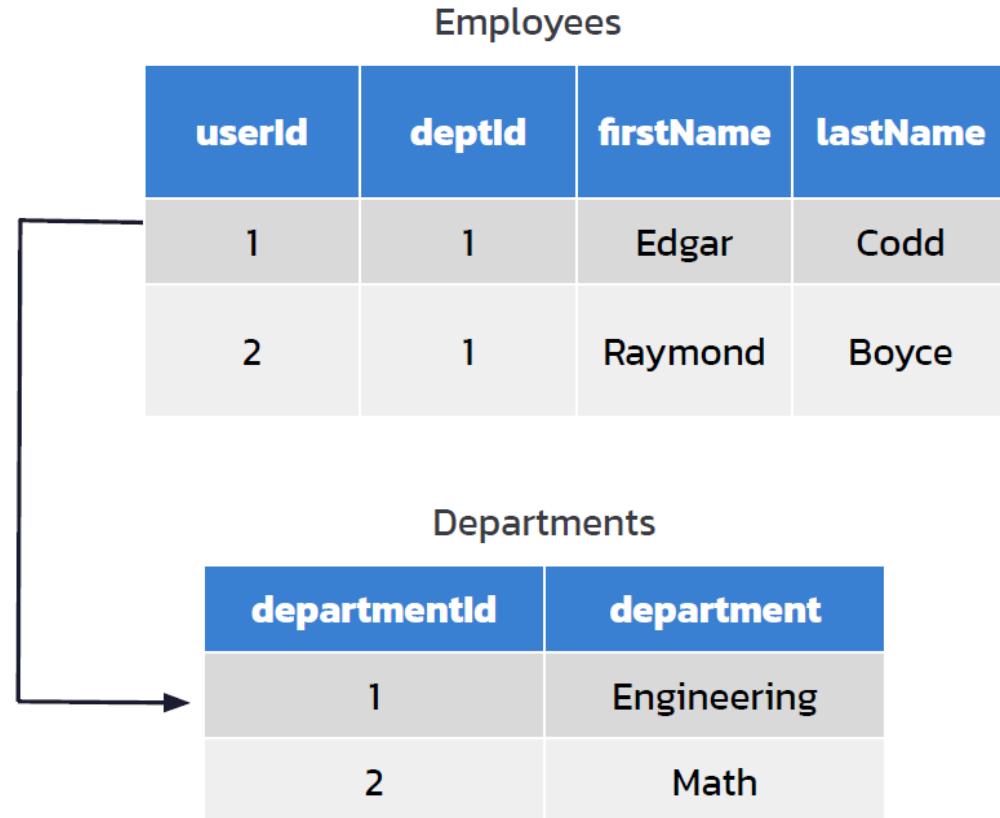
```
PRIMARY KEY ((country), user_id);
```



Normalization

"Database normalization is the process of structuring a relational database in accordance with a series of so-called normal forms in order to reduce data redundancy and improve data integrity. It was first proposed by Edgar F. Codd as part of his relational model."

PROS: Simple write, Data Integrity
CONS: Slow read, Complex Queries



Employees

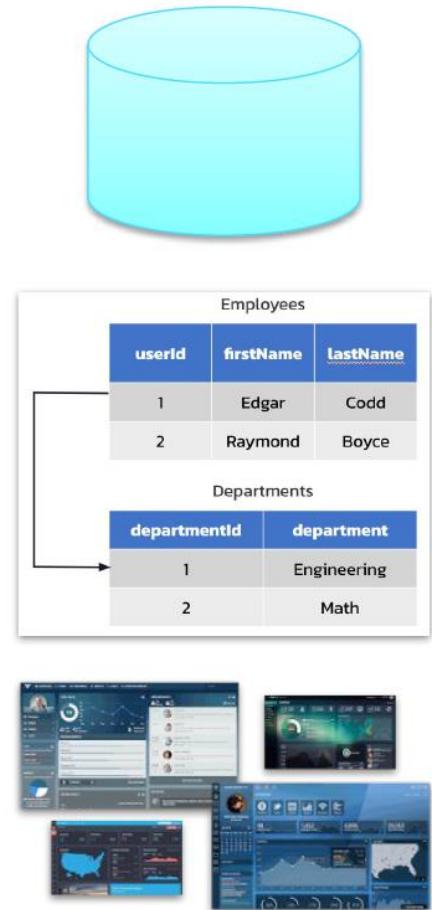
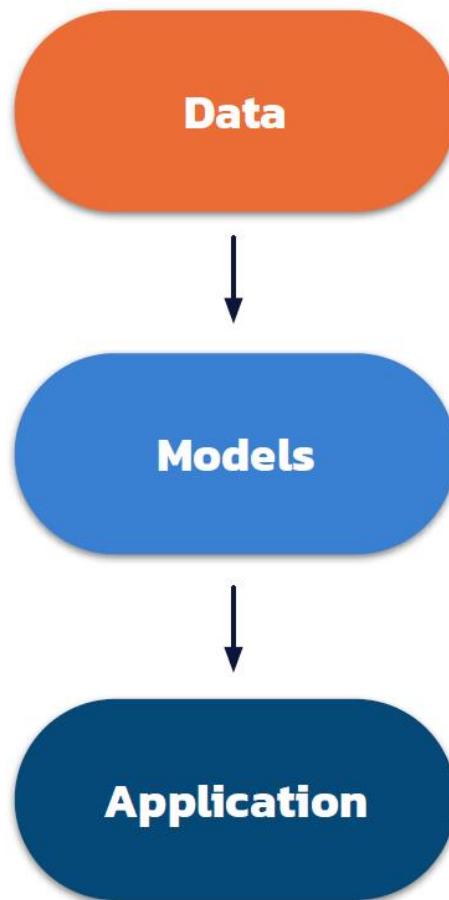
userId	deptId	firstName	lastName
1	1	Edgar	Codd
2	1	Raymond	Boyce

Departments

departmentId	department
1	Engineering
2	Math

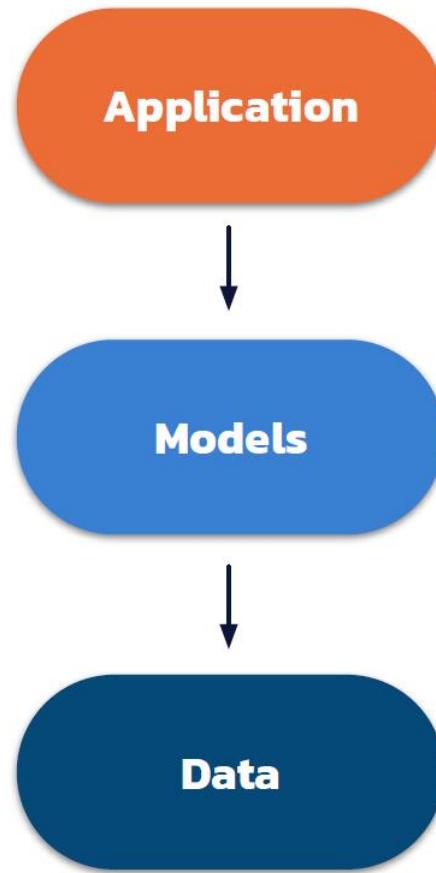
Relational Data Modeling

1. Analyze raw data
2. Identify entities, their properties and relations
3. Design tables, using **normalization** and foreign keys.
4. Use JOIN when doing queries to join normalized data from multiple tables

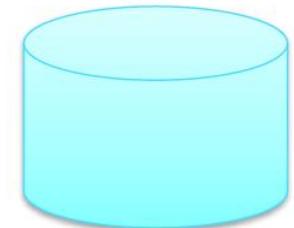


NoSQL Data Modelling

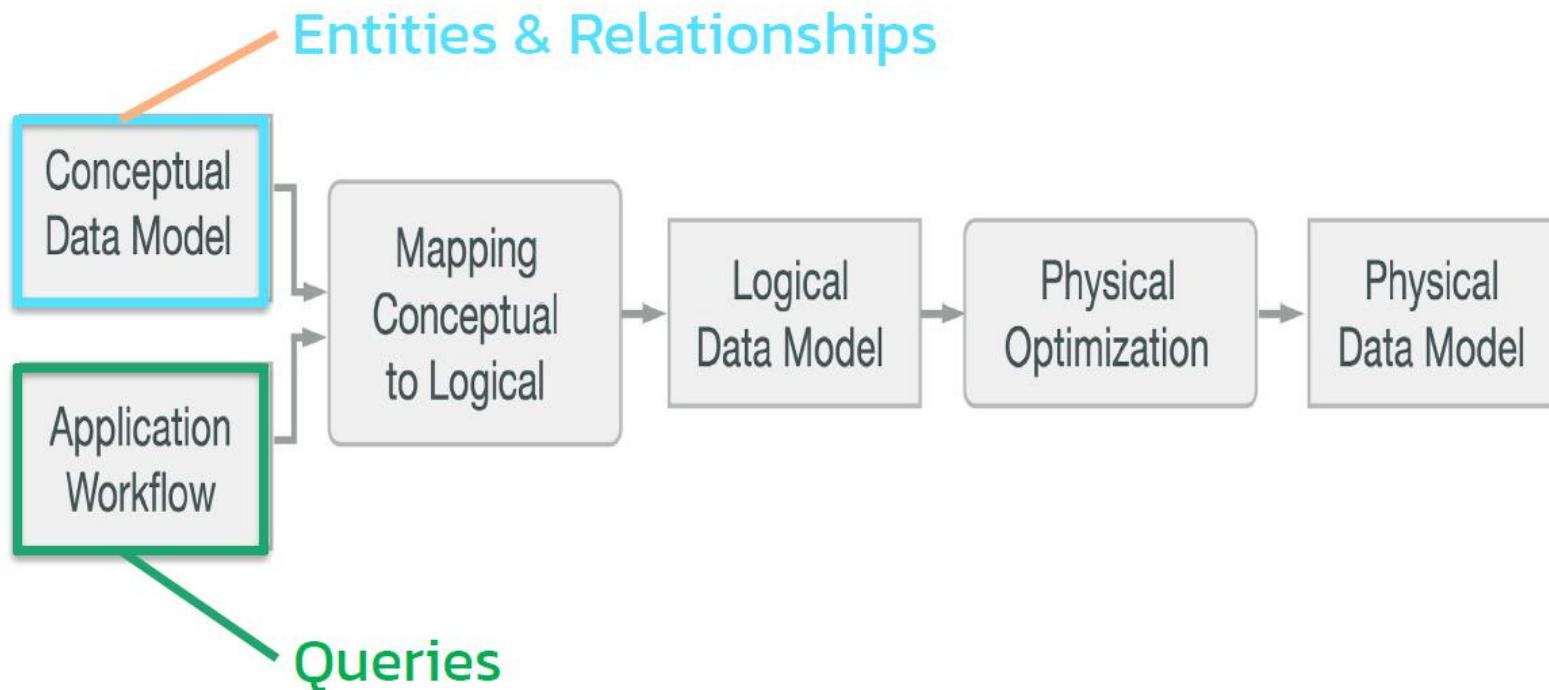
1. Analyze user behaviour
(customer first!)
2. Identify workflows, their dependencies and needs
3. Define Queries to fulfill these workflows
4. Knowing the queries, design tables, using **denormalization**.
5. Use BATCH when inserting or updating denormalized data of multiple tables



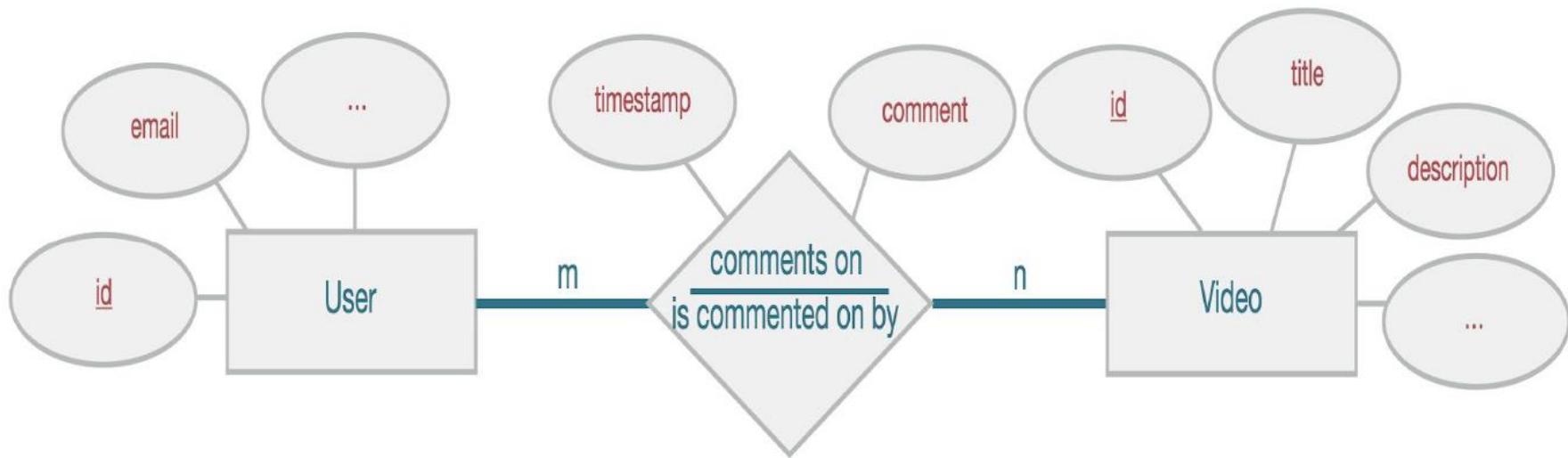
Employees			
userId	firstName	lastName	department
1	Edgar	Codd	Engineering
2	Raymond	Boyce	Math
3	Sage	Lahja	Math
4	Juniper	Jones	Botany



Designing Process Step By Step



Designing Process Conceptual Data Model





Designing Process Application Work Flow

Use-Case I:

- A User opens a Profile

WF2: Find comments related to target user using its identifier, get most recent first

Use-Case II:

- A User opens a Video Page

WF1: Find comments related to target video using its identifier, most recent first



Designing Process Physical Data Model

comments_by_user		
userid	UUID	K
commentid	TIMEUUID	C↓
videoid	UUID	
comment	TEXT	

comments_by_video		
videoid	UUID	K
commentid	TIMEUUID	C↓
userid	UUID	
comment	TEXT	

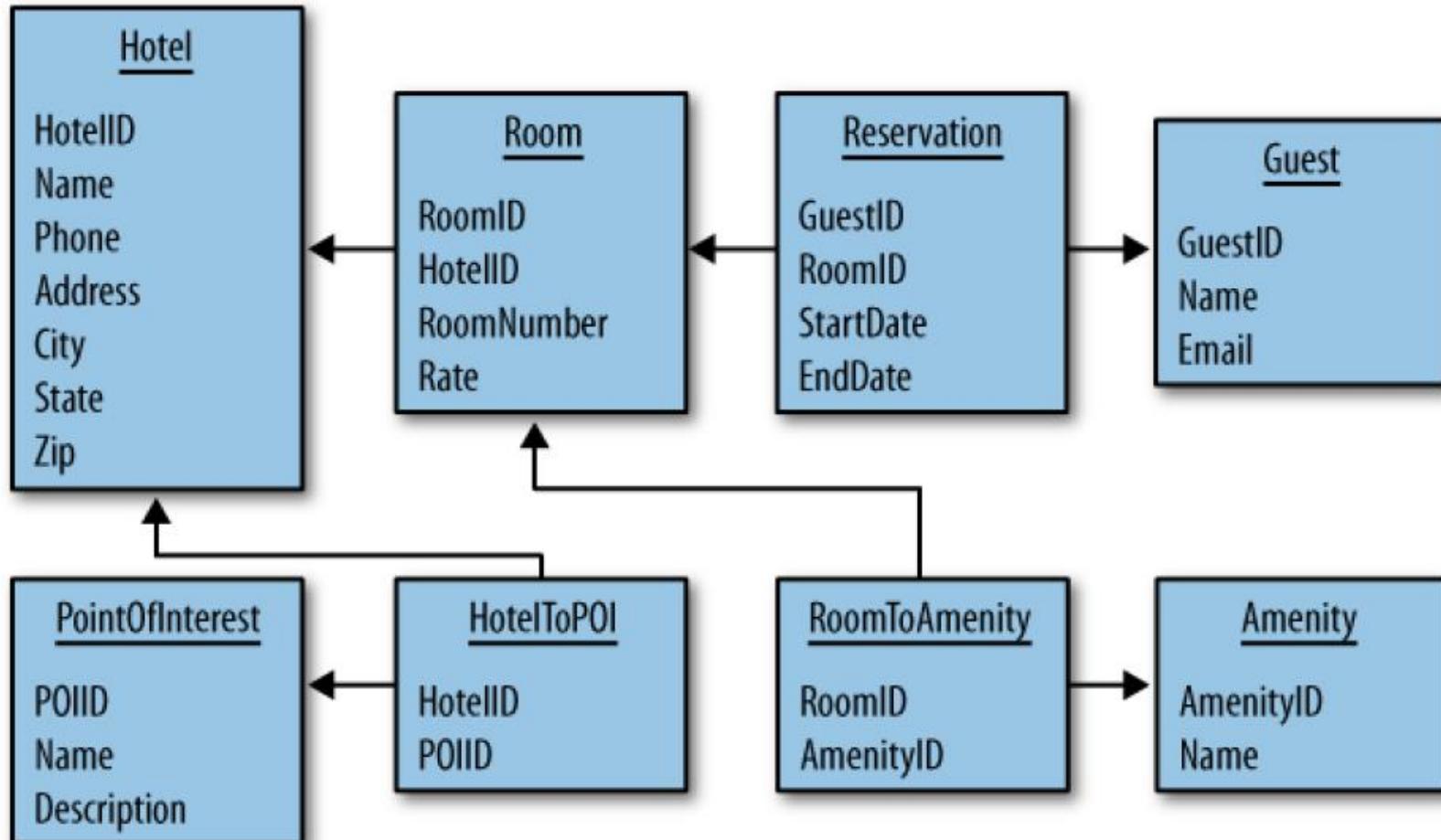


Designing Process Schema DDL

```
CREATE TABLE IF NOT EXISTS comments_by_user (
    userid uuid,
    commentid timeuuid,
    videoid uuid,
    comment text,
    PRIMARY KEY ((userid), commentid)
) WITH CLUSTERING ORDER BY (commentid DESC);
```

```
CREATE TABLE IF NOT EXISTS comments_by_video (
    videoid uuid,
    commentid timeuuid,
    userid uuid,
    comment text,
    PRIMARY KEY ((videoid), commentid)
) WITH CLUSTERING ORDER BY (commentid DESC);
```

Use Case



Use Case

<<CF>>Hotel	<<CF>>HotelByCity	<<SCF>>PointOfInterest
<<RowKey>>#hotelID +name +phone +address +city +state +zip	<<RowKey>>#city:state:hotelID +hotel1 +hotel2 +...	<<SuperColumnName>>#hotelID <<RowKey>> #poiName +desc +Phone
<<CF>>Guest	<<SCF>>RoomAvailability	<<SCF>>Room
<<RowKey>>#phone +fname +lname +email	<<SuperColumnName>>#hotelID <<RowKey>> +date +kk : <unspecified> = 22 +qq : <unspecified> = 14	<<SuperColumnName>>#hotelID <<RowKey>> #roomID +num +type +rate +coffee +tv +hottub +...
<<CF>>Reservation		
	<<RowKey>>#resID +hotelID +roomID +phone +name +arrive +depart +rate +ccNum	

Design Differences Between RDBMS and Cassandra



- No joins
 - We cannot perform joins in Cassandra.
 - If we have designed a data model and find that you need something like a join, we'll have to either do the work on the client side, or create a denormalized second table that represents the join results for you.
 - Cassandra data modeling. Performing joins on the client should be a very rare case;
 - we really want to duplicate (denormalize) the data instead.

Design Differences Between RDBMS and Cassandra



- No referential integrity
 - Cassandra itself has no concept of referential integrity across tables.
 - In a relational database, you could specify foreign keys in a table to reference the primary key of a record in another table.
 - But Cassandra does not enforce this.
 - It is still a common design requirement to store IDs related to other entities in your tables, but operations such as cascading deletes are not available.

Design Differences Between RDBMS and Cassandra



- Denormalization
 - In relational database design, we are often taught the importance of normalization.
 - This is not an advantage when working with Cassandra because it performs best when the data model is denormalized.
 - It is often the case that companies end up denormalizing data in relational databases as well.
 - There are two common reasons for this. One is performance.

Design Differences Between RDBMS and Cassandra



- Denormalization
 - Companies simply can't get the performance they need when they have to do so many joins on years' worth of data, so they denormalize along the lines of known queries.
 - This ends up working but goes against the grain of how relational databases are intended to be designed.
 - Ultimately makes one question whether using a relational database is the best approach in these circumstances..

Design Differences Between RDBMS and Cassandra



- Denormalization
 - A second reason that relational databases get denormalized on purpose is a business document structure that requires retention.
 - That is, we have an enclosing table that refers to a lot of external tables whose data could change over time, but you need to preserve the enclosing document as a snapshot in history.
 - The common example here is with invoices.
 - In customer and product tables, just make an invoice that refers to those tables.

Design Differences Between RDBMS and Cassandra



- Denormalization
 - Customer or price information could change, and then you would lose the integrity of the invoice document as it was on the invoice date, which could violate audits, reports, or laws, and cause other problems.
 - In the relational world, denormalization violates Codd's normal forms, and you try to avoid it.
 - But in Cassandra, denormalization is, well, perfectly normal.

Design Differences Between RDBMS and Cassandra



- Denormalization
 - Historically, denormalization in Cassandra has required designing and managing multiple tables using techniques described in this documentation.
 - Beginning with the 3.0 release, Cassandra provides a feature known as materialized views <materialized-views> which allows you to create multiple denormalized views of data based on a base table design.
 - Cassandra manages materialized views on the server, including the work of keeping the views in sync with the table.

Design Differences Between RDBMS and Cassandra



- Query-first design
 - Relational modeling, in simple terms, means that you start from the conceptual domain and then represent the nouns in the domain in tables.
 - You then assign primary keys and foreign keys to model relationships.
 - When we have a many-to-many relationship, we create the join tables that represent just those keys.
 - The join tables don't exist in the real world.
 - There are side effect of the way relational models work.

Design Differences Between RDBMS and Cassandra



- Query-first design
 - After we have all your tables laid out, we can start writing queries that pull together disparate data using the relationships defined by the keys.
 - The queries in the relational world are very much secondary.
 - It is assumed that we can always get the data we want if we have our tables modeled properly.
 - Even if we must use several complex subqueries or join statements, this is usually true.

Design Differences Between RDBMS and Cassandra



- Query-first design
 - By contrast, in Cassandra we don't start with the data model;
 - We start with the query model.
 - Instead of modeling the data first and then writing queries, with Cassandra we model the queries and let the data be organized around them.
 - Think of the most common query paths your application will use, and then create the tables that we need to support them.

Design Differences Between RDBMS and Cassandra



- Designing for optimal storage
 - In a relational database, it is frequently transparent to the user how tables are stored on disk, and it is rare to hear of recommendations about data modeling based on how the RDBMS might store tables on disk.
 - It is an important consideration in Cassandra.
 - Because Cassandra tables are each stored in separate files on disk, it's important to keep related columns defined together in the same table.
 - A key goal that we will see as we begin creating data models in Cassandra is to minimize the number of partitions that must be searched in order to satisfy a given query.
 - Because the partition is a unit of storage that does not get divided across nodes, a query that searches a single partition will typically yield the best performance

Design Differences Between RDBMS and Cassandra



- Sorting is a design decision
 - In an RDBMS, you can easily change the order in which records are returned to you by using ORDER BY in your query.
 - The default sort order is not configurable; by default, records are returned in the order in which they are written.
 - If you want to change the order, we just modify your query, and we can sort by any list of columns.
 - In Cassandra, however, sorting is treated differently; it is a design decision.
 - The sort order available on queries is fixed.
 - It is determined entirely by the selection of clustering columns we supply in the CREATE TABLE command.
 - The CQL SELECT statement does support ORDER BY semantics, but only in the order specified by the clustering columns.



Defining Application Queries

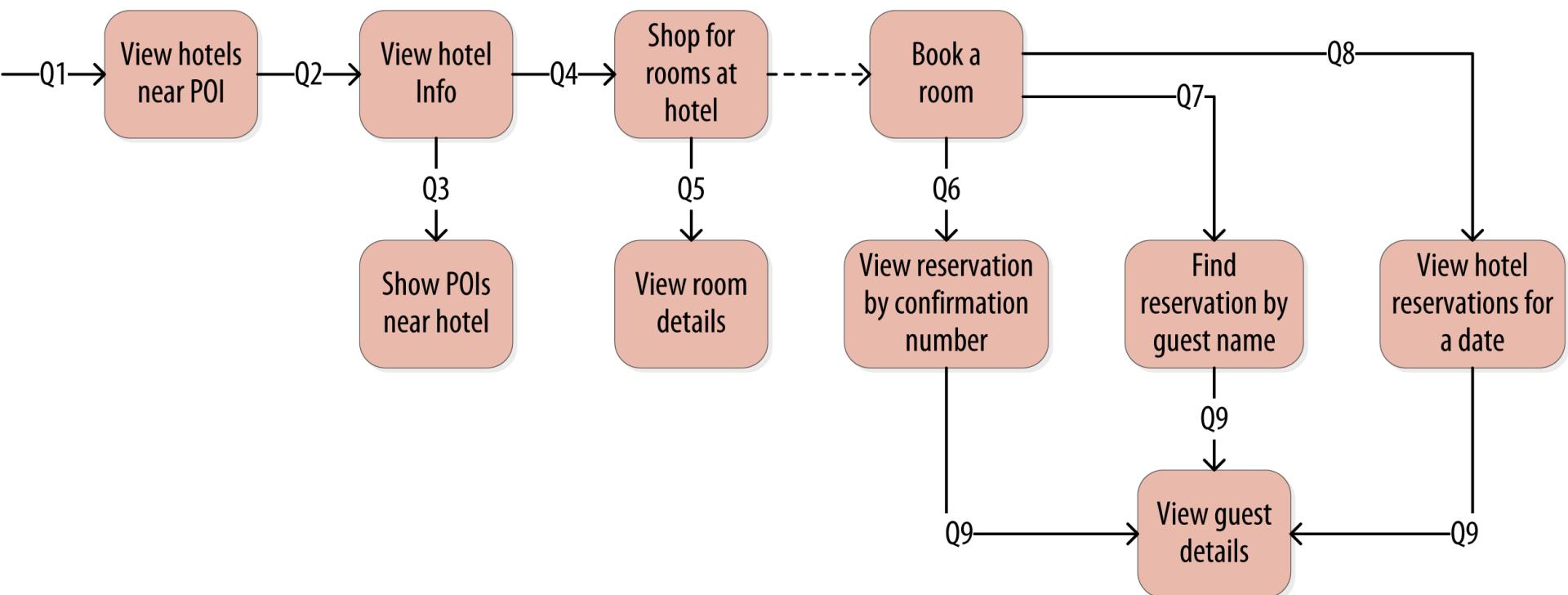
- Q1. Find hotels near a given point of interest.
- Q2. Find information about a given hotel, such as its name and location.
- Q3. Find points of interest near a given hotel.
- Q4. Find an available room in a given date range.
- Q5. Find the rate and amenities for a room.



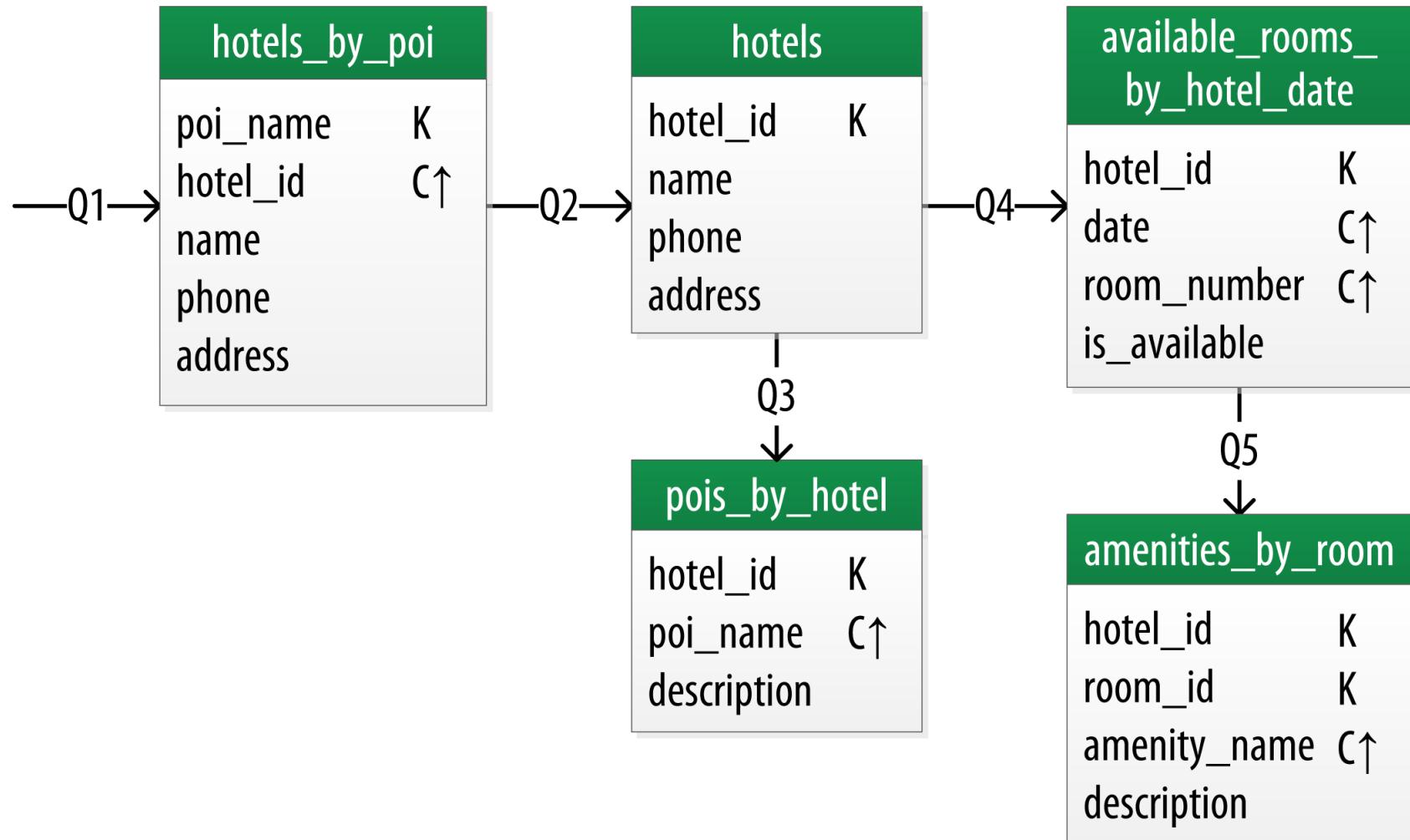
Defining Application Queries

- Q6. Lookup a reservation by confirmation number.
- Q7. Lookup a reservation by hotel, date, and guest name.
- Q8. Lookup all reservations by guest name.
- Q9. View guest details

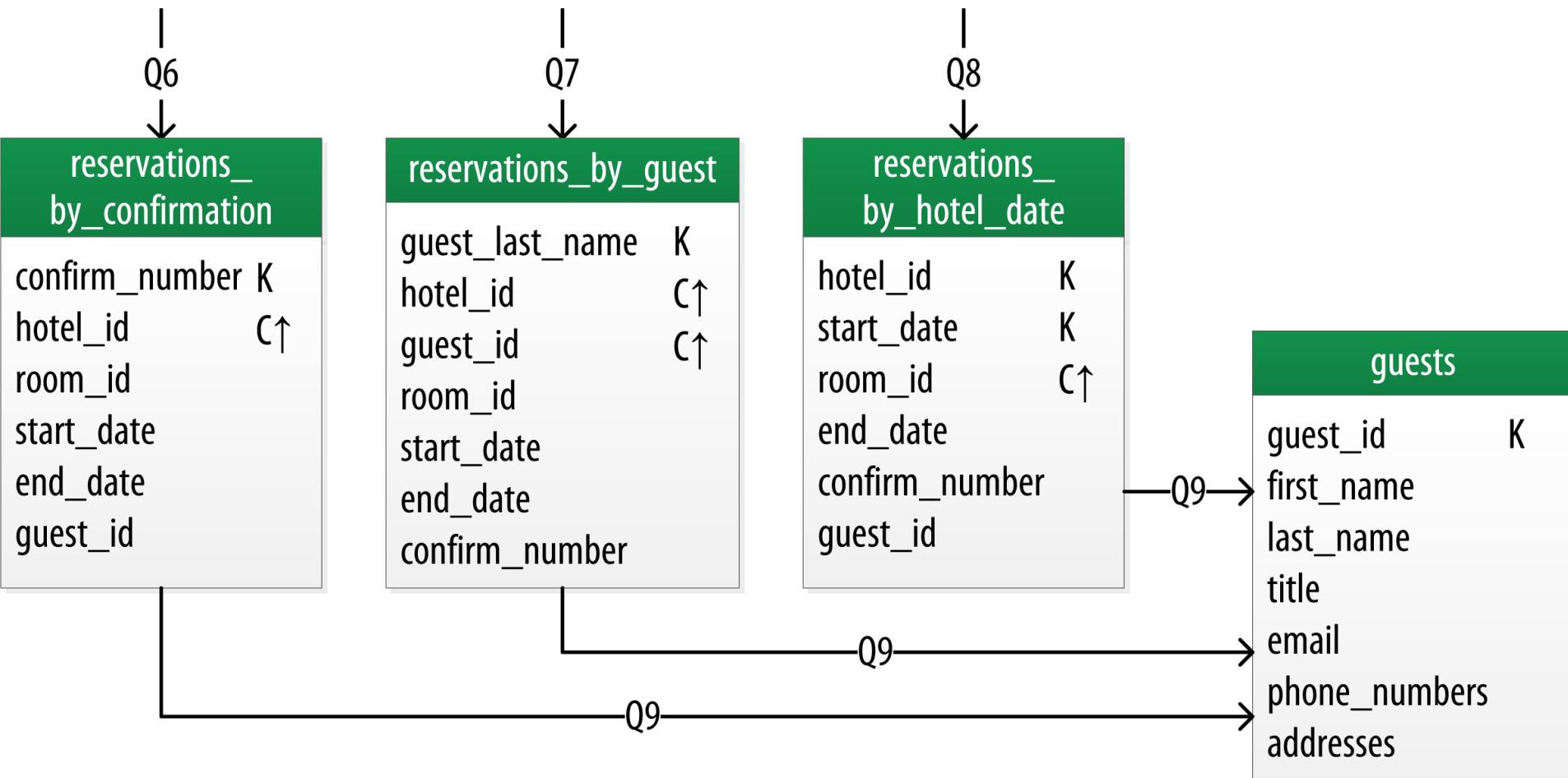
Logical Data Model



Logical Data Model



Logical Data Model



Physical Data Model

hotel keyspace

hotels		
hotel_id	text	K
name	text	
phone	text	
address	address	

hotels_by_poi		
poi_name	text	K
hotel_id	text	C↑
name	text	
phone	text	
address	address	

address		
street	text	
city	text	
state_or_province	text	
postal_code	text	
country	text	

available_rooms_by_hotel_date		
hotel_id	text	K
date	date	C↑
room_number	smallint	C↑
is_available	boolean	

pois_by_hotel		
hotel_id	text	K
poi_name	text	C↑
description	text	

amenities_by_room		
hotel_id	text	K
room_number	smallint	K
amenity_name	text	C↑
description	text	

Physical Data Model

reservation keyspace

reservations_by_hotel_date		
hotel_id	text	K
start_date	date	K
room_number	smallint	C↑
nights	smallint	
confirm_number	text	
guest_id	uuid	

reservations_by_confirmation		
confirm_number	text	K
hotel_id	text	C↑
start_date	date	C↑
room_number	smallint	C↑
nights	smallint	
guest_id	uuid	

reservations_by_guest		
guest_last_name	text	K
hotel_id	text	C↑
room_number	smallint	C↑
start_date	date	C↑
nights	smallint	
confirm_number	text	
guest_id	uuid	

guests		
guest_id	uuid	K
first_name	text	
last_name	text	
title	text	
{emails}	text	
[phone_numbers]	text	
<addresses>	text, address	

address	
street	text
city	text
state_or_province	text
postal_code	text
country	text



Physical Data Model

- Refer HotelSchema.txt

Keyspace

- CREATE KEYSPACE <identifier> WITH <properties>
- or
- Create keyspace KeyspaceName with
replicaton={'class':strategy name, 'replication_factor':
No of replications on different nodes}

Different Components of Keyspaces

- Strategy: There are two types of strategy declaration in Cassandra syntax:
- Simple Strategy: Simple strategy is used in the case of one data center. In this strategy, the first replica is placed on the selected node and the remaining nodes are placed in clockwise direction in the ring without considering rack or node location.
- Network Topology Strategy: This strategy is used in the case of more than one data centers. In this strategy, we must provide replication factor for each data center separately.



Different Components of Keyspaces

- Replication Factor: Replication factor is the number of replicas of data placed on different nodes.
- More than two replication factor are good to attain no single point of failure. So, 3 is good replication factor.
- CREATE KEYSPACE vinspace WITH replication = {'class':'SimpleStrategy', 'replication_factor' : 3};



Verification

- To check whether the keyspace is created or not, use the "DESCRIBE" command.
- By using this command you can see all the keyspaces that are created.
- Describe keyspaces

Nodetool Status – Data Center Status

```
rps@rps-virtual-machine:~/Desktop$ nodetool status
Datacenter: datacenter1
=====
Status=Up/Down
| / State=Normal/Leaving/Joining/Moving
--  Address      Load      Tokens  Owns (effective)  Host ID                               Rack
UN  172.20.0.161  233.85 KiB  16        100.0%          8b5b4796-9366-4b33-a4e4-d27f015d0e88  rack1
UN  172.20.0.160  260.55 KiB  16        100.0%          bcb6ae2a-fc0c-4f56-951e-5f0fd040bc2d  rack1
UN  172.20.0.162  180.94 KiB  16        100.0%          bf27a7a8-bc06-443c-ac95-770e772ae9b3  rack1
rps@rps-virtual-machine:~/Desktop$
```

Create a keyspace NetworkTopologyStrategy on an evaluation cluster



This example shows how to create a keyspace with network topology in a single node evaluation cluster.

`CREATE KEYSPACE cycling`

`WITH REPLICATION = {`

`'class' : 'NetworkTopologyStrategy',`

`'datacenter1' : 1`

`} ;`

Note: `datacenter1` is the default data center name. To display the data center name, use `nodetool status`.

Create a keyspace NetworkTopologyStrategy on an evaluation cluster



Create the cycling keyspace in an environment with multiple data centers

Set the replication factor for the Boston, Seattle, and Tokyo data centers. The data center name must match the name configured in the snitch.

```
CREATE KEYSPACE "Cycling"
```

```
WITH REPLICATION = {  
    'class' : 'NetworkTopologyStrategy',  
    'boston' : 3 , // Datacenter 1  
    'seattle' : 2 , // Datacenter 2  
    'tokyo' : 2 // Datacenter 3  
};
```

Note: For more about replication strategy options, see [Changing keyspace replication strategy](#)

Disabling durable writes

- Durable writes
 - Another keyspace option, which is often not required to tamper with, is `durable_writes`. By default, durable writes is set to true.
 - When a write request is received, the node first writes a copy of the data to an on-disk append-only structure called `commitlog`.
 - Then, it writes the data to an in-memory structure called `memtable`.
 - When the `memtable` is full or reaches a certain size, it is flushed to an on-disk immutable structure called `SSTable`.
 - Setting durable writes to true ensures data is written to the `commitlog`.
 - In case the node restarts, the `memtables` are gone since they reside in memory.
 - However, `memtables` can be reconstructed by replaying the `commitlog`, as the disk structure won't get wiped out even with node restarts.



Disabling durable writes

- Disable write commit log for the cycling keyspace. Disabling the commit log increases the risk of data loss. Do not disable in SimpleStrategy environments.

CREATE KEYSPACE cycling

WITH REPLICATION = {

'class' : 'NetworkTopologyStrategy',

'datacenter1' : 3

}

AND DURABLE_WRITES = false ;



ALTER KEYSPACE

- Modifies the keyspace replication strategy, the number of copies of the data Cassandra creates in each data center, **REPLICATION**, and/or disable the commit log for writes, **DURABLE_WRITES**.
- Restriction: Changing the keyspace name is not supported.



ALTER KEYSPACE

```
ALTER KEYSPACE keyspace_name  
  WITH REPLICATION = {  
    'class' : 'SimpleStrategy', 'replication_factor' : N  
   | 'class' : 'NetworkTopologyStrategy', 'dc1_name' : N  
   [, ...]  
  }  
  [AND DURABLE_WRITES = true|false] ;
```



ALTER KEYSPACE

ALTER KEYSPACE cycling

WITH REPLICATION = {

 'class' : 'NetworkTopologyStrategy',

 'datacenter1' : 3 }

AND DURABLE_WRITES = false ;



Cassandra Drop Keyspace

- DROP KEYSPACE [IF EXISTS] keyspace_name
- DROP KEYSPACE cycling;

Create Table

- CREATE TABLE [IF NOT EXISTS]
[keyspace_name.]table_name (
- column_definition [, ...]
- PRIMARY KEY (column_name [, column_name ...])
- [WITH table_options
 - | CLUSTERING ORDER BY
(clustering_column_name order)
 - | ID = 'table_hash_tag'
 - | COMPACT STORAGE]

Create Table

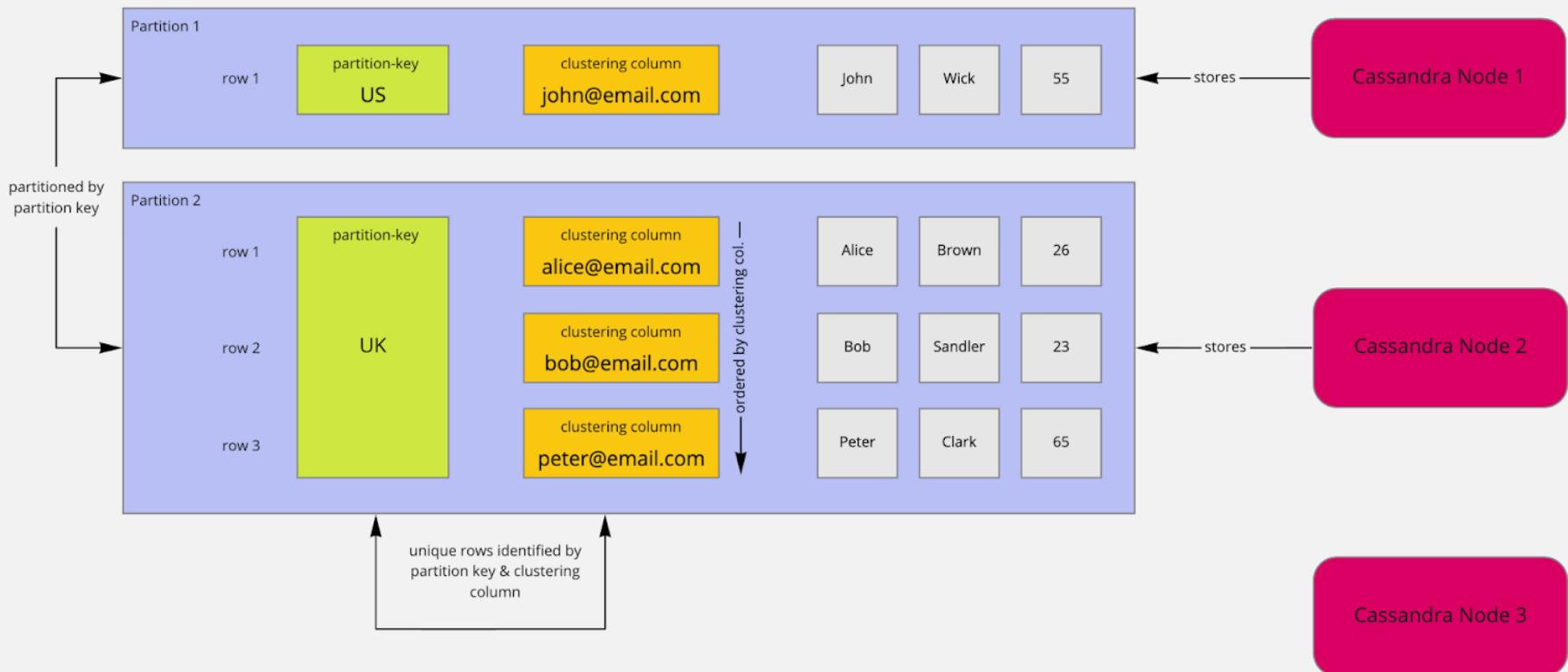
- CREATE TABLE cycling.race_winners (
- race_name text,
- race_position int,
- cyclist_name FROZEN<fullname>,
- PRIMARY KEY (race_name, race_position));

Create Table

```
CREATE TABLE learn_cassandra.users_by_country (  
    country text,  
    user_email text,  
    first_name text,  
    last_name text,  
    age smallint,  
    PRIMARY KEY ((country), user_email)  
)
```



Create Table





Create Table

- CREATE TABLE cycling.cyclist_name (
- id UUID PRIMARY KEY,
- lastname text,
- firstname text);



Composite Key

```
CREATE TABLE cycling.cyclist_category (
    category text,
    points int,
    id UUID,
    lastname text,
    PRIMARY KEY (category, points))
WITH CLUSTERING ORDER BY (points DESC);
```



Table Options

- Table properties tune data handling, including I/O operations, compression, and compaction. Set table properties in the following CQL requests:
- CREATE TABLE
- ALTER TABLE
- CREATE MATERIALIZED VIEW
- ALTER MATERIALIZED VIEW

Bulk Loader

- The Cassandra bulk loader, also called the `sstableloader`, provides the ability to:
 - Bulk load external data into a cluster.
 - Load existing SSTables into another cluster with a different number of nodes or replication strategy.
 - Restore snapshots



Multi Node Cluster Setup – Hardware Choices

- Like most databases, Cassandra throughput improves with more CPU cores, more RAM, and faster disks.
- While Cassandra can be made to run on small servers for testing or development environments (including Raspberry Pis), a minimal production server requires at least 2 cores, and at least 8GB of RAM.
- Typical production servers have 8 or more cores and at least 32GB of RAM.



Multi Node Cluster Setup – Hardware Choices

- CPU
 - Cassandra is highly concurrent, handling many simultaneous requests (both read and write) using multiple threads running on as many CPU cores as possible.
 - The Cassandra write path tends to be heavily optimized (writing to the commitlog and then inserting the data into the memtable).
 - It tends to be CPU bound. Consequently, adding additional CPU cores often increases throughput of both reads and writes.



Multi Node Cluster Setup – Hardware Choices

- Memory
 - Cassandra runs within a Java VM, which will pre-allocate a fixed size heap (java's Xmx system parameter).
 - In addition to the heap, Cassandra will use significant amounts of RAM offheap for compression metadata, bloom filters, row, key, and counter caches, and an in process page cache.
 - Finally, Cassandra will take advantage of the operating system's page cache, storing recently accessed portions files in RAM for rapid re-use.



Multi Node Cluster Setup – Hardware Choices

- Memory
 - For optimal performance, operators should benchmark and tune their clusters based on their individual workload.
- Basic guidelines suggest:
 - ECC RAM should always be used, as Cassandra has few internal safeguards to protect against bit level corruption
 - The Cassandra heap should be no less than 2GB, and no more than 50% of your system RAM
 - Heaps smaller than 12GB should consider ParNew/ConcurrentMarkSweep garbage collection
 - Heaps larger than 12GB should consider either:
 - 16GB heap with 8-10GB of new gen, a survivor ratio of 4-6, and a maximum tenuring threshold of 6
 - G1GC



Multi Node Cluster Setup – Hardware Choices

- Disks
 - Cassandra persists data to disk for two very different purposes.
 - The first is to the commitlog when a new write is made so that it can be replayed after a crash or system shutdown.
 - The second is to the data directory when thresholds are exceeded and memtables are flushed to disk as SSTables.
 - Cassandra performs very well on both spinning hard drives and solid-state disks.
 - In both cases, Cassandra's sorted immutable SSTables allow for linear reads, few seeks, and few overwrites, maximizing throughput for HDDs and lifespan of SSDs by avoiding write amplification.



Multi Node Cluster Setup – Hardware Choices

- Disks
 - Cassandra is designed to provide redundancy via multiple independent, inexpensive servers.
 - For this reason, using NFS or a SAN for data directories is an antipattern and should typically be avoided.
 - Servers with multiple disks are often better served by using RAID0 or JBOD than RAID1 or RAID5.



Multi Node Cluster Setup – Hardware Choices

- Cloud Choices
 - Many large users of Cassandra run in various clouds, including AWS, Azure, and GCE.
 - Cassandra will happily run in any of these environments.
 - Users should choose similar hardware to what would be needed in physical space.
 - In EC2, popular options include:
 - i2 instances, which provide both a high RAM:CPU ratio and local ephemeral SSDs
 - i3 instances with NVMe disks
 - EBS works okay if you want easy backups and replacements
 - m4.2xlarge / c4.4xlarge instances, which provide modern CPUs, enhanced networking and work well with EBS GP2 (SSD) storage.
 - Generally, disk and network performance increases with instance size and generation, so newer generations of instances and larger instance types within each family often perform better than their smaller or older alternatives.



Multi Node Cluster Setup – Hardware Choices

- Disks
 - Cassandra persists data to disk for two very different purposes.
 - The first is to the commitlog when a new write is made so that it can be replayed after a crash or system shutdown.
 - The second is to the data directory when thresholds are exceeded and memtables are flushed to disk as SSTables.
 - Cassandra performs very well on both spinning hard drives and solid-state disks.
 - In both cases, Cassandra's sorted immutable SSTables allow for linear reads, few seeks, and few overwrites, maximizing throughput for HDDs and lifespan of SSDs by avoiding write amplification.



MultiNode Data Center Procedure

- Suppose you install Cassandra on these nodes:
- node0 10.168.66.41 (seed1)
- node1 10.176.43.66
- node2 10.168.247.41
- node3 10.176.170.59 (seed2)
- node4 10.169.61.170
- node5 10.169.30.138



MultiNode Data Center Procedure

- If you have a firewall running in your cluster, you must open certain ports for communication between the nodes. See Configuring firewall port access.
- If Cassandra is running, you must stop the server and clear the data:
 - Doing this removes the default cluster_name (Test Cluster) from the system table. All nodes must use the same cluster name.
 - Package installations:
 - Stop Cassandra:
 - `sudo service cassandra stop` #Stops Cassandra
 - Clear the data:
 - `sudo rm -rf /var/lib/cassandra/*`

Multi Node Properties to Set

- cluster_name:
- num_tokens: recommended value: 256
- seeds: internal IP address of each seed node
- In new clusters. Seed nodes don't perform bootstrap (the process of a new node joining an existing cluster.)
- listen_address:
 - If the node is a seed node, this address must match an IP address in the seeds list. Otherwise, gossip communication fails because it doesn't know that it is a seed.
 - If not set, Cassandra asks the system for the local address, the one associated with its hostname. In some cases Cassandra doesn't produce the correct address and you must specify the listen_address.
- rpc_address: listen address for client connections
- endpoint_snitch: name of snitch (See endpoint_snitch.) If you are changing snitches, see Switching snitches.



Multi Node Properties to Set

- cluster_name: 'MyCassandraCluster'
- num_tokens: 256
- seed_provider:
- - class_name:
org.apache.cassandra.locator.SimpleSeedProvider
- parameters:
- - seeds: "10.168.66.41,10.176.170.59"
- listen_address:
- endpoint_snitch: GossipingPropertyFileSnitch



Multi Node Properties to Set

- In the `cassandra-rackdc.properties` file, assign the datacenter and rack names you determined in the Prerequisites. For example:
- Nodes 0 to 2
- ## Indicate the rack and dc for this node
- `dc=DC1`
- `rack=RAC1`
- Nodes 3 to 5
- ## Indicate the rack and dc for this node
- `dc=DC2`
- `rack=RAC1`



Multi Node Properties to Set

- Add the following in Cassandra_env.sh at the end
- `JVM_OPTS=\"$JVM_OPTS -Dcassandra.ignore_rack=true -Dcassandra.ignore_dc=true\"`



Multi Node Properties to Set

- The `GossipingPropertyFileSnitch` always loads `cassandra-topology.properties` when that file is present.
- Remove the file from each node on any new cluster or any cluster migrated from the `PropertyFileSnitch`.
- After you have installed and configured Cassandra on all nodes, DataStax recommends starting the seed nodes one at a time, and then starting the rest of the nodes.



Multi Node Properties to Set

- To check that the ring is up and running, run:
- Package installations:
- nodetool status
- nodetool ring

Multi Node Ring

Datacenter: datacenter1						
Address	Rack	Status	State	Load	Owns	Token
172.20.0.160	rack1	Up	Normal	369.71 KiB	?	8915758691576523162
172.20.0.161	rack1	Up	Normal	323.88 KiB	?	-8961100800967005060
172.20.0.161	rack1	Up	Normal	323.88 KiB	?	-8853052189992880533
172.20.0.162	rack1	Up	Normal	286.83 KiB	?	-8008675542501718775
172.20.0.160	rack1	Up	Normal	369.71 KiB	?	-7911962086435985156
172.20.0.160	rack1	Up	Normal	369.71 KiB	?	-7520832571064421820
172.20.0.162	rack1	Up	Normal	286.83 KiB	?	-6933816633919871080
172.20.0.161	rack1	Up	Normal	323.88 KiB	?	-6568407312599135535
172.20.0.160	rack1	Up	Normal	369.71 KiB	?	-6449331954724389008
172.20.0.161	rack1	Up	Normal	323.88 KiB	?	-5496906696259102723
172.20.0.160	rack1	Up	Normal	369.71 KiB	?	-5431480549981961992
172.20.0.162	rack1	Up	Normal	286.83 KiB	?	-5155182333096283951
172.20.0.160	rack1	Up	Normal	369.71 KiB	?	-4613808617369611198
172.20.0.161	rack1	Up	Normal	323.88 KiB	?	-4479055291516675707
172.20.0.162	rack1	Up	Normal	286.83 KiB	?	-4059140432861345662
172.20.0.161	rack1	Up	Normal	323.88 KiB	?	-3661383358904324913
172.20.0.160	rack1	Up	Normal	369.71 KiB	?	-3538052736144031658
172.20.0.162	rack1	Up	Normal	286.83 KiB	?	-2967843322169133402
172.20.0.160	rack1	Up	Normal	369.71 KiB	?	-2802137043433138018
172.20.0.161	rack1	Up	Normal	323.88 KiB	?	-2585627477678745373
172.20.0.162	rack1	Up	Normal	286.83 KiB	?	-2123466674677971646
172.20.0.161	rack1	Up	Normal	323.88 KiB	?	-1849711784967851733
172.20.0.160	rack1	Up	Normal	369.71 KiB	?	-1722764235985536361
172.20.0.161	rack1	Up	Normal	323.88 KiB	?	-770338977520250076
172.20.0.162	rack1	Up	Normal	286.83 KiB	?	-683198444775388407
172.20.0.160	rack1	Up	Normal	369.71 KiB	?	-479244230304944388
172.20.0.162	rack1	Up	Normal	286.83 KiB	?	388302171564644405
172.20.0.161	rack1	Up	Normal	323.88 KiB	?	473181028160341896
172.20.0.160	rack1	Up	Normal	369.71 KiB	?	188128812772581110



Preventing problems in gossip communications

This single datacenter example has 5 nodes, where nodeA, nodeB, and nodeC are seed nodes.

Node	IP address	Seed
nodeA	110.82.155.0	✓
nodeB	110.82.155.1	✓
nodeC	110.54.125.1	✓
nodeD	110.54.125.2	
nodeE	110.54.155.2	



Preventing problems in gossip communications

- For nodeA, nodeB, and nodeC, configure only nodeA as seed node:
- In `cassandra.yaml`:
- `seed_provider:`
- - `class_name:`
`org.apache.cassandra.locator.SimpleSeedProvider`
- - `seeds: 110.82.155.0`



Preventing problems in gossip communications

- For nodeA, nodeB, and nodeC, change `cassandra.yaml` to configure nodeA, nodeB, and nodeC as seed nodes:
- In `cassandra.yaml`:
- `seed_provider`:
- - `class_name`:
`org.apache.cassandra.locator.SimpleSeedProvider`
- - `seeds`: `110.82.155.0, 110.82.155.1, 110.54.125.1`
- You do not need to restart nodeA, nodeB, or nodeC after changing the seed node entry in `cassandra.yaml`; the nodes will reread the seed nodes.



Preventing problems in gossip communications

- For nodeD and nodeE, change `cassandra.yaml` to configure nodeA, nodeB, and nodeC as seed nodes:
- In `cassandra.yaml`:
- `seed_provider`:
- - `class_name`:
`org.apache.cassandra.locator.SimpleSeedProvider`
- - `seeds`: `110.82.155.0, 110.82.155.1, 110.54.125.1`
- Start nodeD and nodeE.
- Result: All nodes in the datacenter have the same seed nodes: nodeA, nodeB, and nodeC.

Nodetool remove node

```
rps@rps-virtual-machine:/etc/cassandra$ nodetool status
```

```
Datacenter: datacenter1
```

```
=====
```

```
Status=Up/Down
```

```
|/ State=Normal/Leaving/Joining/Moving
```

--	Address	Load	Tokens	Owns	Host ID	Rack
UN	172.20.0.161	261.78 KiB	16	?	8b5b4796-9366-4b33-a4e4-d27f015d0e88	rack1
DN	172.20.0.160	?	16	?	bcb6ae2a-fc0c-4f56-951e-5f0fd040bc2d	rack1
UN	172.20.0.179	248.57 KiB	16	?	f13a948b-e396-4634-9d9e-3a842abcb2e4	rack1
UN	172.20.0.162	337.41 KiB	16	?	bf27a7a8-bc06-443c-ac95-770e772ae9b3	rack1

```
Note: Non-system keyspaces don't have the same replication settings, effective ownership information is meaningless
```

```
rps@rps-virtual-machine:/etc/cassandra$ nodetool removenode bcb6ae2a-fc0c-4f56-951e-5f0fd040bc2d
```

```
rps@rps-virtual-machine:/etc/cassandra$
```

```
rps@rps-virtual-machine:/etc/cassandra$ nodetool status
```

```
Datacenter: datacenter1
```

```
=====
```

```
Status=Up/Down
```

```
|/ State=Normal/Leaving/Joining/Moving
```

--	Address	Load	Tokens	Owns	Host ID	Rack
UN	172.20.0.161	312.61 KiB	16	?	8b5b4796-9366-4b33-a4e4-d27f015d0e88	rack1
UN	172.20.0.179	248.57 KiB	16	?	f13a948b-e396-4634-9d9e-3a842abcb2e4	rack1
UN	172.20.0.162	385.71 KiB	16	?	bf27a7a8-bc06-443c-ac95-770e772ae9b3	rack1

```
Note: Non-system keyspaces don't have the same replication settings, effective ownership information is meaningless
```

```
rps@rps-virtual-machine:/etc/cassandra$ █
```

nodetool cleanup

- Cleans up keyspaces and partition keys no longer belonging to a node.
- Use this command to remove unwanted data after adding a new node to the cluster.
- Cassandra does not automatically remove data from nodes that lose part of their partition range to a newly added node.
- Run nodetool cleanup on the source node and on neighboring nodes that shared the same subrange after the new node is up and running.
- Failure to run this command after adding a node causes Cassandra to include the old data to rebalance the load on that node.

nodetool cleanup

- Running the nodetool cleanup command causes a temporary increase in disk space usage proportional to the size of your largest SSTable.
- Disk I/O occurs when running this command.
- Optionally, this command takes a list of table names. If you do not specify a keyspace, this command cleans all keyspaces no longer belonging to a node.



Using cassandra-stress for stress testing cluster

- Normally, a stress test is used to determine the system's robustness in terms of extreme load.
- It helps to know if the system will perform sufficiently in case the current load behaves well above the expected maximum.



Using cassandra-stress for stress testing cluster

- These are some use cases for Cassandra's stress tool:
 - Check the performance of your schema
 - Improve your setting and data model
 - Realize the scalability of your database
 - Figure out your production capacity



Using cassandra-stress for stress testing cluster

```
rps@rps-virtual-machine:/etc/cassandra$ cassandra-stress help
Usage:      cassandra-stress <command> [options]
Help usage: cassandra-stress help <command>

---Commands---
read          : Multiple concurrent reads - the cluster must first be populated by a write test
write         : Multiple concurrent writes against the cluster
mixed         : Interleaving of any basic commands, with configurable ratio and distribution - the cluster must first be populated by
a write test
counter_write : Multiple concurrent updates of counters.
counter_read  : Multiple concurrent reads of counters. The cluster must first be populated by a counterwrite test.
user          : Interleaving of user provided queries, with configurable ratio and distribution
help          : Print help for a command or option
print         : Inspect the output of a distribution definition
legacy        : Legacy support mode
version       : Print the version of cassandra stress

---Options---
-pop          : Population distribution and intra-partition visit order
-insert       : Insert specific options relating to various methods for batching and splitting partition updates
-col          : Column details such as size and count distribution, data generator, names, comparator and if super columns should be
used
-rate         : Thread count, rate limit or automatic mode (default is auto)
-mode         : CQL mode options
-errors       : How to handle errors when encountered during stress
-schema       : Replication settings, compression, compaction, etc.
-node         : Nodes to connect to
-log          : Where to log progress to, and the interval at which to do it
-transport    : Custom transport factories
-port         : The port to connect to cassandra nodes on
-sendto       : Specify a stress server to send this command to
-graph        : Graph recorded metrics
-tokenrange   : Token range settings
rps@rps-virtual-machine:/etc/cassandra$
```



Using cassandra-stress for stress testing cluster

- Write a sample test
- If you want to manually prove the execution of the database with random values, set up the next:
- Write: Execute write operations. In this case, we inserted ten million rows with 200 threads.
- n: Number of operations to execute.
- Let's walk through what Cassandra's stress tool does when you execute the next command:
- **cassandra-stress write n=10000000 –node 172.20.0.161 –port native=9042**



Using cassandra-stress for stress testing cluster

```
rps@rps-virtual-machine:/etc/cassandra$ cassandra-stress write n=100000 -node 172.20.0.161 -port native=9042
***** Stress Settings *****
Command:
  Type: write
  Count: 100,000
  No Warmup: false
  Consistency Level: LOCAL_ONE
  Target Uncertainty: not applicable
  Key Size (bytes): 10
  Counter Increment Distribution: add=fixed(1)
Rate:
  Auto: false
  Thread Count: 200
  OpsPer Sec: 0
Population:
  Sequence: 1..100000
  Order: ARBITRARY
  Wrap: true
Insert:
  Revisits: Uniform: min=1,max=1000000
  Visits: Fixed: key=1
  Row Population Ratio: Ratio: divisor=1.000000;delegate=Fixed: key=1
  Batch Type: not batching
Columns:
  Max Columns Per Key: 5
  Column Names: [C0, C1, C2, C3, C4]
  Comparator: AsciiType
  Timestamp: null
  Variable Column Count: false
  Slice: false
  Size Distribution: Fixed: key=34
  Count Distribution: Fixed: key=5
Errors:
  Ignore: false
  Tries: 10
Log:
```



Using cassandra-stress for stress testing cluster

1. Cassandra's stress tool creates a keyspace by executing this DDL:

```
1 CREATE KEYSPACE IF NOT EXISTS keyspace1
2   WITH durable_writes = true
3   AND replication = {
4     'class' : 'SimpleStrategy',
5     'replication_factor' : 1
6   };
```



Using cassandra-stress for stress testing cluster

2. Once the keyspace has been created, Cassandra-stress creates two tables: standard1 and counter1. In this test, we ignored counter1 since we didn't use it. The definition of the standard1 table is the following:

```
1 CREATE TABLE IF NOT EXISTS standard1
2   (key blob PRIMARY KEY,
3    "C0" blob, "C1" blob, "C2" blob, "C3" blob, "C4" blob) WITH COMPACT STORAGE AND compression = {}
```



Using cassandra-stress for stress testing cluster

Results:

```
Op rate           : 33,374 op/s [WRITE: 33,374 op/s]
Partition rate   : 33,374 pk/s [WRITE: 33,374 pk/s]
Row rate          : 33,374 row/s [WRITE: 33,374 row/s]
Latency mean     : 5.3 ms [WRITE: 5.3 ms]
Latency median    : 3.4 ms [WRITE: 3.4 ms]
Latency 95th percentile : 15.1 ms [WRITE: 15.1 ms]
Latency 99th percentile : 39.7 ms [WRITE: 39.7 ms]
Latency 99.9th percentile : 80.9 ms [WRITE: 80.9 ms]
Latency max       : 141.4 ms [WRITE: 141.4 ms]
Total partitions   : 100,000 [WRITE: 100,000]
Total errors        : 0 [WRITE: 0]
Total GC count     : 0
Total GC memory    : 0.000 KiB
Total GC time       : 0.0 seconds
Avg GC time         : NaN ms
StdDev GC time      : 0.0 ms
Total operation time : 00:00:02
```



Using cassandra-stress for stress testing cluster

- Read sample test
- **cassandra-stress read n=10000000 –node 172.20.0.161 –port native=9042**



Using cassandra-stress for stress testing cluster

```
,    3.0,  0.19286,      0,      0,      0,      0,      0,      0
total,
,    4.0,  0.14360,      0,      0,      0,      0,      0
total,
,    4.3,  0.11478,      0,      0,      0,      0,      0,      0
                                                               92441,   26369,   26369,   26369,   0.6,   0.5,   1.0,   1.5,  15.1,  18.6
                                                               100000,   26345,   26345,   26345,   0.6,   0.5,   1.0,   1.7,  11.4,  11.9

Results:
Op rate           : 23,327 op/s [READ: 23,327 op/s]
Partition rate   : 23,327 pk/s [READ: 23,327 pk/s]
Row rate          : 23,327 row/s [READ: 23,327 row/s]
Latency mean     : 0.6 ms [READ: 0.6 ms]
Latency median    : 0.5 ms [READ: 0.5 ms]
Latency 95th percentile : 1.0 ms [READ: 1.0 ms]
Latency 99th percentile : 1.5 ms [READ: 1.5 ms]
Latency 99.9th percentile : 11.6 ms [READ: 11.6 ms]
Latency max       : 31.1 ms [READ: 31.1 ms]
Total partitions  : 100,000 [READ: 100,000]
Total errors      : 0 [READ: 0]
Total GC count    : 0
Total GC memory   : 0.000 KiB
Total GC time     : 0.0 seconds
Avg GC time       : NaN ms
StdDev GC time   : 0.0 ms
Total operation time: 00:00:04

Improvement over 8 threadCount: 53%

Running with 24 threadCount
Running READ with 24 threads for 100000 iteration
Failed to connect over JMX; not collecting these stats
type           total ops,  op/s,  pk/s,  row/s,  mean,  med,  .95,  .99,  .999,  max
,  time,  stderr,  errors,  gc: #,  max ms,  sum ms,  sdv ms,  mb
total,                               24155,   24155,   24155,   24155,   0.7,   0.6,   1.2,   1.9,   7.9,  15.2
,  1.0,  0.00000,      0,      0,      0,      0,      0
total,                               57586,   33431,   33431,   33431,   0.7,   0.6,   1.2,   1.8,  19.8,  44.2
,  2.0,  0.12416,      0,      0,      0,      0,      0
```



Using cassandra-stress for stress testing cluster

```
24 threadCount, READ,
 14.2, 44.2, 3.2, 0.07338, 0, 0, 0, 0, 100000, 31038, 31038, 31038, 0.7, 0.6, 1.2, 1.8,
24 threadCount, total,
 14.2, 44.2, 3.2, 0.07338, 0, 0, 0, 0, 100000, 31038, 31038, 31038, 0.7, 0.6, 1.2, 1.8,
36 threadCount, READ,
 14.4, 50.0, 2.7, 0.07842, 0, 0, 0, 0, 100000, 37700, 37700, 37700, 0.9, 0.7, 1.6, 3.2,
36 threadCount, total,
 14.4, 50.0, 2.7, 0.07842, 0, 0, 0, 0, 100000, 37700, 37700, 37700, 0.9, 0.7, 1.6, 3.2,
54 threadCount, READ,
 19.1, 34.3, 2.8, 0.25662, 0, 0, 0, 0, 100000, 35107, 35107, 35107, 1.1, 0.9, 2.4, 6.4,
54 threadCount, total,
 19.1, 34.3, 2.8, 0.25662, 0, 0, 0, 0, 100000, 35107, 35107, 35107, 1.1, 0.9, 2.4, 6.4,
81 threadCount, READ,
 16.8, 33.8, 2.6, 0.34329, 0, 0, 0, 0, 100000, 38955, 38955, 38955, 1.3, 1.1, 2.9, 5.8,
81 threadCount, total,
 16.8, 33.8, 2.6, 0.34329, 0, 0, 0, 0, 100000, 38955, 38955, 38955, 1.3, 1.1, 2.9, 5.8,
121 threadCount, READ,
 45.4, 67.6, 2.2, 0.23010, 0, 0, 0, 0, 100000, 46056, 46056, 46056, 1.9, 1.4, 4.1, 12.0,
121 threadCount, total,
 45.4, 67.6, 2.2, 0.23010, 0, 0, 0, 0, 100000, 46056, 46056, 46056, 1.9, 1.4, 4.1, 12.0,
181 threadCount, READ,
 54.2, 165.9, 1.7, 0.08886, 0, 0, 0, 0, 100000, 57562, 57562, 57562, 2.7, 1.8, 5.9, 22.2,
181 threadCount, total,
 54.2, 165.9, 1.7, 0.08886, 0, 0, 0, 0, 100000, 57562, 57562, 57562, 2.7, 1.8, 5.9, 22.2,
271 threadCount, READ,
 52.1, 114.2, 2.2, 0.34545, 0, 0, 0, 0, 100000, 44869, 44869, 44869, 3.7, 2.5, 9.8, 21.2,
271 threadCount, total,
 52.1, 114.2, 2.2, 0.34545, 0, 0, 0, 0, 100000, 44869, 44869, 44869, 3.7, 2.5, 9.8, 21.2,
406 threadCount, READ,
 72.9, 223.1, 1.7, 0.22431, 0, 0, 0, 0, 100000, 58950, 58950, 58950, 5.4, 3.3, 15.5, 42.8,
406 threadCount, total,
 72.9, 223.1, 1.7, 0.22431, 0, 0, 0, 0, 100000, 58950, 58950, 58950, 5.4, 3.3, 15.5, 42.8,
609 threadCount, READ,
 113.2, 263.1, 2.2,-10.20888, 0, 0, 0, 0, 100000, 46213, 46213, 46213, 7.6, 4.2, 25.7, 54.6,
609 threadCount, total,
 113.2, 263.1, 2.2,-10.20888, 0, 0, 0, 0, 100000, 46213, 46213, 46213, 7.6, 4.2, 25.7, 54.6,
```



Using cassandra-stress for stress testing cluster

- # Insert (write) one million rows
- \$ cassandra-stress write n=1000000 -rate threads=50
- # Read two hundred thousand rows.
- \$ cassandra-stress read n=200000 -rate threads=50
- # Read rows for a duration of 3 minutes.
- \$ cassandra-stress read duration=3m -rate threads=50
- # Read 200,000 rows without a warmup of 50,000 rows first.
- \$ cassandra-stress read n=200000 no-warmup -rate threads=50



Using cassandra-stress for stress testing cluster

- cassandra-stress user profile=tools/cqlstress-example.yaml n=1000000 ops\insert=3,read1=1\ no-warmup cl=QUORUM



Using cassandra-stress for stress testing cluster

```
total,          6.0,  0.10252,      0,      0,      0,      0,  47640,   9697,   9697,   9697,   0.4,   0.4,   0.5,   0.7,   1.6, 13.9
total,          7.0,  0.09040,      0,      0,      0,      0,  57516,   9876,   9876,   9876,   0.4,   0.4,   0.5,   0.7,   1.8, 17.0
total,          8.0,  0.08058,      0,      0,      0,      0,  67444,   9928,   9928,   9928,   0.4,   0.4,   0.5,   0.7,   1.7, 15.2
total,          9.0,  0.07159,      0,      0,      0,      0,  76739,   9295,   9295,   9295,   0.4,   0.4,   0.6,   0.9,   4.3, 15.0
total,         10.0,  0.06488,      0,      0,      0,      0,  86505,   9766,   9766,   9766,   0.4,   0.4,   0.5,   0.8,   3.9, 13.1
total,         11.0,  0.05906,      0,      0,      0,      0,  95945,   9440,   9440,   9440,   0.4,   0.4,   0.6,   0.8,   3.8, 22.7
total,         11.4,  0.05473,      0,      0,      0,      0, 100000,  10130,  10130,  10130,   0.4,   0.4,   0.5,   0.7,   1.2,  2.2

Results:
Op rate           : 8,772 op/s [insert: 8,772 op/s]
Partition rate   : 8,772 pk/s [insert: 8,772 pk/s]
Row rate          : 8,772 row/s [insert: 8,772 row/s]
Latency mean     : 0.4 ms [insert: 0.4 ms]
Latency median    : 0.4 ms [insert: 0.4 ms]
Latency 95th percentile : 0.6 ms [insert: 0.6 ms]
Latency 99th percentile : 0.9 ms [insert: 0.9 ms]
Latency 99.9th percentile : 4.3 ms [insert: 4.3 ms]
Latency max       : 33.7 ms [insert: 33.7 ms]
Total partitions  : 100,000 [insert: 100,000]
Total errors      : 0 [insert: 0]
Total GC count   : 0
Total GC memory   : 0.000 KiB
Total GC time     : 0.0 seconds
Avg GC time       : NaN ms
StdDev GC time   : 0.0 ms
Total operation time : 00:00:11

Sleeping for 15s
^C
```



Cassandra Monitoring Tool - jconsole

Java Monitoring & Management Console

Connection Window Help

total,
, 7.0, 0.09040, 0, 0,
total,
, 8.0, 0.08058, 0, 0,
total,
, 9.0, 0.07159, 0, 0,
total,
, 10.0, 0.06488, 0, 0,
total,
, 11.0, 0.05906, 0, 0,
total,
, 11.4, 0.05473, 0, 0,

Results:
Op rate : 8,772 op/s
Partition rate : 8,772 pk/s
Row rate : 8,772 row/s
Latency mean : 0.4 ms [inclusive]
Latency median : 0.4 ms [inclusive]
Latency 95th percentile : 0.6 ms [inclusive]
Latency 99th percentile : 0.9 ms [inclusive]
Latency 99.9th percentile : 4.3 ms [inclusive]
Latency max : 33.7 ms [inclusive]
Total partitions : 100,000 [inclusive]
Total errors : 0 [inclusive]
Total GC count : 0
Total GC memory : 0.000 KiB
Total GC time : 0.0 seconds
Avg GC time : NaN ms
StdDev GC time : 0.0 ms
Total operation time : 00:00:11

Sleeping for 15s

^C

rps@rps-virtual-machine:~\$ jconsole

Gtk-Message: 01:22:38.981: Failed to load

Saved to this PC

JConsole: New Connection

New Connection

Local Process:

Name: jdk.jconsole/sun.tools.jconsole.JConsole PID: 3072...

Note: The management agent will be enabled on this process.

Remote Process:

Usage: <hostname>:<port> OR service:jmx:<protocol>:<service>

Username: Password:

Cancel Connect

JConsole: New Connection



Cassandra Monitoring Tool - jconsole

```
total,          7.0,  0.09040,      0,      0,
total,          8.0,  0.08058,      0,      0,
total,          9.0,  0.07159,      0,      0,
total,         10.0,  0.06488,      0,      0,
total,         11.0,  0.05906,      0,      0,
total,         11.4,  0.05473,      0,      0,

Results:
Op rate           : 8,772 op/s
Partition rate   : 8,772 pk/s
Row rate          : 8,772 row/s
Latency mean     : 0.4 ms [in]
Latency median    : 0.4 ms [in]
Latency 95th percentile : 0.6 ms [in]
Latency 99th percentile : 0.9 ms [in]
Latency 99.9th percentile : 4.3 ms [in]
Latency max       : 33.7 ms [in]
Total partitions  : 100,000 [in]
Total errors      : 0 [in]
Total GC count    : 0
Total GC memory   : 0.000 KiB
Total GC time     : 0.0 seconds
Avg GC time       : NaN ms
StdDev GC time   : 0.0 ms
Total operation time: 00:00:11

Sleeping for 15s
^C
rps@rps-virtual-machine:~$ jconsole
Gtk-Message: 01:22:38.981: Failed to load

```

Connection Window Help

pid: 307285 jdk.jconsole/sun.tools.jconsole.JConsole

Overview Memory Threads Classes VM Summary MBeans

Time Range: All

Heap Memory Usage

Used: 16.6 Mb Committed: 26.2 Mb Max: 8.4 Gb

Threads

Live: 38 Peak: 39 Total: 49

Classes

Loaded: 5,269 Unloaded: 2 Total: 5,271

CPU Usage

CPU Usage: 2.3%



Node Repair

- Cassandra is designed to remain available if one of its nodes is down or unreachable.
- When a node is down or unreachable, it needs to eventually discover the writes it missed.
- Hints attempt to inform a node of missed writes, but are a best effort, and aren't guaranteed to inform a node of 100% of the writes it missed.
- These inconsistencies can eventually result in data loss as nodes are replaced or tombstones expire.

Node Repair

- These inconsistencies are fixed with the repair process.
- Repair synchronizes the data between nodes by comparing their respective datasets for their common token ranges, and streaming the differences for any out of sync sections between the nodes.
- It compares the data with merkle trees, which are a hierarchy of hashes.



Incremental and Full Repairs

- There are 2 types of repairs: full repairs, and incremental repairs.
- Full repairs operate over all of the data in the token range being repaired.
- Incremental repairs only repair data that's been written since the previous incremental repair.
- Incremental repairs are the default repair type, and if run regularly, can significantly reduce the time and io cost of performing a repair.



Incremental and Full Repairs

- It's important to understand that once an incremental repair marks data as repaired, it won't try to repair it again.
- This is fine for syncing up missed writes, but it doesn't protect against things like disk corruption, data loss by operator error, or bugs in Cassandra.
- For this reason, full repairs should still be run occasionally



Incremental and Full Repairs

- Incremental repair is the default and is run with the following command:
 - nodetool repair
- A full repair can be run with the following command:
 - nodetool repair --full
- Additionally, repair can be run on a single keyspace:
 - nodetool repair [options] <keyspace_name>
- Or even on specific tables:
 - nodetool repair [options] <keyspace_name> <table1> <table2>



Full Repair Example

- Full repair is typically needed to redistribute data after increasing the replication factor of a keyspace or after adding a node to the cluster.
- Full repair involves streaming SSTables.
- To demonstrate full repair start with a three node cluster.

```
[ec2-user@ip-10-0-2-238 ~]$ nodetool status
Datacenter: us-east-1
=====
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
-- Address      Load      Tokens  Owns    Host ID          Rack
UN 10.0.1.115  547 KiB     256     ?  b64cb32a-b32a-46b4-9eeb-e123fa8fc287  us-east-1b
UN 10.0.3.206  617.91 KiB   256     ?  74863177-684b-45f4-99f7-d1006625dc9e  us-east-1d
UN 10.0.2.238  670.26 KiB   256     ?  4dcdadd2-41f9-4f34-9892-1f20868b27c7  us-east-1c
```

Create a keyspace with replication factor 3:



Full Repair Example

Create a keyspace with replication factor 3:

```
cqlsh> DROP KEYSPACE cqlkeyspace;
cqlsh> CREATE KEYSPACE CQLKeyspace
... WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};
```

Add a table to the keyspace:

```
cqlsh> use cqlkeyspace;
cqlsh:cqlkeyspace> CREATE TABLE t (
...   id int,
...   k int,
...   v text,
...   PRIMARY KEY (id)
... );
```

Add table data:

```
cqlsh:cqlkeyspace> INSERT INTO t (id, k, v) VALUES (0, 0, 'val0');
cqlsh:cqlkeyspace> INSERT INTO t (id, k, v) VALUES (1, 1, 'val1');
cqlsh:cqlkeyspace> INSERT INTO t (id, k, v) VALUES (2, 2, 'val2');
```

A query lists the data added:



Full Repair Example

```
cqlsh:cqlkeyspace> SELECT * FROM t;
```

id	k	v
1	1	val1
0	0	val0
2	2	val2

(3 rows)

Make the following changes to a three node cluster:

Increase the replication factor from 3 to 4.

Add a 4th node to the cluster

When the replication factor is increased the following message gets output indicating that a full repair is needed as per ([CASSANDRA-13079](#)):

```
cqlsh:cqlkeyspace> ALTER KEYSPACE CQLKeyspace
    ... WITH replication = {'class': 'simplestrategy', 'replication_factor' : 4};
Warnings :
When increasing replication factor you need to run a full (-full) repair to distribute the
data.
```

Perform a full repair on the keyspace `cqlkeyspace` table `t` with following command:



Full Repair Example

```
cqlsh:cqlkeyspace> SELECT * FROM t;
```

id	k	v
1	1	val1
0	0	val0
2	2	val2

(3 rows)

Make the following changes to a three node cluster:

Increase the replication factor from 3 to 4.

Add a 4th node to the cluster

When the replication factor is increased the following message gets output indicating that a full repair is needed as per ([CASSANDRA-13079](#)):

```
cqlsh:cqlkeyspace> ALTER KEYSPACE CQLKeyspace
    ... WITH replication = {'class': 'simplestrategy', 'replication_factor' : 4};
Warnings :
When increasing replication factor you need to run a full (-full) repair to distribute the
data.
```

Perform a full repair on the keyspace `cqlkeyspace` table `t` with following command:



Full Repair Example

Perform a full repair on the keyspace `cqlkeyspace` table `t`:

```
nodetool repair -full cqlkeyspace t
```

Full repair completes in about a second as indicated by:

```
[ec2-user@ip-10-0-2-238 ~]$ nodetool repair -full cqlkeyspace t
[2019-08-17 03:06:21,445] Starting repair command #1 (fd576d
pace cqlkeyspace with repair options (parallelism: parallel,
threads: 1, ColumnFamilies: [t], dataCenters: [], hosts: [],
air: false, force repair: false, optimise streams: false)
[2019-08-17 03:06:23,059] Repair session fd8e5c20-c09b-11e9-
505,-8786320730900698730], (-5454146041421260303,-5439402053
52322], ... , (4350676211955643098,4351706629422088296]] fin
[2019-08-17 03:06:23,077] Repair completed successfully
[2019-08-17 03:06:23,077] Repair command #1 finished in 1 se
[ec2-user@ip-10-0-2-238 ~]$
```

The `nodetool tpstats` command should list a repair has been completed column value of 1:

```
[ec2-user@ip-10-0-2-238 ~]$ nodetool tpstats
Pool Name Active Pending Completed Blocked All time blo
Readstage 0 0 99 0 0
...
Repair-Task 0 0 1 0 0
```



How is the consistency level configured?

- Consistency levels in Cassandra can be configured to manage availability versus data accuracy.
- Configure consistency for a session or per individual read or write operation.
- Within cqlsh, use **CONSISTENCY**, to set the consistency level for all queries in the current cqlsh session.
- For programming client applications, set the consistency level using an appropriate driver.
- For example, using the Java driver, call `QueryBuilder.insertInto` with `setConsistencyLevel` to set a per-insert consistency level.
- The consistency level defaults to **ONE** for all write and read operations.



How is the Write consistency level configured?

Write Consistency Levels

Level	Description	Usage
ALL	A write must be written to the commit log and memtable on all replica nodes in the cluster for that partition.	Provides the highest consistency and the lowest availability of any other level.
EACH_QUORUM	Strong consistency. A write must be written to the commit log and memtable on a quorum of replica nodes in each datacenter .	Used in multiple datacenter clusters to strictly maintain consistency at the same level in each datacenter. For example, choose this level if you want a write to fail when a datacenter is down and the QUORUM cannot be reached on that datacenter.
QUORUM	A write must be written to the commit log and memtable on a quorum of replica nodes across <i>all</i> datacenters.	Used in either single or multiple datacenter clusters to maintain strong consistency across the cluster. Use if you can tolerate



How is the write consistency level configured?

LOCAL_QUORUM	Strong consistency. A write must be written to the commit log and memtable on a quorum of replica nodes in the same datacenter as the coordinator . Avoids latency of inter-datacenter communication.	Used in multiple datacenter clusters with a rack-aware replica placement strategy, such as NetworkTopologyStrategy , and a properly configured snitch. Use to maintain consistency locally (within the single datacenter). Can be used with SimpleStrategy .
ONE	A write must be written to the commit log and memtable of at least one replica node.	Satisfies the needs of most users because consistency requirements are not stringent.
TWO	A write must be written to the commit log and memtable of at least two replica nodes.	Similar to ONE.
THREE	A write must be written to the commit log and memtable of at least three replica nodes.	Similar to TWO.



How is the write consistency level configured?

LOCAL_ONE	A write must be sent to, and successfully acknowledged by, at least one replica node in the local datacenter.	In a multiple datacenter clusters, a consistency level of ONE is often desirable, but cross-DC traffic is not. LOCAL_ONE accomplishes this. For security and quality reasons, you can use this consistency level in an offline datacenter to prevent automatic connection to online nodes in other datacenters if an offline node goes down.
ANY	A write must be written to at least one node. If all replica nodes for the given partition key are down, the write can still succeed after a hinted handoff has been written. If all replica nodes are down at write time, an ANY write is not readable until the replica nodes for that partition have recovered.	Provides low latency and a guarantee that a write never fails. Delivers the lowest consistency and highest availability.



How is the Read consistency level configured?

Level	Description	Usage
ALL	Returns the record after all replicas have responded. The read operation will fail if a replica does not respond.	Provides the highest consistency of all levels and the lowest availability of all levels.
EACH_QUORUM	Not supported for reads.	
QUORUM	Returns the record after a quorum of replicas from all datacenters has responded.	Used in either single or multiple datacenter clusters to maintain strong consistency across the cluster. Ensures strong consistency if you can tolerate some level of failure.
LOCAL_QUORUM	Returns the record after a quorum of replicas in the current datacenter as the <u>coordinator</u> has reported. Avoids latency of inter-datacenter communication.	Used in multiple datacenter clusters with a rack-aware replica placement strategy (<code>NetworkTopologyStrategy</code>) and a properly configured snitch. Fails when using <code>SimpleStrategy</code> .



How is the Read consistency level configured?

ONE	Returns a response from the closest replica, as determined by the snitch . By default, a read repair runs in the background to make the other replicas consistent.	Provides the highest availability of all the levels if you can tolerate a comparatively high probability of stale data being read. The replicas contacted for reads may not always have the most recent write.
TWO	Returns the most recent data from two of the closest replicas.	Similar to ONE.
THREE	Returns the most recent data from three of the closest replicas.	Similar to TWO.
LOCAL_ONE	Returns a response from the closest replica in the local datacenter.	Same usage as described in the table about write consistency levels.



How is the Read consistency level configured?

LOCAL_ONE	Returns a response from the closest replica in the local datacenter.	Same usage as described in the table about write consistency levels.
SERIAL	Allows reading the current (and possibly uncommitted) state of data without proposing a new addition or update. If a SERIAL read finds an uncommitted transaction in progress, it will commit the transaction as part of the read. Similar to QUORUM.	To read the latest value of a column after a user has invoked a lightweight transaction to write to the column, use SERIAL. Cassandra then checks the inflight lightweight transaction for updates and, if found, returns the latest data.
LOCAL_SERIAL	Same as SERIAL, but confined to the datacenter. Similar to LOCAL_QUORUM.	Used to achieve linearizable consistency for lightweight transactions.



How QUORUM is calculated

- The QUORUM level writes to the number of nodes that make up a quorum.
- A quorum is calculated, and then rounded down to a whole number, as follows:
- $\text{quorum} = (\text{sum_of_replication_factors} / 2) + 1$
- The sum of all the replication_factor settings for each datacenter is the sum_of_replication_factors.
- $\text{sum_of_replication_factors} = \text{datacenter1_RF} + \text{datacenter2_RF} + \dots + \text{datacenter}_n\text{_RF}$



Setting Consistency Level

- CONSISTENCY QUORUM
- CONSISTENCY SERIAL
- SELECT * FROM cycling.race_winners ;
- Inserts with CONSISTENCY SERIAL fail:
- Updates with CONSISTENCY SERIAL also fail:

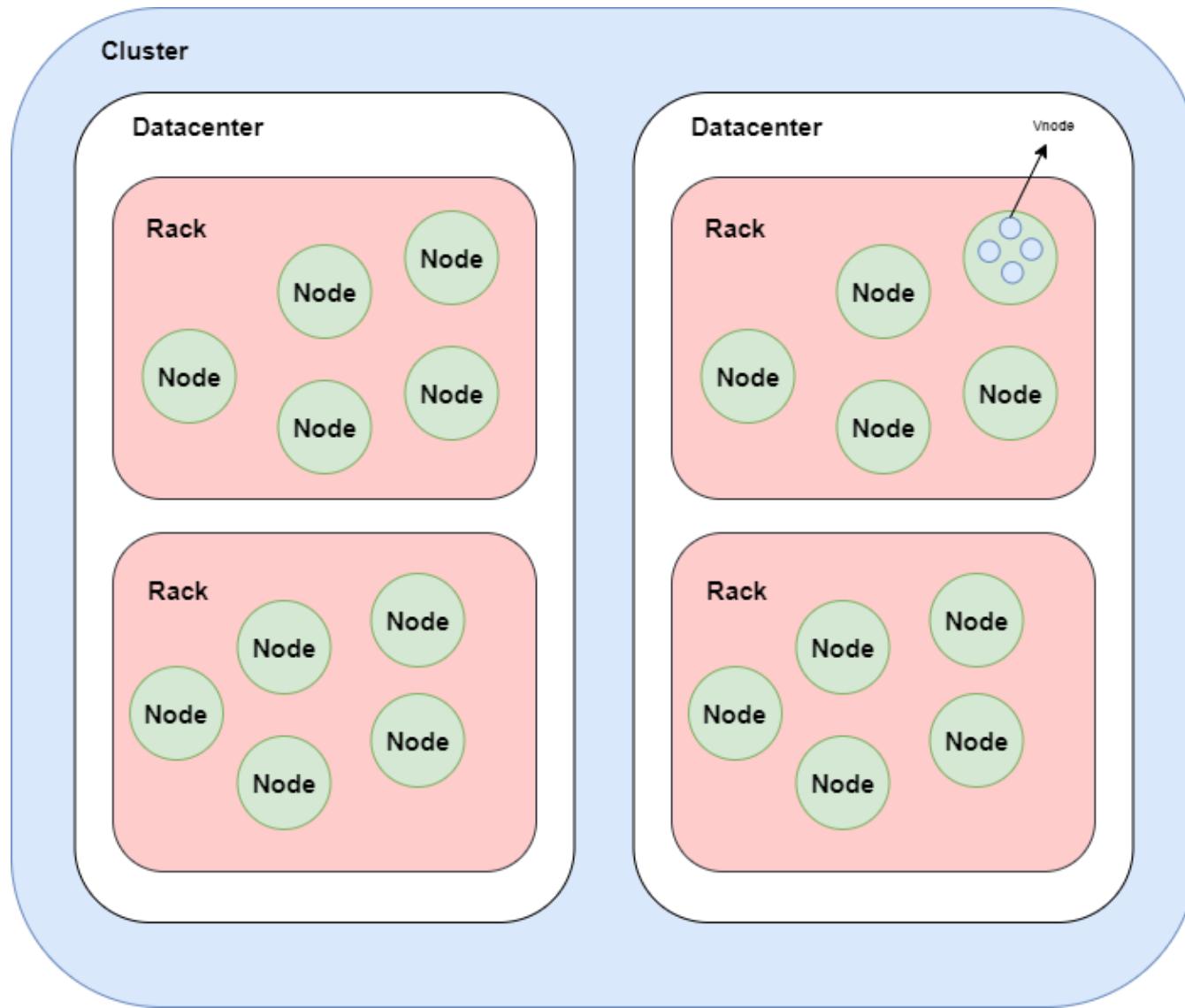
Hinted Hand Off

- There are two different mechanics at hand here. Let me give you an example to clarify.
- Assume I have a cluster of 3 nodes with 3 replication and consistency Quorum(2).
- This means that when I write to the database, I will have to get two responses to satisfy my query.
- After I satisfy my query, the node will have to send this write to the 3rd node.
- The node which has handled the query tries to send the write to the 3rd node.

Hinted Hand Off

- That node is currently unavailable, which causes the node to write a hinted handoff instead.
- Then it returns success to the client.
- Note that hinted handoff happens after the consistency has been met.
- The hinted handoff makes sure that the write gets to all nodes which hold a replica.
- Now there is one exception to this which is mentioned in the article you posted.
- Consistency level ANY is a consistency level that will be satisfied by writing a hinted handoff if there is a node which can serve the request

Multiple Data Center



Multiple Data Center

- Use system;
- Select data_center from local;
- CREATE KEYSPACE Excalibur WITH strategy_class = 'NetworkTopologyStrategy'
- AND strategy_options:DC1 = 2 AND strategy_options:DC2 = 2;



Multiple Data Center

- In the `cassandra-rackdc.properties` file, assign the datacenter and rack names you determined in the Prerequisites.
- For example:
 - Nodes 0 to 2
 - ## Indicate the rack and dc for this node
 - `dc=DC1`
 - `rack=RAC1`
 - Nodes 3 to 5



Multiple Data Center

- In the `cassandra-rackdc.properties` file, assign the datacenter and rack names you determined in the Prerequisites.
- For example:
- Nodes 0 to 2
 - ## Indicate the rack and dc for this node
 - `dc=DC1`
 - `rack=RAC1`
- Nodes 3 to 5
 - ## Indicate the rack and dc for this node
 - `dc=DC2`
 - `rack=RAC1`



Multiple Data Center

- The `GossipingPropertyFileSnitch` always loads `cassandra-topology.properties` when that file is present.
- Remove the file from each node on any new cluster or any cluster migrated from the `PropertyFileSnitch`.
- After you have installed and configured Cassandra on all nodes, DataStax recommends starting the seed nodes one at a time, and then starting the rest of the nodes.
- Note: If the node has restarted because of automatic restart, you must first stop the node and clear the directories, as described above.



Multiple Data Center

- The `GossipingPropertyFileSnitch` always loads `cassandra-topology.properties` when that file is present.
- Remove the file from each node on any new cluster or any cluster migrated from the `PropertyFileSnitch`.
- After you have installed and configured Cassandra on all nodes, DataStax recommends starting the seed nodes one at a time, and then starting the rest of the nodes.
- Note: If the node has restarted because of automatic restart, you must first stop the node and clear the directories, as described above.



Multiple Data Center

- Package installations:
- `sudo service cassandra start` #Starts Cassandra



Cluster Configuration

- https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/configuration/configCassandra_yaml.html

Materialized View

- Materialized view is work like a base table and it is defined as CQL query which can queried like a base table.
- Materialized view is very important for de-normalization of data in Cassandra Query Language is also good for high cardinality and high performance.

Materialized View

- CREATE TABLE User1.User_information (
- User_name text PRIMARY KEY,
- User_email text,
- User_password text,
- User_address text
-);

Materialized View

- CREATE MATERIALIZED VIEW [IF NOT EXISTS] [keyspace_name.] view_name
- AS SELECT column_list
- FROM [keyspace_name.] base_table_name
- WHERE column_name IS NOT NULL
 - [AND column_name IS NOT NULL ...]
 - [AND relation...]
- PRIMARY KEY (column_list)
- [WITH [table_properties]]
- [AND CLUSTERING ORDER BY (cluster_column_name order_option)]]



Materialized View

```
CREATE MATERIALIZED VIEW
User1.Users_by_User_email AS
SELECT User_email, User_password, User_address
FROM User_information
WHERE User_name IS NOT NULL AND User_email IS
NOT NULL
PRIMARY KEY (User_email, User_name );
```



Materialized View

```
SELECT *  
FROM Users_by_User_email  
WHERE User_email = ? ;
```

Materialized View

- In materialized view there are following restriction that must follow.
- In materialized view whatever the primary column in base table must contain in materialized view table that ensure every row of MV (materialized view) is correspond to the base table.
- In materialized view only we can add one more column that is not a primary column in base table.



What is a tombstone?

- In Cassandra, deleted data is not immediately purged from the disk.
- Instead, Cassandra writes a special value, known as a tombstone, to indicate that data has been deleted.
- Tombstones prevent deleted data from being returned during reads.
- It will eventually allow the data to be dropped via compaction.



What is a tombstone?

- Tombstones are writes – they go through the normal write path, take up space on disk, and make use of Cassandra's consistency mechanisms.
- Tombstones can be propagated across the cluster via hints and repairs.
- If a cluster is managed properly, this ensures that data will remain deleted even if a node is down when the delete is issued.

What is the normal lifecycle of tombstones?

- Tombstones are written with a timestamp.
- Under ideal circumstances, tombstones (and their associated data) will be dropped during compactions after a certain amount of time has passed.
- The following three criteria must be met for tombstones to be removed:
 - The tombstones were created more than `gc_grace_seconds` ago.
 - The table containing the tombstone is involved in a compaction.
 - All sstables that could contain the relevant data are involved in the compaction.

What is the normal lifecycle of tombstones?

- Each table has a `gc_grace_seconds` setting. By default, this is set to 864000, which is equivalent to 10 days.
- The intention is to provide time for the cluster to achieve consistency via repairs (and hence, prevent the resurrection of deleted data).

What is the normal lifecycle of tombstones?

- Size-Tiered Compaction Strategy will compact sstables of similar size together.
- Data tends to move into larger sstables as it ages, so the tombstone (in a new, small sstable) is unlikely to be compacted with the data (in an old, large sstable).
- Leveled Compaction Strategy is split into many levels that are compacted separately.
- The tombstone will be written into level 0.
- It will effectively ‘chase’ the data through the levels – it should eventually catch up.
- Time-Window Compaction Strategy (or Date-Tiered Compaction Strategy) will never compact the tombstone with the data if they are written into different time windows.



When do tombstones cause problems?

- When data is deleted, the space will not actually be freed for at least the gc_grace period set in the table settings.
- This can cause problems if a cluster is rapidly filling up.
- Under some circumstances, the space will never be freed without manual intervention.

When do tombstones cause problems?

- Read performance
- Serious performance problems can occur if reads encounter large numbers of tombstones.
- Performance issues are most likely to happen with the following types of query:
- Queries that run over all partitions in a table (“select * from keyspace.table”)
- Range queries (“select * from keyspace.table WHERE value > x”, or “WHERE value IN (value1, value2, ...)”)
- Any query that can only be run with an “ALLOW FILTERING” clause.



When do tombstones cause problems?

- These performance issues occur because of the behaviour of tombstones during reads.
- In a range query, your Cassandra driver will normally use paging, which allows nodes to return a limited number of responses at a time.
- When cassandra tombstones are involved, the node needs to keep the tombstones that it has encountered in memory and return them to the coordinator.
- In case one of the other replicas is unaware that the relevant data has been deleted.



When do tombstones cause problems?

- The tombstones cannot be paged because it is essential to return all of them, so latency and memory use increase proportionally to the number of tombstones encountered.
- Whether the tombstones will be encountered depends on the way the data is stored and retrieved.
- For example, if Cassandra is used to store data in a queue (which is not recommended), queries may encounter tens of thousands of tombstones to return a few rows of data.



How can I avoid tombstone issues?

- Avoid queries that will run on all partitions in the table (eg queries with no WHERE clause, or any query that requires ALLOW FILTERING).
- Alter range queries to avoid querying deleted data, or operate on a narrower range of data.
- Performance problems only occur if the tombstones are read, and scale with the number of tombstones read.
- Design the data model to avoid deleting large amounts of data.
- If planning to delete all the data in a table, truncate or drop the table to remove all the data without generating tombstones.



How can I avoid tombstone issues?

- Use a default time-to-live value.
- This only works efficiently if the primary key of your data is time-based, your data is written in chronological order, and the data will be deleted at a known date.
- To do this, set a default TTL in the table-level options, and set a time-based compaction strategy (`TimeWindowCompactionStrategy` if available, `DateTieredCompactionStrategy` otherwise).
- This will still create tombstones, but whole sstables will be efficiently dropped once the TTL on all of their contents have passed.



How can I get rid of existing tombstones?

- Under most circumstances, the best approach is to wait for the tombstone to compact away normally.
- If urgent performance or disk usage issues require more immediate action, there are two nodetool commands that can be used to force compactions, which can assist in dropping tombstones.
- These should be considered a last resort – in a healthy cluster with a well-designed data model, it is not necessary to run manual compactions.



How can I get rid of existing tombstones?

- Running nodetool compact forces a compaction of all sstables.
- This requires a large amount of free disk space.
- Keyspace and table arguments should be used to limit the compaction to the tables where tombstones are a problem.
- On tables where Size-Tiered Compaction Strategy is used, this command can lead to the creation of one enormous sstable.
- It will never have peers to compact with; if the –split-output flag is available, it should be used.



How can I get rid of existing tombstones?

- The nodetool garbagecollect command is available from Cassandra 3.10 onwards.
- This command runs a series of smaller compactions that also check overlapping sstables.
- It is more CPU intensive and time-consuming than nodetool compact, but requires less free disk space.



How can I get rid of existing tombstones?

- Tombstones will only be removed if `gc_grace_seconds` have elapsed since the tombstones were created.
- The intended purpose of `gc_grace_seconds` is to provide time for repairs to restore consistency to the cluster.
- Hence be careful when modifying it – prematurely removing tombstones can result in the resurrection of deleted data.



How can I get rid of existing tombstones?

- Also, the `gc_grace_seconds` setting affects expiration of hints generated for hinted handoff.
- It is dangerous to reduce `gc_grace_seconds` below the duration of the hinted handoff window (by default, 3 hours).



What is a tombstone?

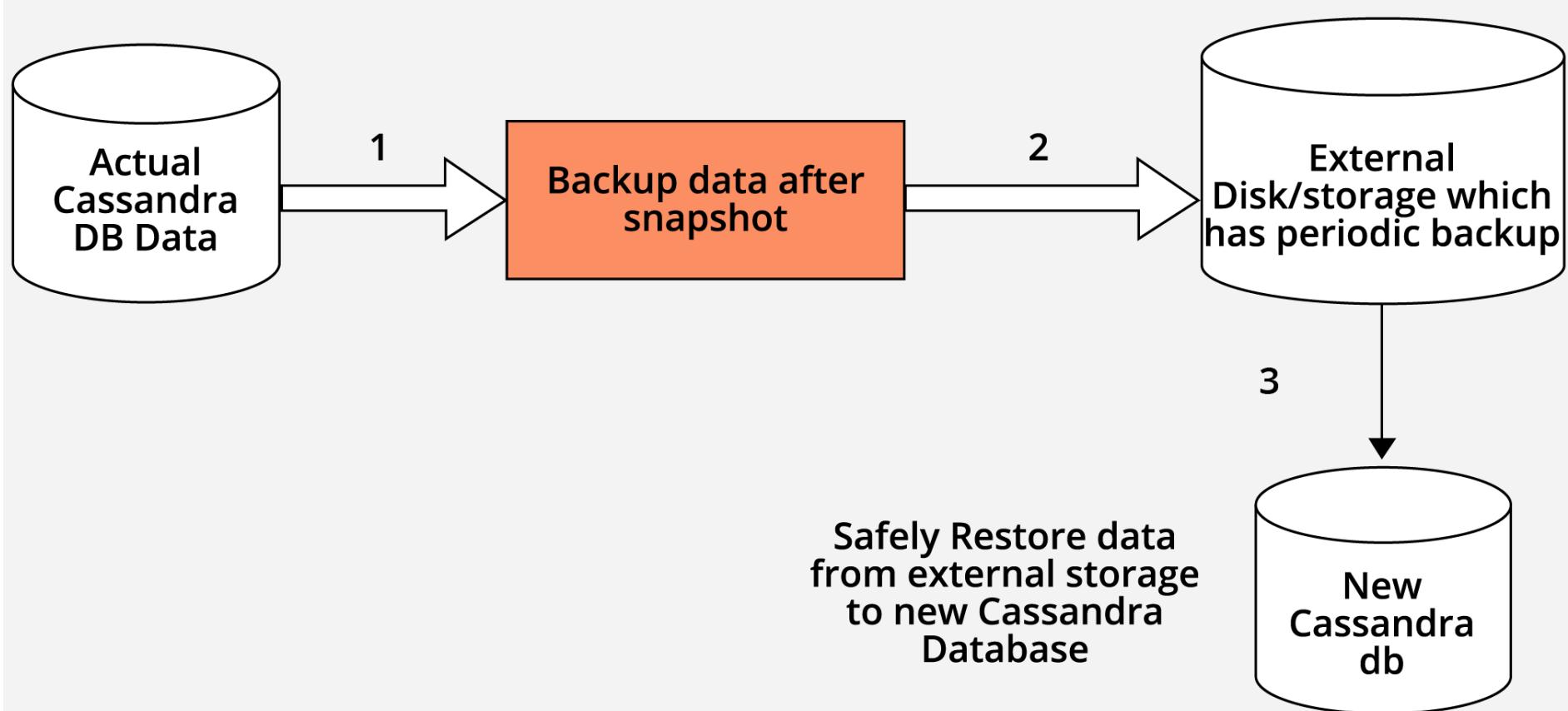
- Tombstones are generated by:
 - DELETE statements
 - Setting TTLs
 - Inserting null values
 - Inserting data into parts of a collection.

Understanding Backup & Restore Concepts in Cassandra



- Though Cassandra sounds unbeatable, backups are still necessary to recover during the following scenarios:
 - Disk failure
 - Corrupted data
 - Accidental deletions
 - Errors made in data by client applications
 - Rolling back the cluster to a known good state
 - Catastrophic failure that requires rebuilding the entire cluster

Understanding Backup & Restore Concepts in Cassandra





Types of Cassandra backup

- Cassandra provides two types of backup methods:
 - Snapshot based backup
 - Incremental backup

Snapshot

- A snapshot first flushes all in-memory writes to disk, then makes a hard link of the SSTable files for each keyspace.
- We must have enough free disk space on the node to accommodate making snapshots of your data files.
- A single snapshot requires little disk space.
- Snapshots can cause your disk usage to grow more quickly over time because a snapshot prevents old obsolete data files from being deleted.
- After the snapshot is complete, we can move the backup files to another location if needed, or we can leave them in place.



Snapshot based backup method

- Cassandra provides nodetool utility which is a command-line interface for managing a cluster.
- The nodetool utility gives a useful command for creating snapshots of the data.
- The nodetool snapshot command flushes memtables to the disk and creates a snapshot by creating a hard link to SSTables, which are immutable.
- The nodetool snapshot command takes snapshot per node basis.



Snapshot based backup method

- To take an entire cluster snapshot, the nodetool snapshot command should be run using a parallel ssh utility such as pssh.
- Alternatively, snapshot of each node can be taken one by one.
- It is possible to take a snapshot of all keyspaces in a cluster, or certain selected keyspaces, or a single table in a keyspace.
- Note that you must have enough free disk space on the node for taking a snapshot of your data files.



Snapshot Procedure

- Run nodetool cleanup to ensure that invalid replicas are removed.
 - nodetool cleanup <keyspace>
- Run the nodetool snapshot command, specifying the hostname, JMX port, and keyspace. For example:
- nodetool -h localhost -p 7199 snapshot mykeyspace

All keyspaces snapshot

- nodetool snapshot
- The snapshot directory is
`/var/lib/data/keyspace_name/table_name-UUID/
snapshots/`
- List snapshots
- nodetool listsnapshots

```
rps@rps-virtual-machine:/$ nodetool snapshot
Requested creating snapshot(s) for [all keyspaces] with snapshot name [1668090325958] and options {skipFlush=false}
Snapshot directory: 1668090325958
rps@rps-virtual-machine:/$
```

All keyspaces snapshot

```
rps@rps-virtual-machine: ~$ cd /var/lib
rps@rps-virtual-machine:/var/lib$ ls
AccountsService  boltd          emacsclient-common  ispell          PackageKit    sudo           unattended-upgrades  xffonts
acpi-support     BrAPI          fprint            libreoffice    pam           systemd       update-manager      xkb
alsa             cassandra      fwupd              locales        plymouth      tpm            update-notifier    xml-core
app-info          colord         gdm3              logrotate     polkit-1      ubiquity      upower
apport            command-not-found  geoclue          man-db        private       ubuntu-advantage
apt               dbus           ghostscript       misc          python       ubuntu-drivers-common
aspell            dhcp           grub              NetworkManager sgml-base    ubuntu-release-upgrader vim
avahi-autoipd    dictionaries-common  hp                openvpn      snapd        ucf           vmware
bluetooth        dpkg           initramfs-tools  os-prober    snmp         udisks2      whoopsie
rps@rps-virtual-machine:/var/lib$ cd cassandra
rps@rps-virtual-machine:/var/lib/cassandra$ ls
commitlog  data  hints  saved_caches
rps@rps-virtual-machine:/var/lib/cassandra$ cd data
rps@rps-virtual-machine:/var/lib/cassandra/data$ ls
abcbank  cqlkeyspace  globalbank  keyspace1  reservation  system_auth  system_schema
catalog  edutube      hotel      library    system       system_distributed  system_traces
rps@rps-virtual-machine:/var/lib/cassandra/data$ cd edutube
rps@rps-virtual-machine:/var/lib/cassandra/data/edutube$ ls
videos-443f50f0610e11ed882a21d0414c8520
rps@rps-virtual-machine:/var/lib/cassandra/data/edutube$ █
```



Single keyspace snapshot

- Assuming you created the keyspace university.
- To take a snapshot of the keyspace and you want a name of the snapshot, run the command below:
- `$ nodetool snapshot -t 2017.05.31 university`



Single table snapshot

- If you want to take a snapshot of only the student table in the university keyspace then run the command below:
- `$ nodetool snapshot --table student university`

Deleting snapshot files

- When taking a snapshot, previous snapshot files are not automatically deleted.
- We should remove old snapshots that are no longer needed.
- The **nodetool clearsnapshot –all** command removes all existing snapshot files from the snapshot directory of each keyspace.
- We should make it part of your back-up process to clear old snapshots before taking a new one.

Deleting snapshot files

- Procedure
- To delete all snapshots for a node, run the nodetool clearsnapshot command. For example:
`nodetool -h localhost -p 7199 clearsnapshot`
- To delete snapshots on all nodes at once, run the nodetool clearsnapshot command using a parallel ssh utility.
- To delete a single snapshot, run the clearsnapshot command with the snapshot name:
`nodetool clearsnapshot -t <snapshot_name>`
- The file name and path vary according to the type of snapshot.



Pros & Cons of Snapshot based backup

- Advantages:
 - Simple and much easier to manage
 - Cassandra nodetool utility provides nodetool clearsnapshot command which removes the snapshot files
- Disadvantages:
 - For large datasets, it may be hard to take a daily backup of the entire keyspace
 - It is expensive to transfer large snapshot data to a safe location like AWS S3



Incremental backup method

- By default, incremental backup is disabled in Cassandra.
- This can be enabled by changing the value of “incremental_backups” to “true” in the `cassandra.yaml` file.
- Once enabled, Cassandra creates a hard link to each memtable flushed to SSTable to a backup’s directory under the keyspace data directory.
- In Cassandra, incremental backups contain only new SSTable files, as they are dependent on the last snapshot created.
- Also, incremental backup requires less disk space as it only contains links to new SSTable files generated in the last full snapshot.



Pros & Cons of Incremental backup method

- Advantages:
 - Reduces disk space requirements
 - Reduces transfer cost
- Disadvantages:
 - Cassandra does not automatically clear incremental backup files. Removing hard-link files requires writing our own script, as there is no built-in tool to clear them.
 - Creates many small size files in backup, making file management and recovery cumbersome.
 - Cannot select a subset of column families for incremental backup.



Restoring from a snapshot

- Restoring a keyspace from a snapshot requires all snapshot files for the table.
- if using incremental backups, any incremental backup files created after the snapshot was taken.
- Streamed SSTables (from repair, decommission, and so on) are also hardlinked and included.
- Note: Restoring from snapshots and incremental backups temporarily causes intensive CPU and I/O activity on the node being restored.



Restoring from a snapshot

- Restoring from local nodes
- This method copies the SSTables from the snapshot's directory into the correct data directories.
- Make sure the table schema exists.
- Cassandra can only restore data from a snapshot when the table schema exists.
- If the schema does not exist and has not been backed up, you must recreate the schema.

Restoring from a snapshot

- If necessary, truncate the table.
 - Note: Need not to truncate under certain conditions.
 - For example, if a node lost a disk, we might restart before restoring so that the node continues to receive new writes before starting the restore procedure.
- Truncating is usually necessary.
 - For example, if there was an accidental deletion of data, the tombstone from that delete has a later write timestamp than the data in the snapshot.
 - If you restore without truncating (removing the tombstone), Cassandra continues to shadow the restored data.
 - This behavior also occurs for other types of overwrites and causes the same problem.



Restoring from a snapshot

- Locate the most recent snapshot folder.
- For example:
 - `data_directory/keyspace_name/table_name-UUID/snapshots/snapshot_name`
- Copy the most recent snapshot SSTable directory to the `data_directory/keyspace/table_name-UUID` directory.
- Run nodetool refresh and restart your nodes.



Restoring from centralized backups

- This method uses sstableloader to restore snapshots.
 - Make sure the table schema exists.
 - Cassandra can only restore data from a snapshot when the table schema exists.
 - If the schema does not exist and has not been backed up, you must recreate the schema.
 - Restore the most recent snapshot using the sstableloader tool on the backed-up SSTables.
 - The sstableloader streams the SSTables to the correct nodes.
 - Need not to remove the commitlogs or drain or restart the nodes.



Restoring from centralized backups

```
cqlsh> quit
rps@rps-virtual-machine:/var/lib/cassandra/data/edutube$ sstableloader -d 172.20.0.161 /var/lib/cassandra/data/edutube/videos-443f50f0610e11e
d882a21d0414c8520/
Established connection to initial hosts
Opening sstables and calculating sections to stream

Summary statistics:
  Connections per host      : 1
  Total files transferred   : 0
  Total bytes transferred   : 0.000KiB
  Total duration            : 2909 ms
  Average transfer rate    : 0.000KiB/s
  Peak transfer rate        : 0.000KiB/s

rps@rps-virtual-machine:/var/lib/cassandra/data/edutube$ █
```



Restoring from centralized backups

```
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520/backups$ nodetool snapshot -t edutube_keyspace_backup edutube
Requested creating snapshot(s) for [edutube] with snapshot name [edutube_keyspace_backup] and options {skipFlush=false}
Snapshot directory: edutube_keyspace_backup
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520/backups$ ls
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520/backups$ ls
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520/backups$ cd ..
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520$ ls
backups          nb-4-big-Data.db      nb-4-big-Filter.db    nb-4-big-Statistics.db   nb-4-big-TOC.txt
nb-4-big-CompressionInfo.db nb-4-big-Digest.crc32 nb-4-big-Index.db    nb-4-big-Summary.db   snapshots
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520$ cd snapshots
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520/snapshots$ ls
edutube_keyspace_backup
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520/snapshots$ cd ..
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520$ cd ..
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube$ ls
edusnapshot edutubesnap import_edutube_videos.err videos-443f50f0610e11ed882a21d0414c8520
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube$ cd videos-443f50f0610e11ed882a21d0414c8520
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520$ ls
backups          nb-4-big-Data.db      nb-4-big-Filter.db    nb-4-big-Statistics.db   nb-4-big-TOC.txt
nb-4-big-CompressionInfo.db nb-4-big-Digest.crc32 nb-4-big-Index.db    nb-4-big-Summary.db   snapshots
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520$ cd snapshots
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520/snapshots$ ls
edutube_keyspace_backup
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520/snapshots$ ls
edutube_keyspace_backup
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520/snapshots$ cd edutube_keyspace_backup
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520/snapshots/edutube_keyspace_backup$ ls
manifest.json      nb-4-big-Data.db      nb-4-big-Filter.db    nb-4-big-Statistics.db   nb-4-big-TOC.txt
nb-4-big-CompressionInfo.db nb-4-big-Digest.crc32 nb-4-big-Index.db    nb-4-big-Summary.db   schema.cql
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520/snapshots/edutube_keyspace_backup$ cd ..
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520/snapshots$ cd ..
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520$ sudo cp ./snapshots/edutube_keyspace_backup/
.
cp: cannot stat '/snapshots/edutube_keyspace_backup/': No such file or directory
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520$ sudo cp ./snapshots/edutube_keyspace_backup/
```



Restoring from centralized backups

```
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520/snapshots/edutube_keyspace_backup$ cd ..
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520/snapshots$ cd ..
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520$ sudo cp ./snapshots/edutube_keyspace_backup/
.
cp: cannot stat '/snapshots/edutube_keyspace_backup/': No such file or directory
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520$ sudo cp ./snapshots/edutube_keyspace_backup/
.
cp: -r not specified; omitting directory './snapshots/edutube_keyspace_backup/'
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520$ sudo cp -r ./snapshots/edutube_keyspace_back
up/
.
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520$ ls
backups          nb-4-big-CompressionInfo.db  nb-4-big-Digest crc32  nb-4-big-Index.db      nb-4-big-Summary.db  snapshots
edutube_keyspace_backup nb-4-big-Data.db        nb-4-big-Filter.db    nb-4-big-Statistics.db nb-4-big-TOC.txt
rps@rps-virtual-machine: /var/lib/cassandra/data/edutube/videos-443f50f0610e11ed882a21d0414c8520$ nodetool repair edutube
[2022-11-11 07:00:26,370] Starting repair command #10 (6b4fa5d0-6160-11ed-882a-21d0414c8520), repairing keyspace edutube with repair options
(parallelism: parallel, primary range: false, incremental: true, job threads: 1, ColumnFamilies: [], dataCenters: [], hosts: [], previewKind:
NONE, # of ranges: 48, pull repair: false, force repair: false, optimise streams: false, ignore unreplicated keyspaces: false)
[2022-11-11 07:00:26,496] Repair session 6b5ee810-6160-11ed-882a-21d0414c8520 for range [(7406352872789520534, 7767342626330269001], (-6831984
44775388407, 257737511060817260], (7325754139062624524, 7406352872789520534], (-5496906696259102723, -5207556498107599823], (-405914043286134566
2, -3661383358904324913], (-3271882739924225807, -2967843322169133402], (3030549823752776033, 3299581390145001754], (5627718571965933406, 6177338
073816370363], (-770338977520250076, -760113893681609755], (222382550891942215, 2887081017609641234], (-3661383358904324913, -32718827399242258
071], (257737511060817260, 388302171564644405], (-6933816633919871080, -6568407312599135535], (-6568407312599135535, -6303598398342538112], (4035
497082855895395, 4649573119449819330], (1075409443673168054, 1406153576307071420], (3966453825057242892, 4035497082855895395], (4649573119449819
330, 5114869890303497053], (8502394773024458823, 8915758691576523162], (-8082232699166125241, -8008675542501718775], (-258562747767845373, -2123
466674677971646], (1406153576307071420, 1440545271238877395], (-5207556498107599823, -5155182333096283951], (5114869890303497053, 52099738307378
34864], (8915758691576523162, -9060378151682239317], (-8853052189992880533, -8082232699166125241], (-4116259387415387563, -4059140432861345662],
(-8008675542501718775, -7911962086435985156], (2151165324898747593, 222382550891942215], (6358389895984089025, 7325754139062624524], (-7601138
93681609755, -683198444775388407], (388302171564644405, 473181028160341896], (1440545271238877395, 2151165324898747593], (-1849711784967851733, -1831614510021642567],
(7767342626330269001, 8502394773024458823], (-1831614510021642567, -770338977520250076], (5209973830737834864, 56277185719
65933406], (-2123466674677971646, -1849711784967851733], (-7911962086435985156, -6933816633919871080], (-4479055291516675707, -41162593874153875
63], (-9060378151682239317, -8853052189992880533], (2887081017609641234, 3030549823752776033], (-6303598398342538112, -5496906696259102723], (-2
967843322169133402, -2585627477678745373], (6177338073816370363, 6358389895984089025], (473181028160341896, 1075409443673168054], (3299581390145
001754, 3966453825057242892], (-5155182333096283951, -4479055291516675707]) finished (progress: 7%)
[2022-11-11 07:00:26,608] Repair completed successfully
[2022-11-11 07:00:26,610] Repair command #10 finished in 0 seconds
[2022-11-11 07:00:26,615] condition satisfied queried for parent session status and discovered repair completed.
[2022-11-11 07:00:26,615] Repair completed successfully
```



Monitoring Cassandra

- nodetool tablestats displays statistics for each table and keyspace.
- nodetool tablehistograms provides statistics about a table, including read/write latency, row size, column count, and number of SSTables.
- nodetool netstats provides statistics about network operations and connections.
- nodetool tpstats provides statistics about the number of active, pending, and completed tasks for each stage of database operations by thread pool.



Monitoring Cassandra

- nodetool proxyhistograms
- Jconsole
 - Memory
 - Displays information about memory use.
 - Threads
 - Displays information about thread use.
 - Classes
 - Displays information about class loading.
 - VM Summary
 - Displays information about the Java Virtual Machine (VM).
 - Mbeans
 - Displays information about MBeans.



Monitoring Cassandra

- For individual tables, ColumnFamilyStoreMBean provides the same general latency attributes as StorageProxyMBean.

Attribute	Description
MemtablePageSize	The total size consumed by this table's data (not including metadata).
MemtableColumnsCount	Returns the total number of columns present in the memtable (across all keys).
MemtableSwitchCount	How many times the memtable has been flushed out.
RecentReadLatencyMicros	The average read latency since the last call to this bean.
RecentWriterLatencyMicros	The average write latency since the last call to this bean.
LiveSSTableCount	The number of live SSTables for this table.



Monitoring Cassandra

- Thread pool and read/write latency statistics.
- Cassandra maintains distinct thread pools for different stages of execution.
- Each of the thread pools provide statistics on the number of tasks that are active, pending, and completed. Trends on these pools for increases in the pending tasks column indicate when to add additional capacity.
- After a baseline is established, configure alarms for any increases above normal in the pending tasks column.



Monitoring Cassandra

- Use nodetool tpstats on the command line to view the thread pool details

Thread Pool	Description
AntiEntropyStage	Tasks related to repair
CacheCleanupExecutor	Tasks related to cache maintenance (counter cache, row cache)
CompactionExecutor	Tasks related to compaction
CounterMutationStage	Tasks related to leading counter writes
GossipStage	Tasks related to the gossip protocol
HintsDispatcher	Tasks related to sending hints
InternalResponseStage	Tasks related to miscellaneous internal task responses
MemtableFlushWriter	Tasks related to flushing memtables



Monitoring Cassandra

- Use nodetool tpstats on the command line to view the thread pool details

MemtablePostFlush	Tasks related to maintenance after memtable flush completion
MemtableReclaimMemory	Tasks related to reclaiming memtable memory
MigrationStage	Tasks related to schema maintenance
MiscStage	Tasks related to miscellaneous tasks, including snapshots and removing hosts
MutationStage	Tasks related to writes
Native-Transport-Requests	Tasks related to client requests from CQL
PendingRangeCalculator	Tasks related to recalculating range ownership after bootstraps/decommissions
PerDiskMemtableFlushWriter_*	Tasks related to flushing memtables to a given disk
ReadRepairStage	Tasks related to performing read repairs



Monitoring Cassandra

- Use nodetool tpstats on the command line to view the thread pool details

ReadStage	Tasks related to reads
RequestResponseStage	Tasks for callbacks from intra-node requests
Sampler	Tasks related to sampling statistics
SecondaryIndexManagement	Tasks related to secondary index maintenance
ValidationExecutor	Tasks related to validation compactions
ViewMutationStage	Tasks related to maintaining materialized views

Monitoring Cassandra

- Compaction metrics
- Monitoring compaction performance is an important aspect of knowing when to add capacity to your cluster.
- The following attributes are exposed through CompactionManagerMBean:

Attribute	Description
BytesCompacted	Total number of bytes compacted since server [re]start
CompletedTasks	Number of completed compactions since server [re]start
PendingTasks	Estimated number of compactions remaining to perform
TotalCompactionsCompleted	Total number of compactions since server [re]start

Tuning Java resources

- For Cassandra 3.0 and later, using the Concurrent-Mark-Sweep (CMS) or G1 garbage collector depends on these factors:
 - G1 is recommended in the following circumstances and reasons:
 - Heap sizes from 16 GB to 64 GB.
- G1 performs better than CMS for larger heaps because it scans the regions of the heap containing the most garbage objects first, and compacts the heap on-the-go, while CMS stops the application when performing garbage collection.
- The workload is variable, that is, the cluster is performing the different processes all the time.
- For future proofing, as CMS will be deprecated in Java 9.
 - G1 is easier to configure.
 - G1 is self tuning.



Tuning Java resources

- CMS is recommended in the following circumstances:
- Allocating more memory to the heap, can result in diminishing performance as the garbage collection facility increases the amount of Cassandra metadata in heap memory.
- Heap sizes no larger than 16 GB.
- The workload is fixed, that is, the cluster performs the same processes all the time.
- The environment requires the lowest latency possible. G1 incurs some latency due to profiling.



Tuning Java resources

- Setting G1 as the Java garbage collector
 - Open jvm.options.
 - Comment out the -Xmn800M line.
 - Comment out all lines in the ### CMS Settings section.
 - Uncomment the relevant G1 settings in the ### G1 Settings section:
 - ## Use the Hotspot garbage-first collector.
 - -XX:+UseG1GC
 - #
 - ## Have the JVM do less remembered set work during STW, instead
 - ## preferring concurrent GC. Reduces p99.9 latency.
 - #-XX:G1RSetUpdatingPauseTimePercent=5

Tuning Java resources

- Determining the heap size
 - Set the Java heap to consume the majority of the computer's RAM.
 - It can interfere with the operation of the OS page cache.
 - Recent operating systems maintain the OS page cache for frequently accessed data and are very good at keeping this data in memory.
 - Properly tuning the OS page cache usually results in better performance than increasing the Cassandra row cache.
 - Cassandra automatically calculates the maximum heap size (`MAX_HEAP_SIZE`) based on this formula:
 - $\max(\min(1/2 \text{ ram}, 1024\text{MB}), \min(1/4 \text{ ram}, 8\text{GB}))$



Tuning Java resources

- For production use, you may wish to adjust heap size for your environment using the following guidelines:
 - Heap size is usually between $\frac{1}{4}$ and $\frac{1}{2}$ of system memory.
 - Do not devote all memory to heap because it is also used for offheap cache and file system cache.
 - Always enable GC logging when adjusting GC.
 - Adjust settings gradually and test each incremental change.
 - Enable parallel processing for GC.



Tuning Java resources

- Cassandra's GCInspector class logs information about any garbage collection that takes longer than 200 ms.
- Garbage collections that occur frequently and take a moderate length of time (seconds) to complete, indicate excessive garbage collection pressure on the JVM.
- In addition to adjusting the garbage collection options, other remedies include adding nodes, and lowering cache sizes.
- For a node using G1, the Cassandra community recommends a MAX_HEAP_SIZE as large as possible, up to 64 GB.

Tuning Java resources

- MAX_HEAP_SIZE
- The recommended maximum heap size depends on which GC is used:

Hardware setup	Recommended MAX_HEAP_SIZE
Older computers	Typically 8 GB.
CMS for newer computers (8+ cores) with up to 256 GB RAM	No more 16 GB.
G1 for newer computers (8+ cores) with up to 256 GB RAM	16 GB to 64 GB.



Tuning Java resources

- The easiest way to determine the optimum heap size for your environment is:
- Set the maximum heap size in the `jvm.options` file to a high arbitrary value on a single node. For example, when using G1:
 - `-Xms48G`
 - `-Xmx48G`
 - Set the min (`-Xms`) and max (`-Xmx`) heap sizes to the same value to avoid stop-the-world GC pauses during resize, and to lock the heap in memory on startup which prevents any of it from being swapped out.
- Enable GC logging.
- Check the logs to view the heap used by that node and use that value for setting the heap size in the cluster:



Tuning Java resources

- **HEAP_NEWSIZE**
 - For CMS, you may also need to adjust **HEAP_NEWSIZE**.
 - This setting determines the amount of heap memory allocated to newer objects or young generation.
 - Cassandra calculates the default value for this property (in MB) as the lesser of:
 - 100 times the number of cores
 - $\frac{1}{4}$ of **MAX_HEAP_SIZE**
 - As a starting point, set **HEAP_NEWSIZE** to 100 MB per physical CPU core.
 - For example, for a modern 8-core+ machine:
 - `-Xmn800M`



Tuning Java resources

- Cassandra performs the following major operations within JVM heap:
 - To perform reads, Cassandra maintains the following components in heap memory:
 - Bloom filters
 - Partition summary
 - Partition key cache
 - Compression offsets
 - SSTable index summary



Tuning Java resources

- Cassandra gathers replicas for a read or for anti-entropy repair and compares the replicas in heap memory.
- Data written to Cassandra is first stored in memtables in heap memory.
- Memtables are flushed to SSTables on disk.



Tuning Java resources

- To improve performance, Cassandra also uses off-heap memory as follows:
- Page cache. Cassandra uses additional memory as page cache when reading files on disk.
- The Bloom filter and compression offset maps reside off-heap.
- Cassandra can store cached rows in native memory, outside the Java heap.
- This reduces JVM heap requirements, which helps keep the heap size in the sweet spot for JVM garbage collection performance.



Data Caching

- Cassandra includes integrated caching and distributes cache data around the cluster for you.
- The integrated cache solves the cold start problem by virtue of saving your cache to disk periodically.
- We can read contents back in when it restarts.
- We never have to start with a cold cache.



Data Caching

- There are two layers of cache:
 - Partition key cache
 - Row cache



Data Caching

- Configuring Key and Row Caches
- Key caching is enabled by default in Cassandra, and high levels of key caching are recommended for most scenarios.
- Cases for row caching are more specialized, but whenever it can coexist peacefully with other demands on memory resources, row caching provides the most dramatic gains in efficiency.

Data Caching

- Key Cache
- The key cache holds the location of keys in memory on a per-column family basis.
- For column family level read optimizations, turning this value up can have an immediate impact as soon as the cache warms.
- Key caching is enabled by default, at a level of 200,000 keys.

Data Caching

- Row Cache
- Unlike the key cache, the row cache holds the entire contents of the row in memory.
- It is best used when you have a small subset of data to keep hot and you frequently need most or all of the columns returned.
- For these use cases, row cache can have substantial performance benefits.

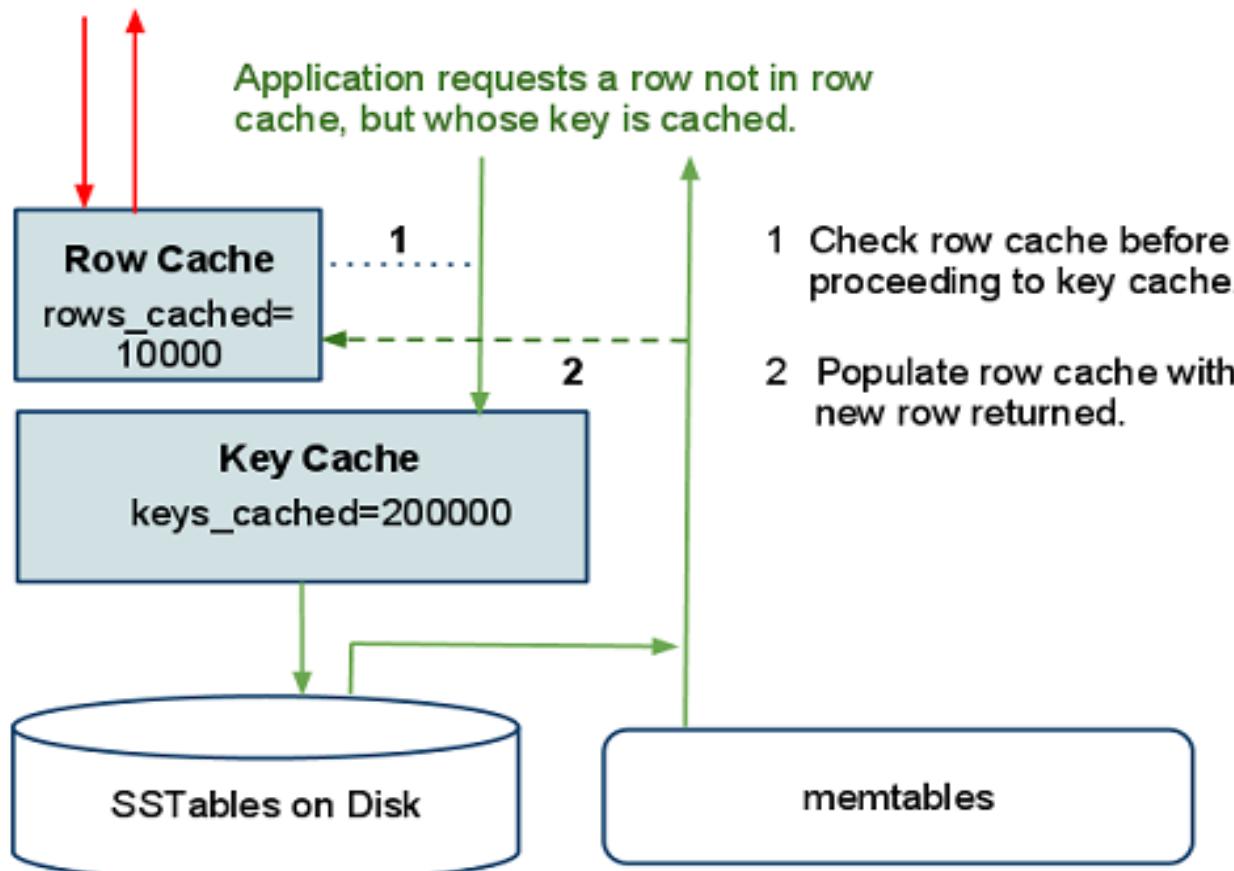


Key and Row Cache Hits at Runtime

- In a scenario where both row and key caches are configured, the row cache will return results whenever possible.
- In the case of a row cache miss, the key cache may still provide a hit that makes the disk seek much more efficient.
- In this example with both caches already populated, two read operations on a column family are depicted:

Key and Row Cache Hits at Runtime

Application requests a "hot," frequently accessed row.





Key and Row Cache Hits at Runtime

- One read operation hits the row cache, returning the requested row without a disk seek.
- The other read operation requests a row that is not present in the row cache but is present in the key cache.
- After accessing the row in the SSTable, the system returns the data and populates the row cache with this read operation.



Data Modeling for Cache Performance

- If your requirements permit it, a data model that logically separates heavily-read data into discrete column families can help optimize caching.
- Column families with relatively small, “narrow” rows lend themselves to highly efficient row caching.
- By the same token, it can make sense to separately store lower-demand data, or data with extremely long rows, in a column family with minimal caching, if any.
- Row caching in such contexts brings the most benefit when access patterns follow a normal (Gaussian) distribution.
- When the keys most frequently requested follow such patterns, cache hit rates tend to increase.
- If you have particularly “hot” rows in your data model, row caching can bring significant performance improvements



Tuning for Optimal Caching

- Careful, incremental testing is essential to maximizing benefit from Cassandra's caching features.
- Adjustments that increase your cache hit rate are likely to decrease the system resources available for your write load and other operations.
- After making changes to cache configuration, it is best to monitor Cassandra as a whole for unintended impact on the system.
- The jconsole GUI can be a helpful tool for monitoring caching metrics exposed through JMX.
- For each node and each column family, you can view your cache hit rate, cache size, and number of hits by expanding org.apache.cassandra.db in the MBeans tab.

Tuning for Optimal Caching

184.106.228.116:8080

Overview Memory Threads Classes VM Summary MBeans

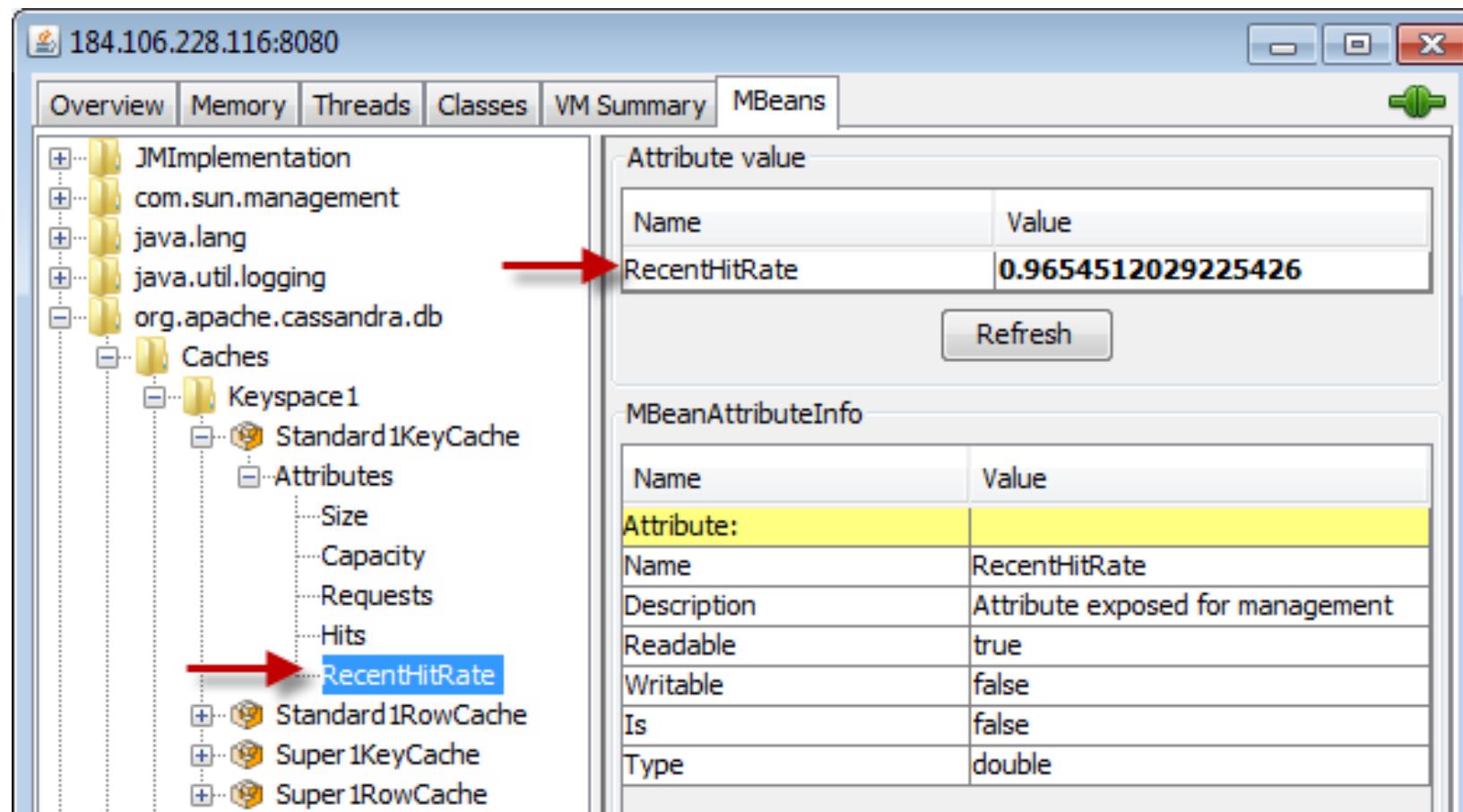
Attribute value

Name	Value
RecentHitRate	0.9654512029225426

Refresh

MBeanAttributeInfo

Name	Value
Attribute:	
Name	RecentHitRate
Description	Attribute exposed for management
Readable	true
Writable	false
Is	false
Type	double





Tuning for Optimal Caching

```
rps@rps-virtual-machine:/etc/cassandra$ nodetool info
ID : 8b5b4796-9366-4b33-a4e4-d27f015d0e88
Gossip active : true
Native Transport active: true
Load : 106.74 MiB
Generation No : 1668152404
Uptime (seconds) : 7961
Heap Memory (MB) : 668.47 / 7946.00
Off Heap Memory (MB) : 0.99
Data Center : DC1
Rack : RAC1
Exceptions : 0
Key Cache : entries 85, size 7.68 KiB, capacity 100 MiB, 150 hits, 239 requests, 0.628 recent hit rate, 14400 save period in
  seconds
Row Cache : entries 0, size 0 bytes, capacity 0 bytes, 0 hits, 0 requests, NaN recent hit rate, 0 save period in seconds
Counter Cache : entries 0, size 0 bytes, capacity 50 MiB, 0 hits, 0 requests, NaN recent hit rate, 7200 save period in seconds
Percent Repaired : 2.4094871593497214%
Token : (invoke with -T/--tokens to see all 16 tokens)
rps@rps-virtual-machine:/etc/cassandra$
```

Ch
from
bac
from



Data Caching

- Use CQL to enable or disable caching by configuring the caching table property.
- Set parameters in the `cassandra.yaml` file to configure global caching properties:
 - Partition key cache size
 - Row cache size
 - How often Cassandra saves partition key caches to disk
 - How often Cassandra saves row caches to disk



Data Caching

- Configuring the `row_cache_size_in_mb` (in the `cassandra.yaml` configuration file) determines how much space in memory Cassandra allocates to store rows from the most frequently read partitions of the table.



Data Caching

```
CREATE TABLE users (
    userid text PRIMARY KEY,
    first_name text,
    last_name text,
)
WITH caching = { 'keys' : 'NONE', 'rows_per_partition' :
    '120' };
```



Data Caching

- Tips for Efficient Cache Use
- Some tips for efficient cache use are:
 - Store lower-demand data or data with extremely long rows in a table with minimal or no caching.
 - Deploy a large number of Cassandra nodes under a relatively light load per node.
 - Logically separate heavily-read data into discrete tables



Monitoring and adjusting caching

- Make changes to cache options in small, incremental adjustments, then monitor the effects of each change using the nodetool utility.



Monitoring and adjusting caching

- The output of the nodetool info command shows the following row cache and key cache setting values, which are configured in the `cassandra.yaml` file:
 - Cache size in bytes
 - Capacity in bytes
 - Number of hits
 - Number of requests
 - Recent hit rate
 - Duration in seconds after which Cassandra saves the key cache.



Monitoring and adjusting caching

- ID : 387d15ba-7103-491b-9327-1a691dbb504a
- Gossip active : true
- Thrift active : true
- Native Transport active: true
- Load : 65.87 KB
- Generation No : 1400189757
- Uptime (seconds) : 148760



Monitoring and adjusting caching

- Heap Memory (MB) : 392.82 / 1996.81
- datacenter : datacenter1
- Rack : rack1
- Exceptions : 0
- Key Cache : entries 10, size 728 (bytes), capacity 103809024 (bytes), 93 hits, 102 requests, 0.912 recent hit rate, 14400 save period in seconds
- Row Cache : entries 0, size 0 (bytes), capacity 0 (bytes), 0 hits, 0 requests, NaN recent hit rate, 0 save period in seconds
- Counter Cache : entries 0, size 0 (bytes), capacity 51380224 (bytes), 0 hits, 0 requests, NaN recent hit rate, 7200 save period in seconds
- Token : -9223372036854775808

Sstable loader

- The Cassandra bulk loader, also called the `sstableloader`, provides the ability to:
 - Bulk load external data into a cluster.
 - Load existing SSTables into another cluster with a different number of nodes or replication strategy.
 - Restore snapshots.

Sstable loader

- The sstableloader streams a set of SSTable data files to a live cluster.
- It does not simply copy the set of SSTables to every node, but transfers the relevant part of the data to each node, conforming to the replication strategy of the cluster.
- The table into which the data is loaded does not need to be empty.



Sstable loader

- Run `sstableloader` specifying the path to the SSTables and passing it the location of the target cluster.
- When using the `sstableloader` be aware of the following:
 - Bulkloading SSTables created in versions prior to Cassandra 3.0 is supported only in Cassandra 3.0.5 and later.
 - Repairing tables that have been loaded into a different cluster does not repair the source tables.

Sstable loader

- Prerequisites
 - The source data loaded by sstableloader must be in SSTables.
 - Because sstableloader uses the streaming protocol, it requires a direct connection over the port 7000 (storage port) to each connected node.

Sstable loader

- Generating SStables
 - When using `sstableloader` to load external data, you must first put the external data into SSTables.
 - `SSTableWriter` is the API to create raw Cassandra data files locally for bulk load into your cluster.
 - The Cassandra source code includes the `CQLSSTableWriter` implementation for creating SSTable files from external data.
 - We need not to understand the details of how those map to the underlying storage engine.
 - Import the `org.apache.cassandra.io.sstable.CQLSSTableWriter` class, and define the schema for the data you want to import, a writer for the schema, and a prepared insert statement.



Sstable loader

- \$ sstableloader -d host_url (,host_url ...) [options]
path_to_keyspace

Sstable loader

- `sstablemetadata`
 - The `sstablemetadata` utility prints metadata about a specified SSTable, including:
 - SSTable name
 - partitioner
 - SSTable level (for Leveled Compaction only)
 - number of tombstones and Dropped timestamps (in epoch time)
 - number of cells and size (in bytes) per row



Sstable loader

- Sudo apt install Cassandra-tools
- Switch to the CASSANDRA_HOME directory.
- Enter the command /tools/bin/sstablemetadata followed by the filenames of one or more SSTables.
- \$ tools/bin/sstablemetadata <sstable_name filenames>



Sstable loader

```
at org.apache.cassandra.tools.SSTableMetadataViewer.printSSTableMetadata(SSTableMetadataViewer.java:520)
at org.apache.cassandra.tools.SSTableMetadataViewer.main(SSTableMetadataViewer.java:546)
rps@rps-virtual-machine:/var/lib/cassandra/data/hotel/hotels_by.poi-76aa5a705ff711eda675658e3c9138f4$ sstablemetadata /var/lib/cassandra/data
/hotel/hotels_by.poi-76aa5a705ff711eda675658e3c9138f4/nb-7-big/Index.db
SSTable: /var/lib/cassandra/data/hotel/hotels_by.poi-76aa5a705ff711eda675658e3c9138f4/nb-7-big
Partitioner: org.apache.cassandra.dht.Murmur3Partitioner
Bloom Filter FP chance: 0.01
Minimum timestamp: 1667976024136617 (11/09/2022 12:10:24)
Maximum timestamp: 1667976024136618 (11/09/2022 12:10:24)
SSTable min local deletion time: 1667976024 (11/09/2022 12:10:24)
SSTable max local deletion time: 2147483647 (no tombstones)
Compressor: org.apache.cassandra.io.compress.LZ4Compressor
Compression ratio: 0.7314814814815
TTL min: 0
TTL max: 0
First token: 2724029245854047125 (Bangalore)
Last token: 2724029245854047125 (Bangalore)
minClusteringValues: [103]
maxClusteringValues: [102]
Estimated droppable tombstones: 0.14285714285714285
SSTable Level: 0
Repaired at: 1668024269290 (11/10/2022 01:34:29)
Pending repair: --
Replay positions covered: {CommitLogPosition(segmentId=1667747778424, position=361435)=CommitLogPosition(segmentId=1667747778424, position=48
2164)}
totalColumnsSet: 6
totalRows: 2
Estimated tombstone drop times:
Drop Time          | Count (%) Histogram
1667976060 (11/09/2022 12:11:00) |    2 (100) 0000000000000000000000000000000000000000
Percentiles
50th      1996099046 (04/03/2033 05:27:26)
75th      1996099046 (04/03/2033 05:27:26)
95th      1996099046 (04/03/2033 05:27:26)
98th      1996099046 (04/03/2033 05:27:26)
99th      1996099046 (04/03/2033 05:27:26)
Min       1663415873 (09/17/2022 17:27:53)
```

Batch

```
BEGIN UNLOGGED BATCH
  USING TIMESTAMP timestamp
  dml_statement;
  dml_statement;
  ...
APPLY BATCH;
```

Batch

- A BATCH statement combines multiple data modification language (DML) statements (INSERT, UPDATE, DELETE) into a single logical operation, and sets a client-supplied timestamp for all columns written by the statements in the batch.
- Batching multiple statements can save network exchanges between the client/server and server coordinator/replicas.
- However, because of the distributed nature of Cassandra, spread requests across nearby nodes as much as possible to optimize performance.

Batch

- Batches are atomic by default. In the context of a Cassandra batch operation, atomic means that if any of the batch succeeds, all of it will.
- To achieve atomicity, Cassandra first writes the serialized batch to the batchlog system table that consumes the serialized batch as blob data.
- When the rows in the batch have been successfully written and persisted (or hinted) the batchlog data is removed.
- There is a performance penalty for atomicity. If you do not want to incur this penalty, prevent Cassandra from writing to the batchlog system by using the UNLOGGED option:
BEGIN UNLOGGED BATCH

Batch

- BEGIN BATCH
- INSERT INTO purchases (user, balance) VALUES ('user1', -8) USING TIMESTAMP 19998889022757000;
- INSERT INTO purchases (user, expense_id, amount, description, paid)
- VALUES ('user1', 1, 8, 'burrito', false);
- APPLY BATCH;

Batch

- BEGIN BATCH
- INSERT INTO purchases (user, balance) VALUES ('user1', -8) IF NOT EXISTS;
- INSERT INTO purchases (user, expense_id, amount, description, paid)
VALUES ('user1', 1, 8, 'burrito', false);
- APPLY BATCH;
-
- BEGIN BATCH
- UPDATE purchases SET balance = -208 WHERE user='user1' IF balance = -8;
- INSERT INTO purchases (user, expense_id, amount, description, paid)
VALUES ('user1', 2, 200, 'hotel room', false);
- APPLY BATCH;

Batch

- BEGIN BATCH USING TIMESTAMP 1481124356754405
-
- INSERT INTO cycling.cyclist_expenses (
- cyclist_name, expense_id, amount, description, paid
-) VALUES (
- 'Vera ADRIAN', 2, 13.44, 'Lunch', true
-);
-
- INSERT INTO cycling.cyclist_expenses (
- cyclist_name, expense_id, amount, description, paid
-) VALUES (
- 'Vera ADRIAN', 3, 25.00, 'Dinner', true
-);
- APPLY BATCH;