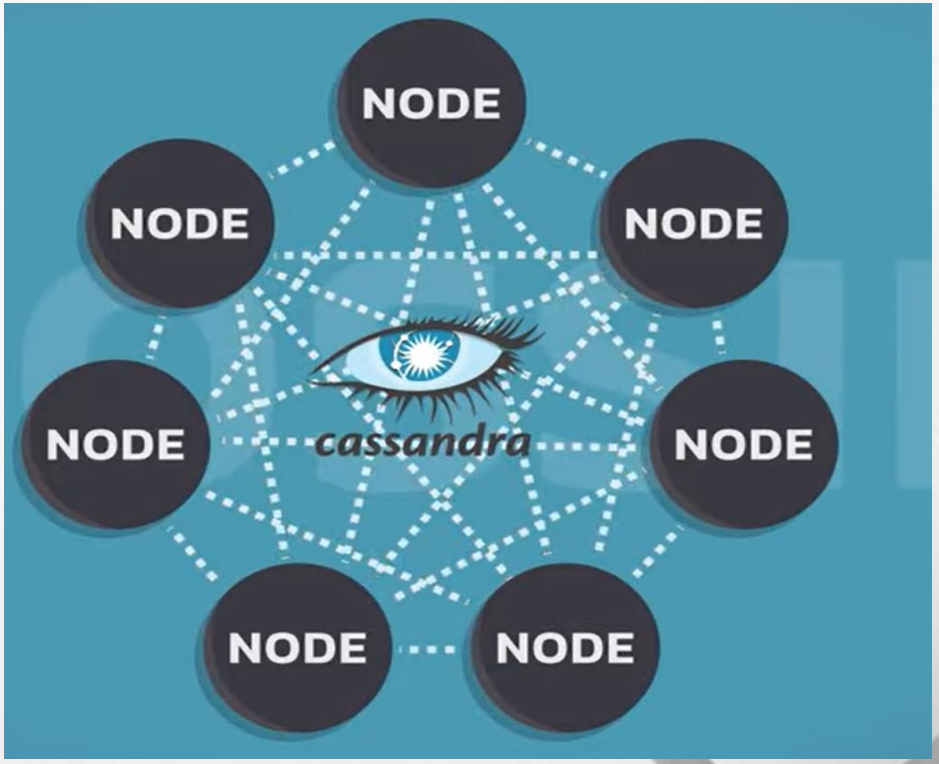
NOSQL Databases
Cassandra



cassandra

High performance. Delivered.

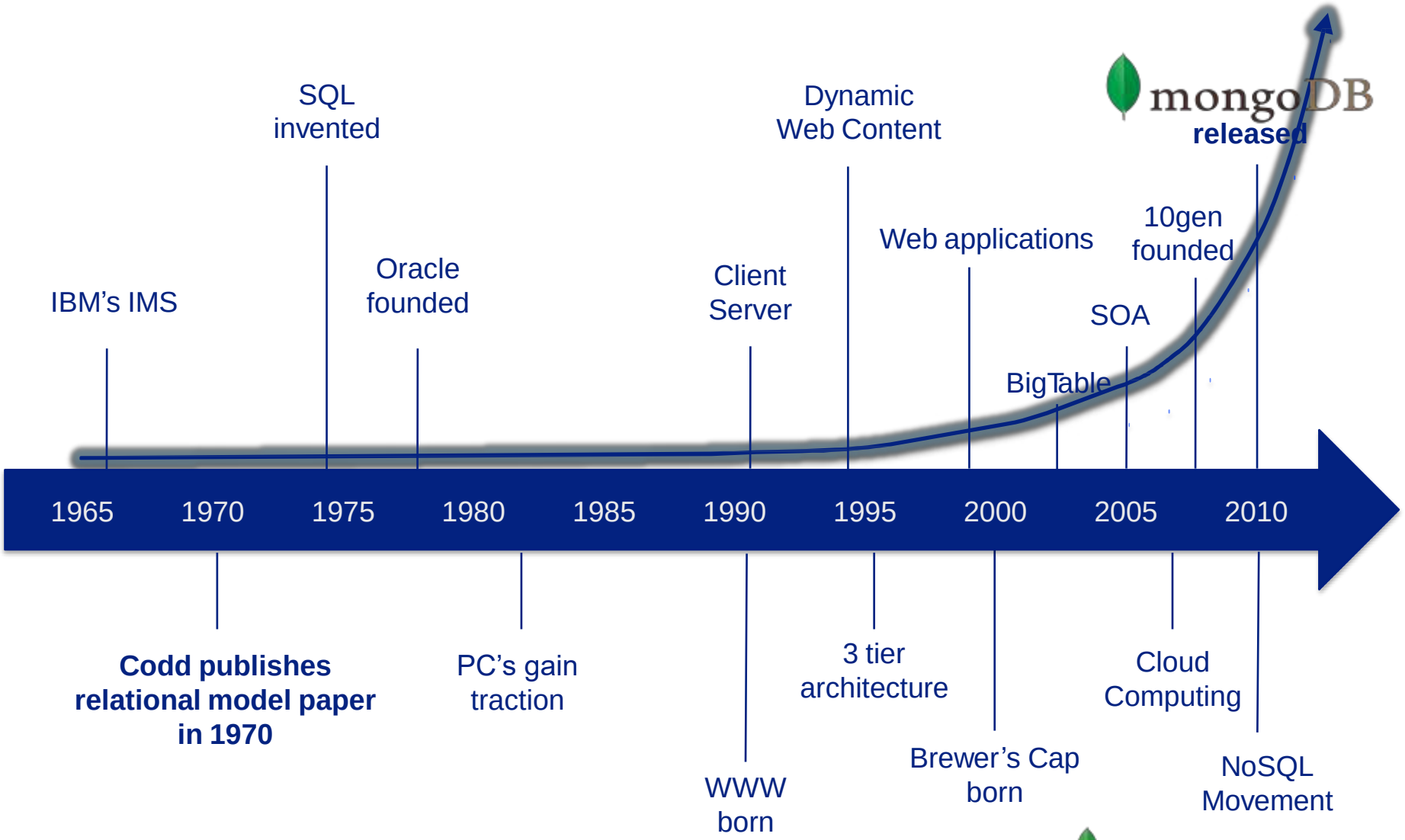
Cassandra Goals

- 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



4

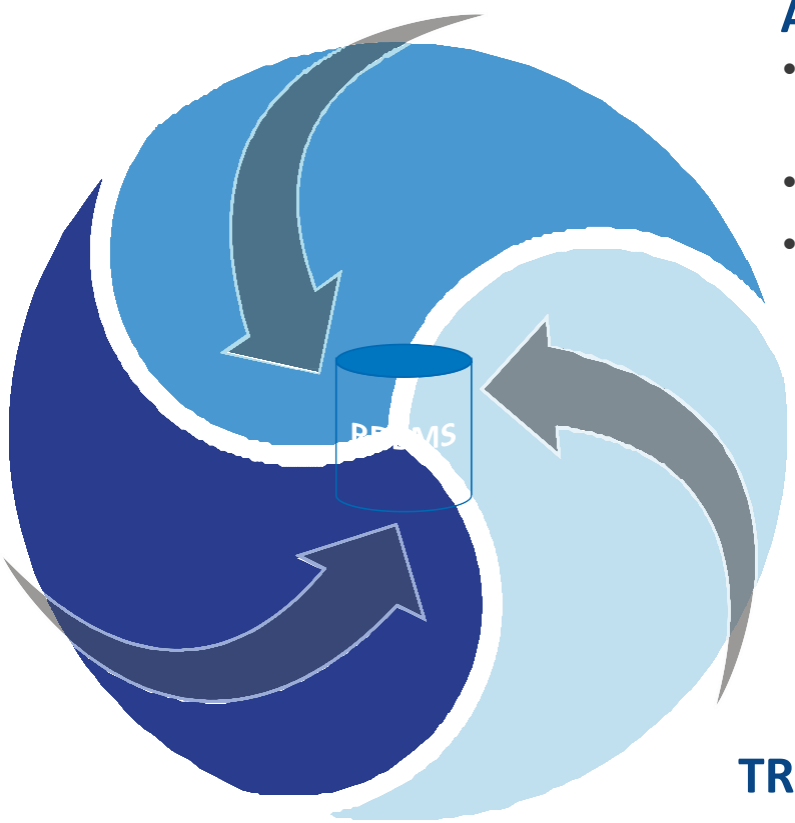
Relational Database Strengths

- 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



VOLUME & NEW ARCHITECTURES

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

DATA VARIETY & VOLATILITY

•Extremely difficult to find a single fixed schema

TRANSACTIONAL MODEL

- N x 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

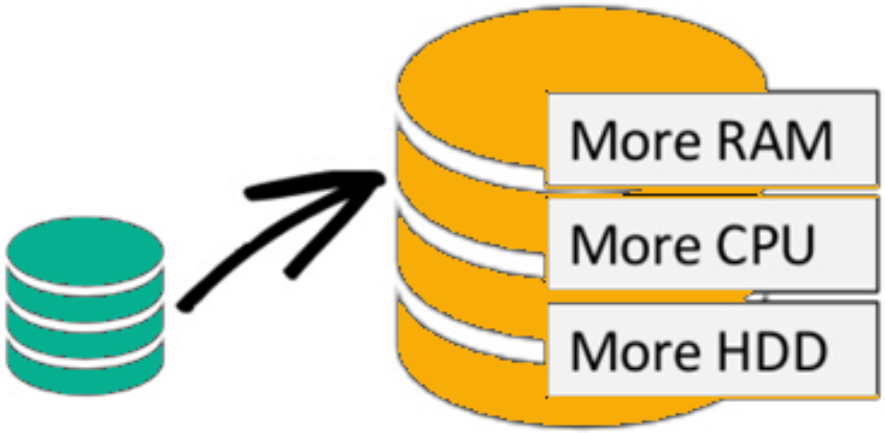
- NoSQL Database is a non-relational Data Management System, that does not require a fixed schema.
- It avoids joins, and is easy to scale.
- The major purpose of using a NoSQL database is for distributed data stores with humongous data storage needs.
- NoSQL is used for Big data and real-time web apps.
- 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

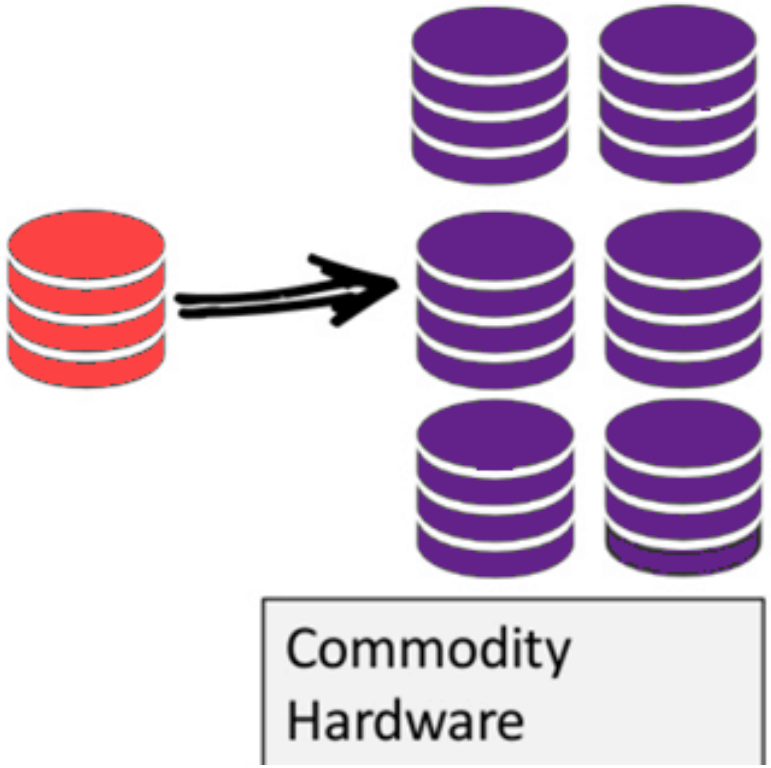
- The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data.
- The system response time becomes slow when you use RDBMS for massive volumes of data.
- To resolve this problem, we could "scale up" our systems by upgrading 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):

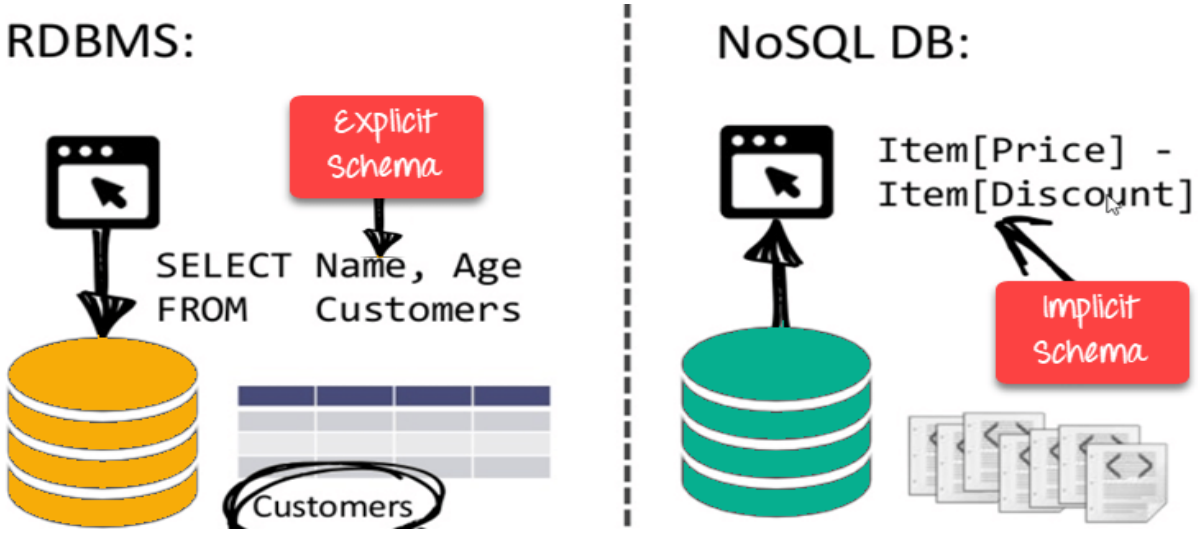


Scale-Out (*horizontal* scaling):



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



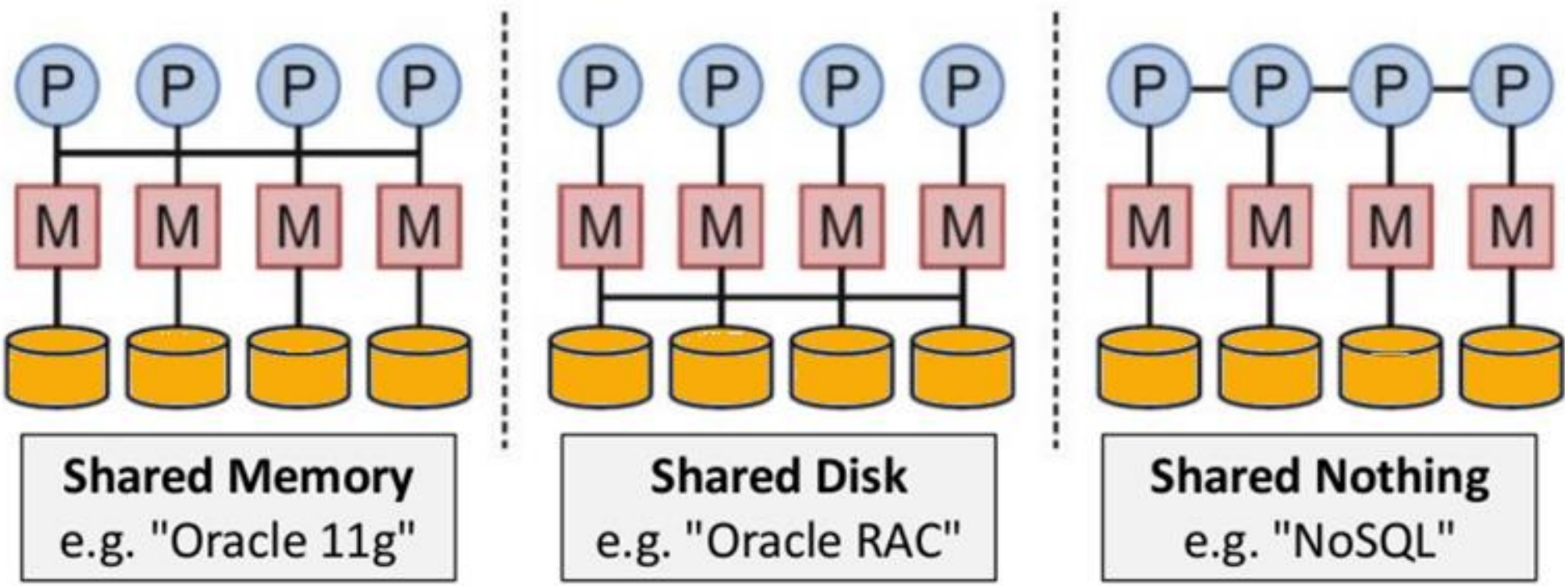
Simple API

- 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

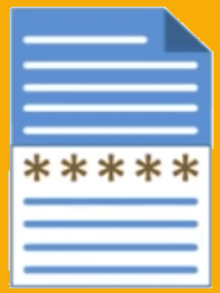
Types of NoSQL Databases

cassandra

Key Value

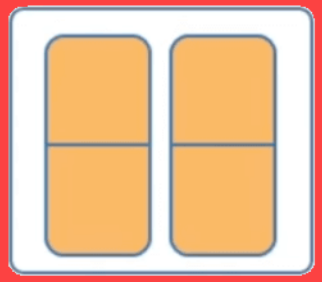
Example:
Riak, Tokyo Cabinet, Redis server, Memcached, Scalaris

Document-Based



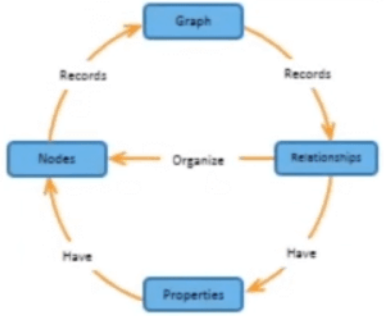
Example:
MongoDB, CouchDB, OrientDB, RavenDB

Column-Based



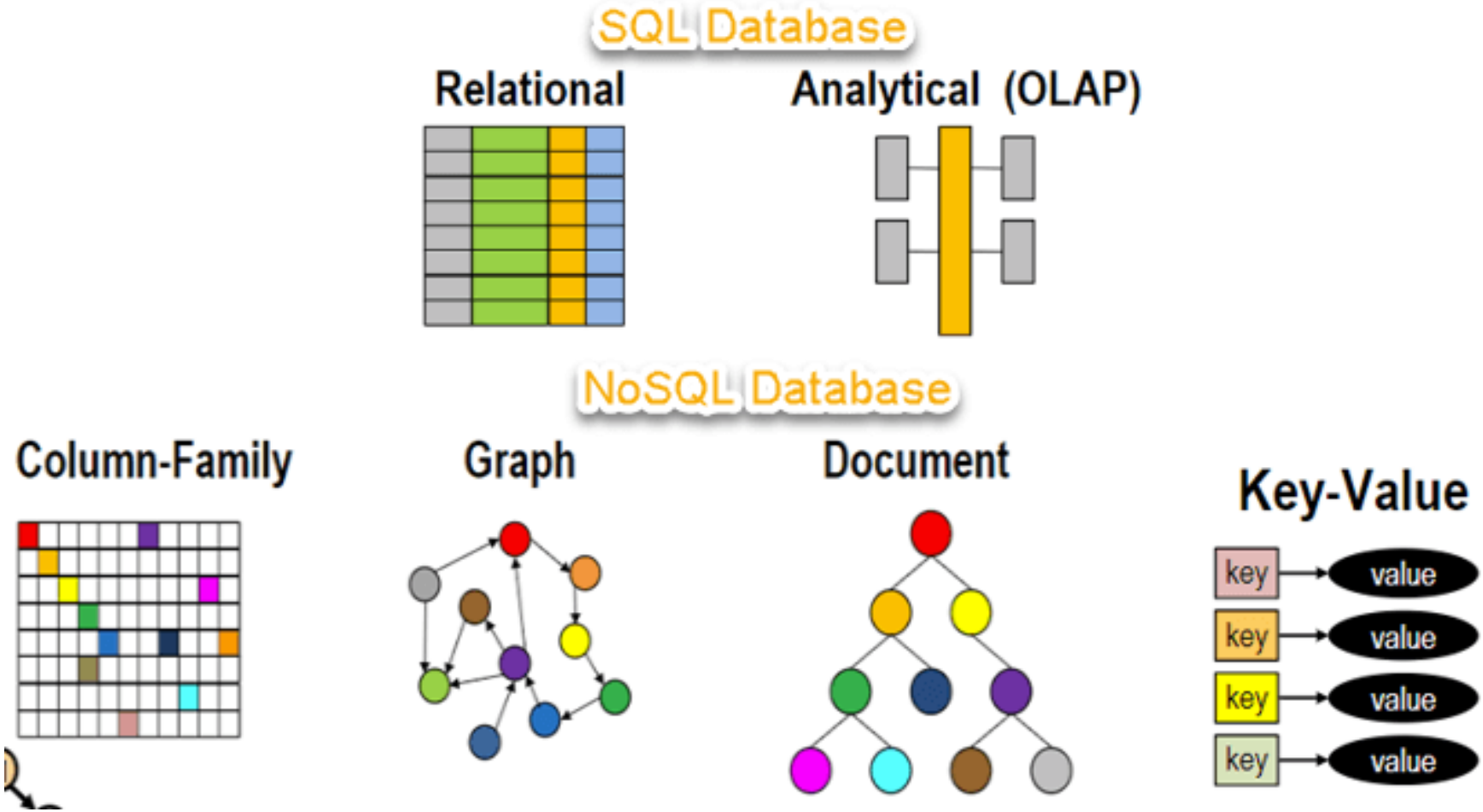
Example:
BigTable, Cassandra, Hbase, Hypertable

Graph-Based



Example:
Neo4J, InfoGrid, Infinite Graph, Flock DB

What is NoSQL



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.

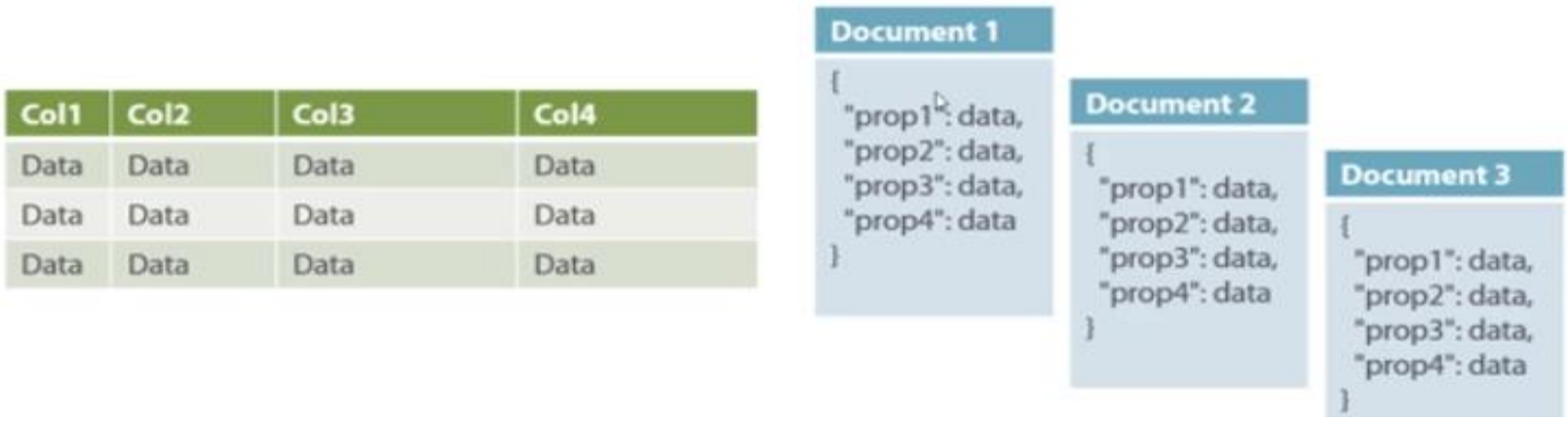
ColumnFamily			
Row Key	Column Name		
	Key	Key	Key
	Value	Value	Value
	Column Name		
	Key	Key	Key
	Value	Value	Value

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.



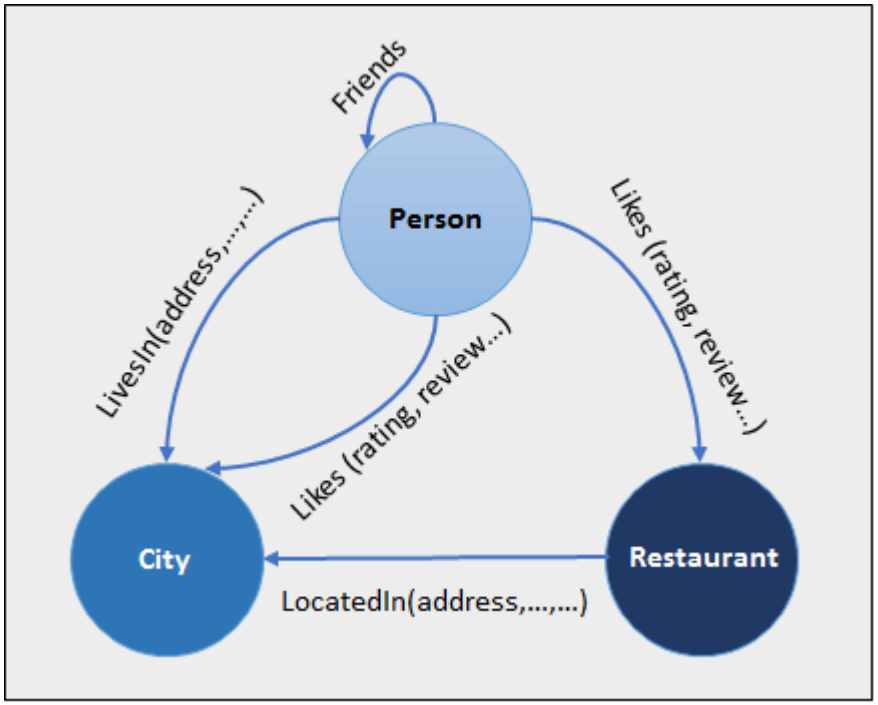
Col1	Col2	Col3	Col4
Data	Data	Data	Data
Data	Data	Data	Data
Data	Data	Data	Data

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
•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	• 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 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?

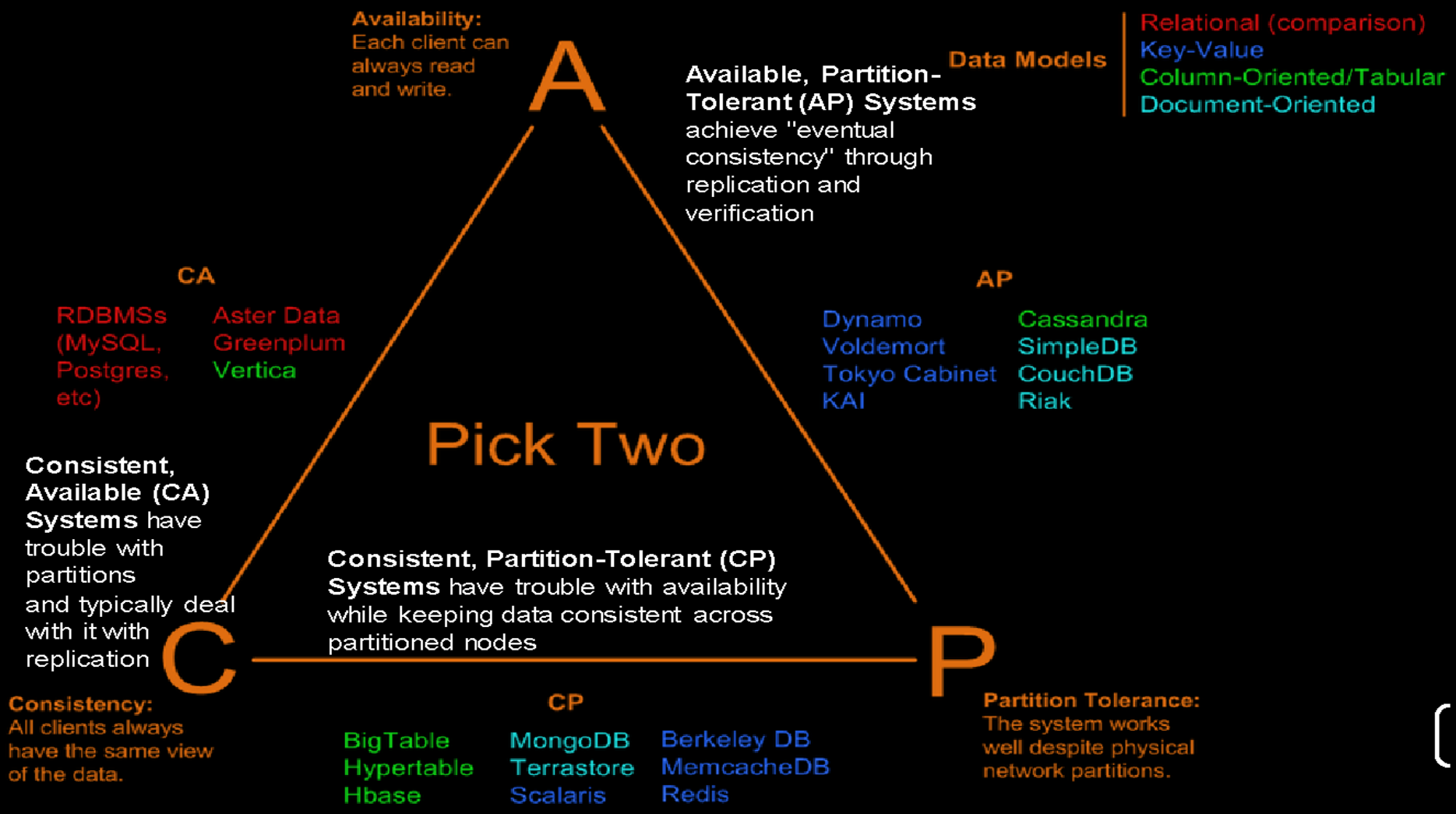
- Partition Tolerance
  - Partition Tolerance means that the system should continue to function even if the communication among the servers is not stable.
  - For example, the servers can be partitioned into multiple groups which may not communicate with each other. Here, if part of the database is unavailable,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

What is the CAP Theorem?

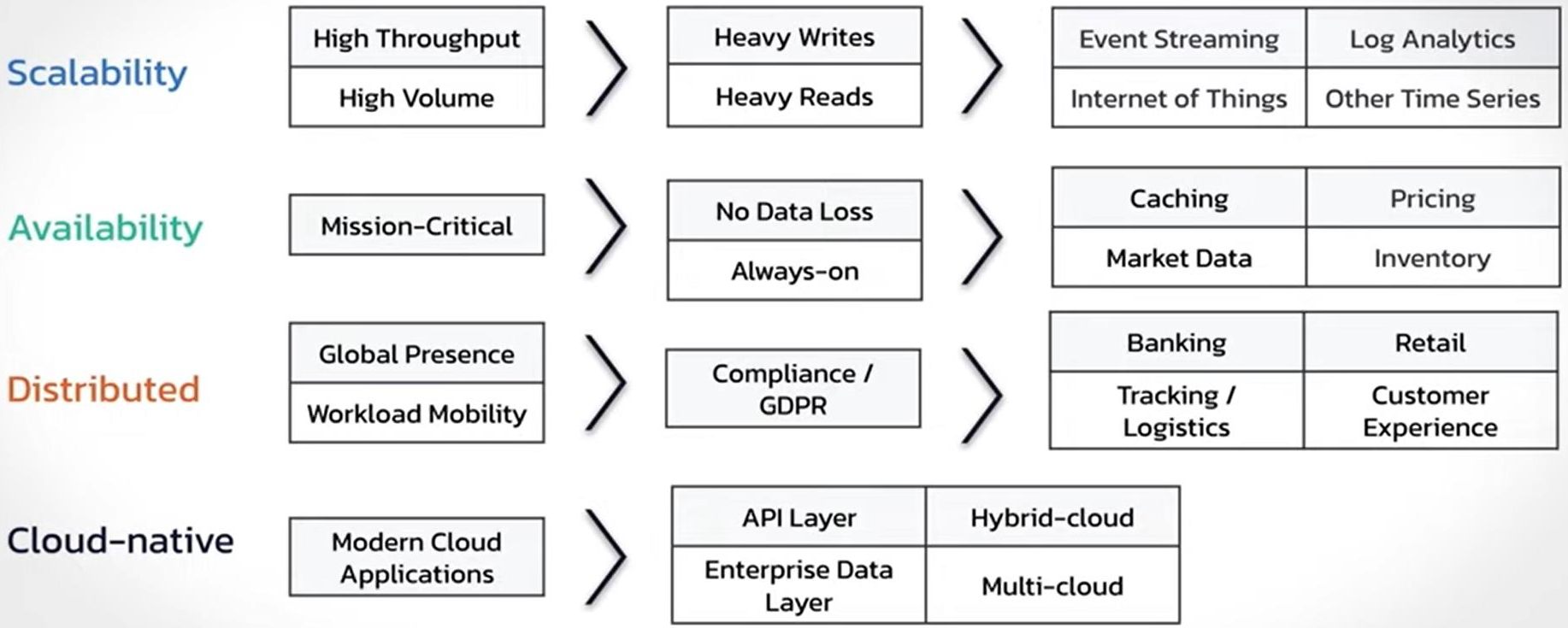
- Partition Tolerance
  - Partition Tolerance means that the system should continue to function even if the communication among the servers is not stable.
  - For example, the servers can be partitioned into multiple groups which may not communicate with each other. Here, if part of the database is unavailable,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

cassandra

Visual Guide to NoSQL Systems



Apache Cassandra



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

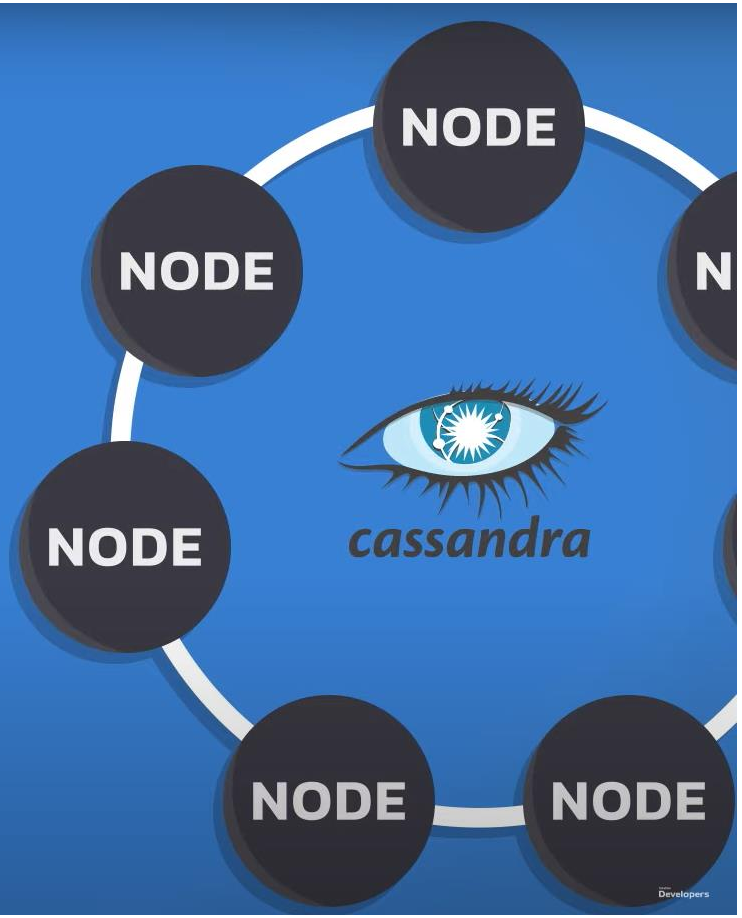
- **Consistency** essentially means that a read always returns the most recently written value.
- Consider two customers are attempting to put the same item into their shopping carts on an ecommerce site.
- If I place the last item in stock into my cart an instant after you do, you should get the item 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

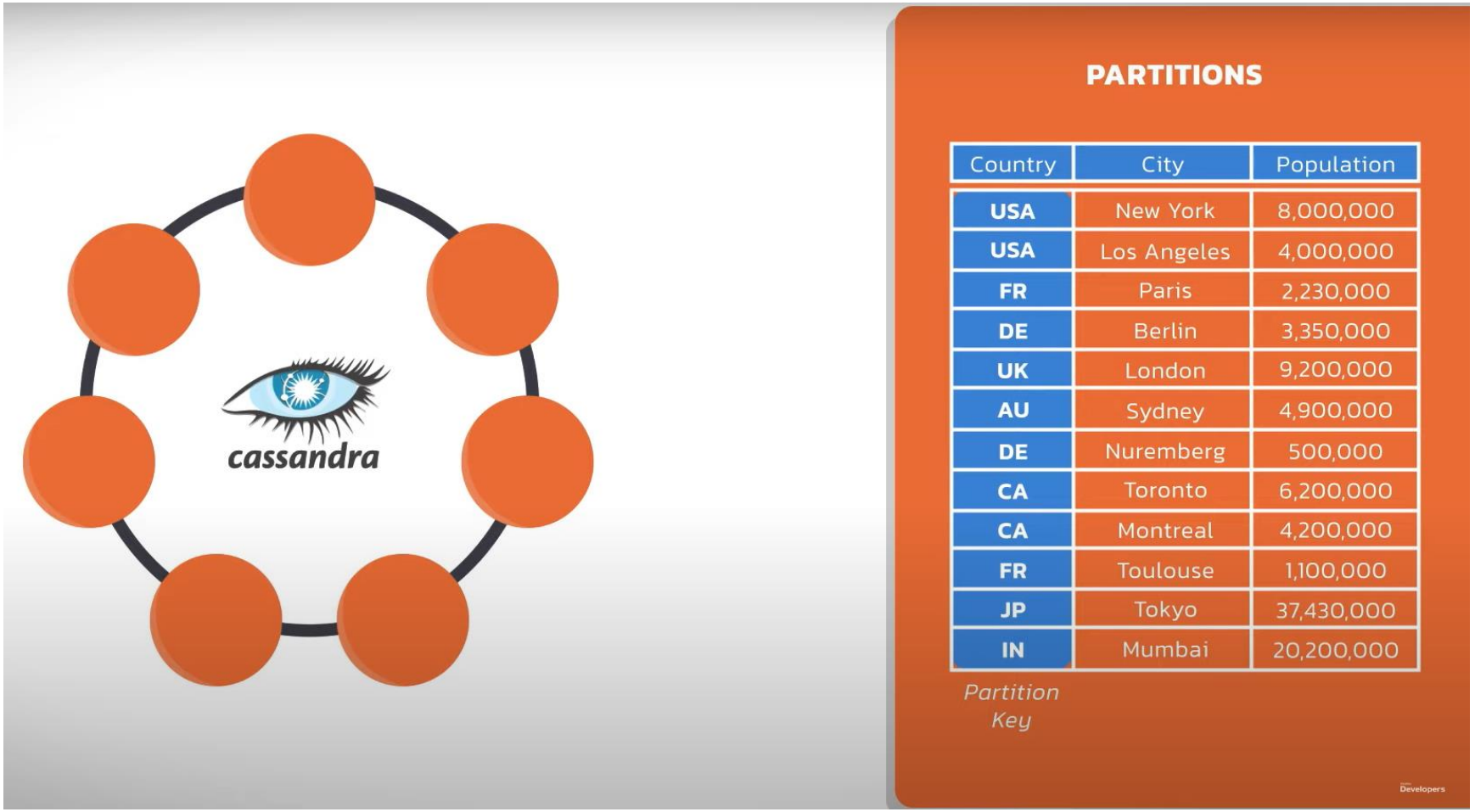
- Cassandra is more accurately termed “tuneably consistent,” which means it allows you to easily decide the level of consistency you require, in balance with the level of availability.

Why Apache Cassandra

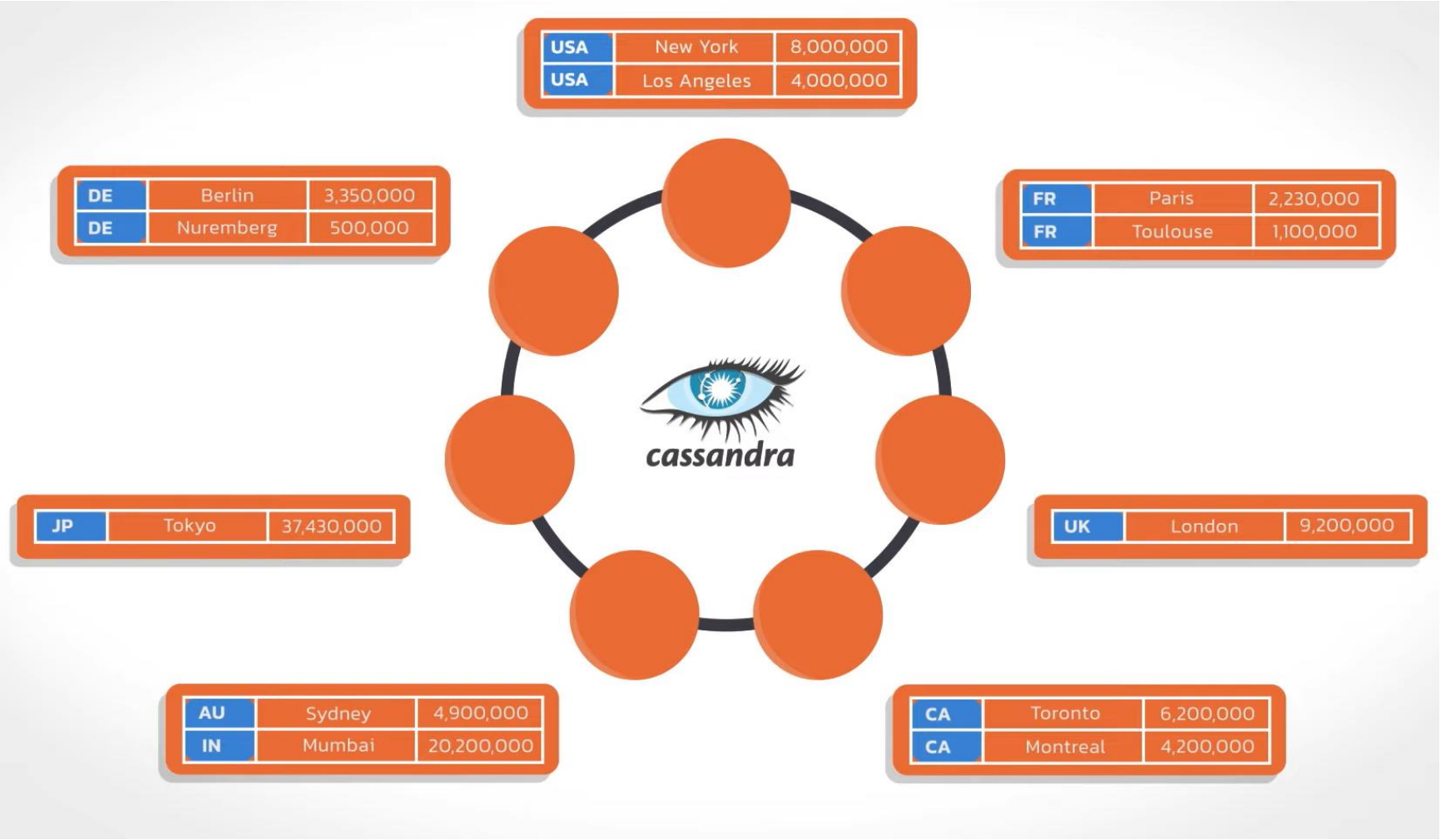
- Petabyte Database
- High Availability
- Geographic Distribution
- Performance
- Vendor Independant



Why Apache Cassandra



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

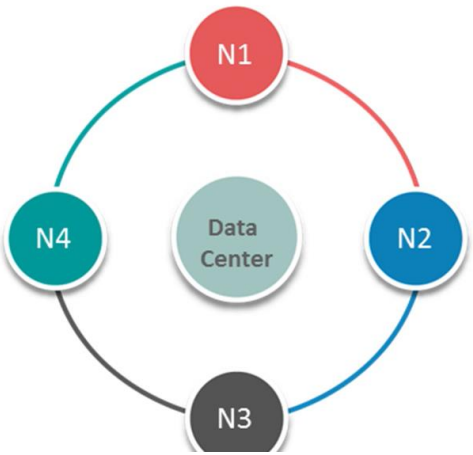
- Mahalo uses it for its primary near-time data store.
- Facebook still uses it for inbox search, though they are using a proprietary fork.
- Digg uses it for its primary near-time data store.
- Rackspace uses it for its cloud service, monitoring, and logging.
- Reddit uses it as a persistent cache.
- 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.
- Onespot uses it for a subset of its main data store.

Column Based Database Cassandra

Version	Original Release Date	Latest Version	Release Date	Status
0.6	2010-04-12	0.6.13	2011-04-18	No longer supported.
0.7	2011-01-10	0.7.10	2011-10-31	No longer supported.
0.8	2011-06-03	0.8.10	2012-02-13	No longer supported.
1.0	2011-10-18	1.0.12	2012-10-04	No longer supported.
1.1	2012-04-24	1.1.12	2013-05-27	No longer supported.
1.2	2013-01-02	1.2.19	2014-09-18	No longer supported.
2.0	2013-09-03	2.0.17	2015-09-21	No longer supported.
2.1	2014-09-16	2.1.17	2017-02-21	Still supported.
2.2	2015-07-20	2.2.9	2017-02-21	Still supported.
3.0	2015-11-09	3.0.11	2017-02-21	Still supported.
3.10	2017-02-03	3.10	2017-02-03	Latest release.
3.11	2017-02-22	3.11	2017-02-11	Github 3.11 branch.

Cassandra Architecture

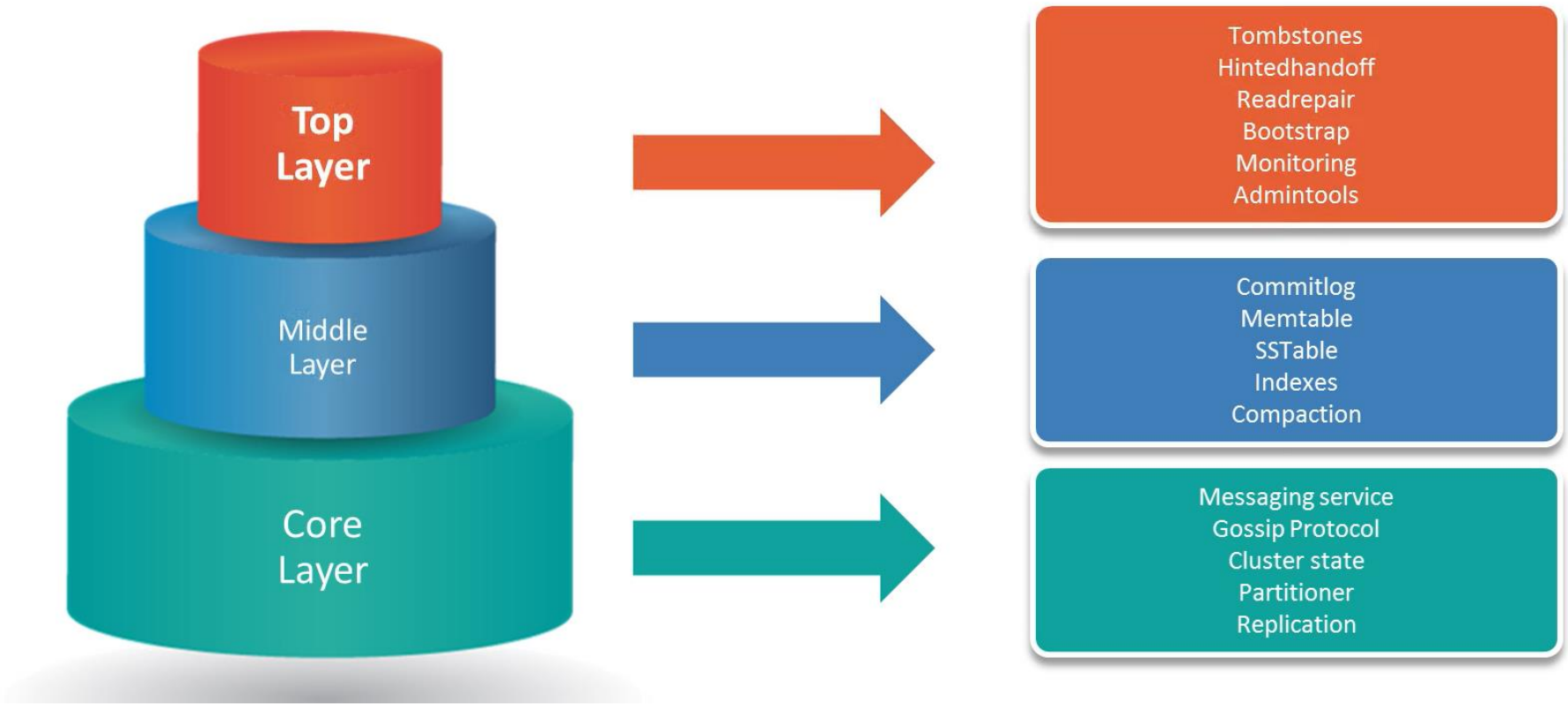
- The architecture of Cassandra contributes to a database that *scales* and *performs with continuous availability*
- It has a *masterless "ring"* distributed architecture that is elegant, and easy to set up and maintain
- Cassandra's built-for-scale architecture is *capable of handling large amounts 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.

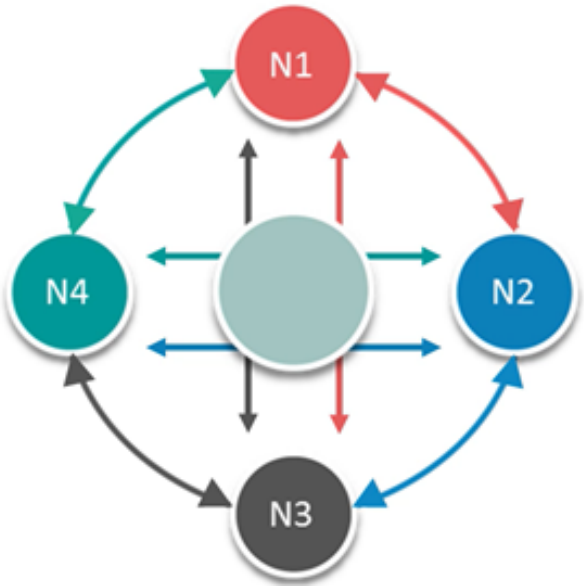
Cassandra Architecture



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



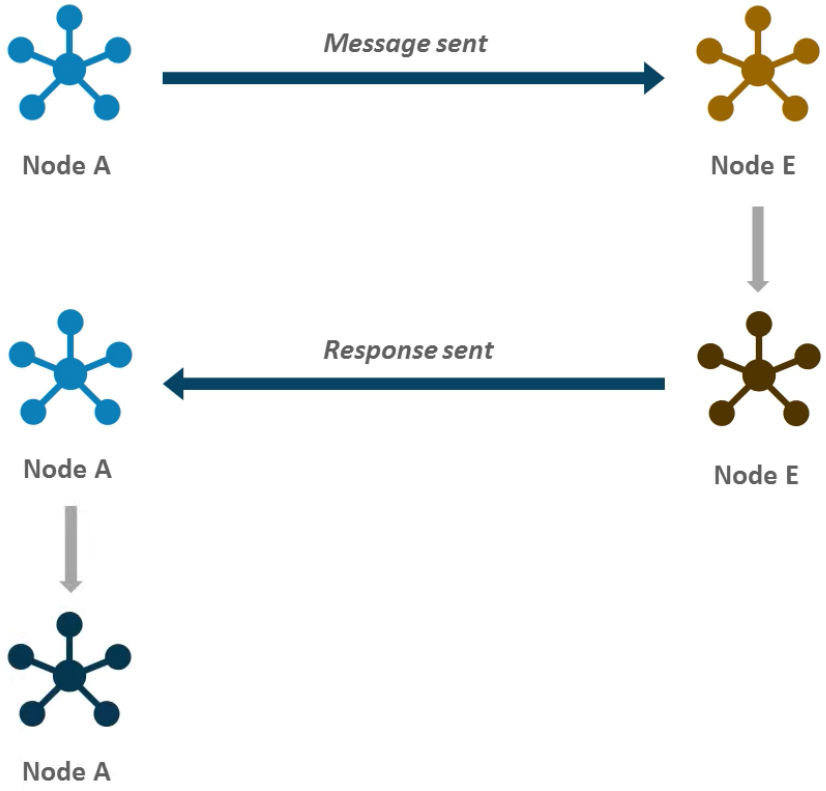
Gossip Protocol Operation

Consider the following Nodes in a Cluster:

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

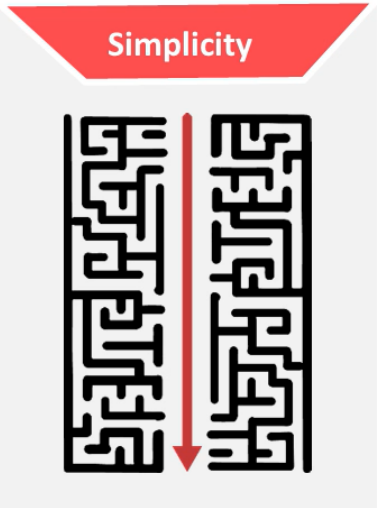
Gossip Protocol Operation

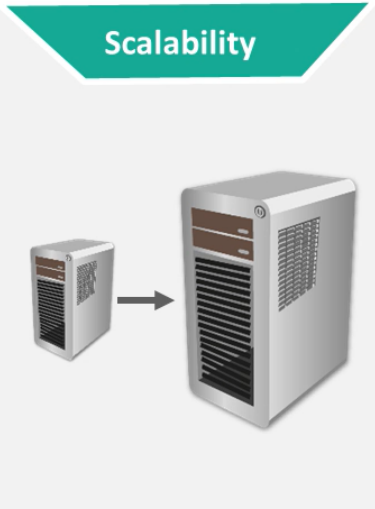
1. *Node A* randomly selects a *Node E* from a list of nodes known to it
2. *A* sends a *message* to *E* containing the data from *A*
3. *E* updates its data set with the received information
4. *E* sends back a *response* to *A* containing its data
5. *A* updates its data set with the received information



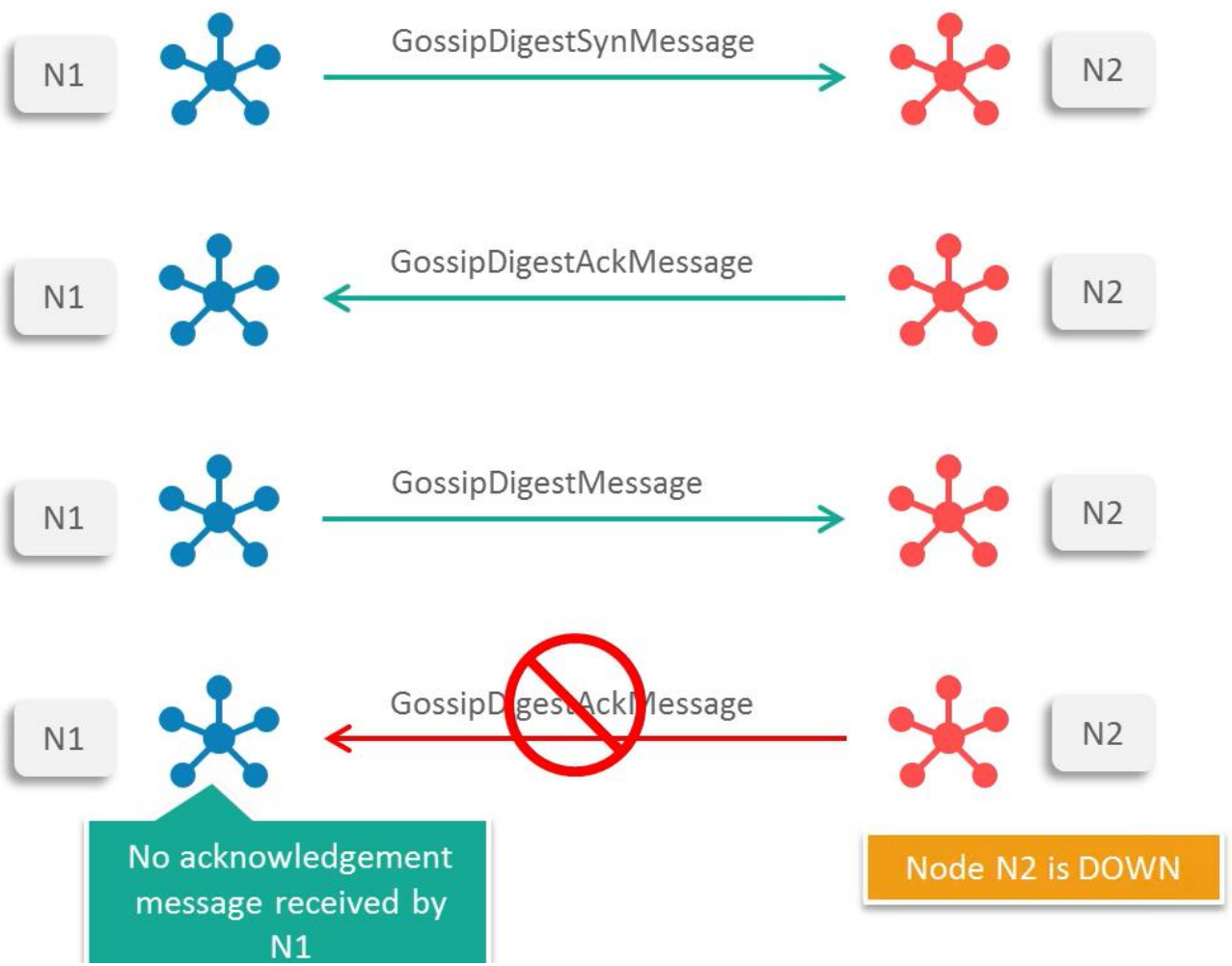
Gossip Protocol Features

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





Gossip Protocol Failure Detection



Cassandra Distributed Database

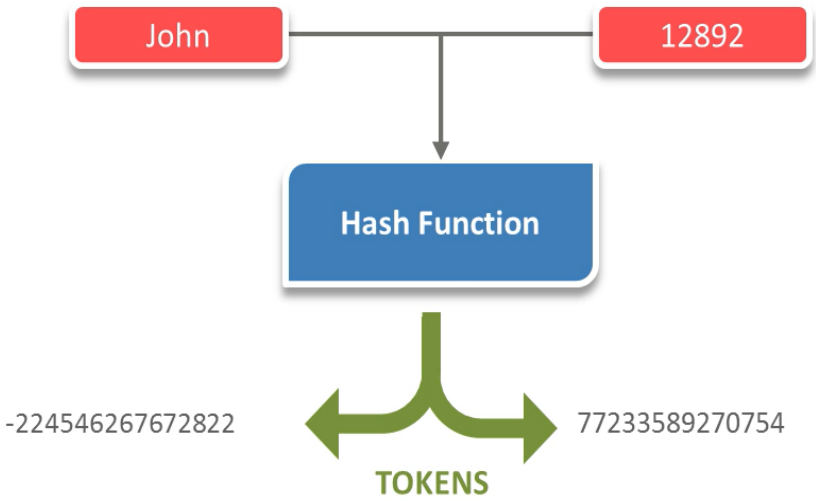
Cassandra distributes the data across the entire cluster

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

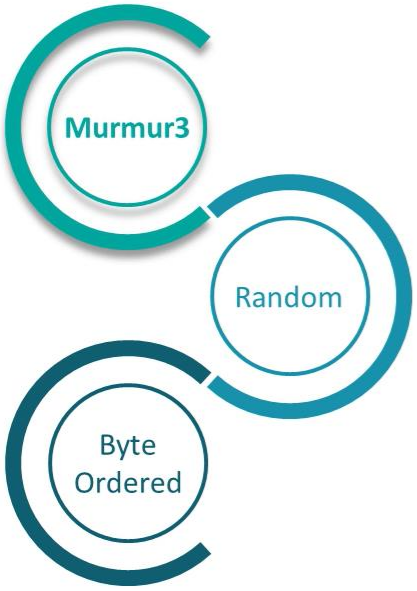
Partition: It is *a hash function located on each node which hashes tokens* from designated values in rows being added

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



Types of Partitioners



Murmur3Partitioner is the *default partitioner*

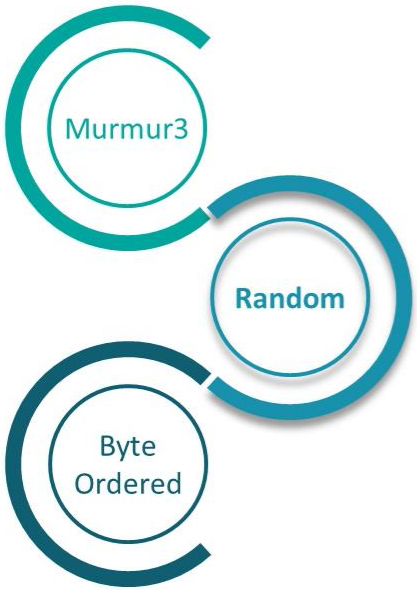
Improved and faster than RandomPartitioner

Uniformly distributes data based on *MurmurHash* function

64 bit hash value partition key

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

Types of Partitioners



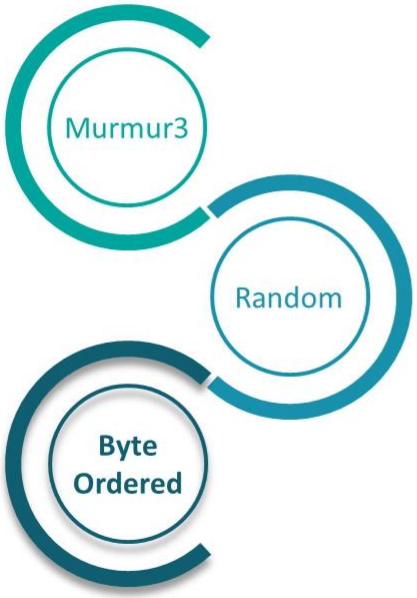
RandomPartitioner was the default partitioner *prior to Cassandra 1.2*

It is used with vnodes

Uniformly Distribution : MD5 hash values

Range: 0 to 2^{127}-1

Types of Partitioners



ByteOrderedPartioner 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
```

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

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:

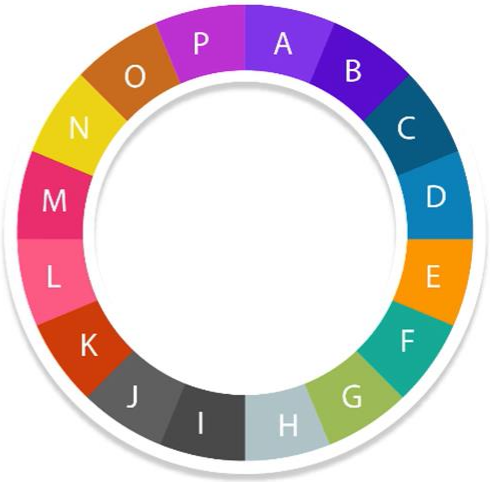
1. While configuring a cluster, we need to *calculate and assign tokens* to each *node manually*

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*

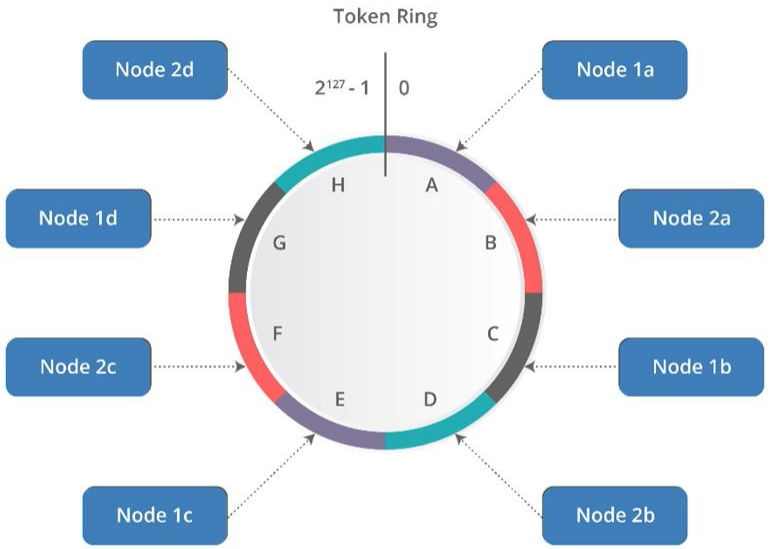
Solution Virtual Nodes

- 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



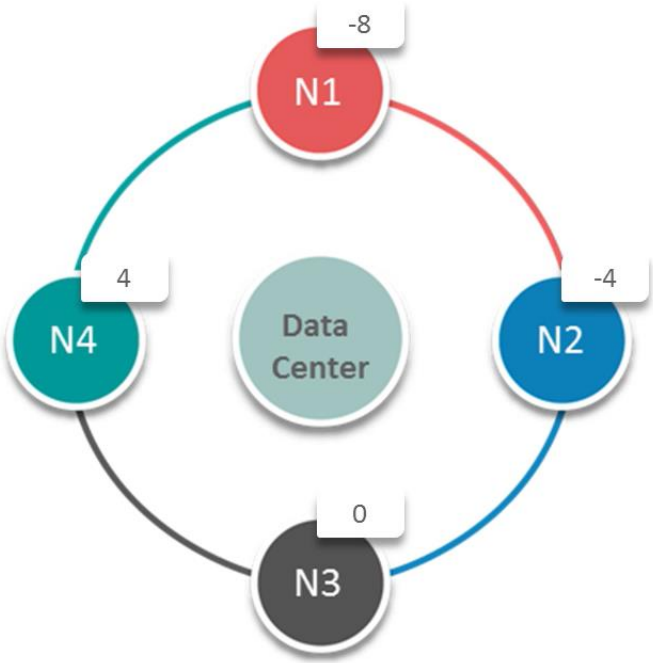
Tokens and Virtual Nodes

- *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

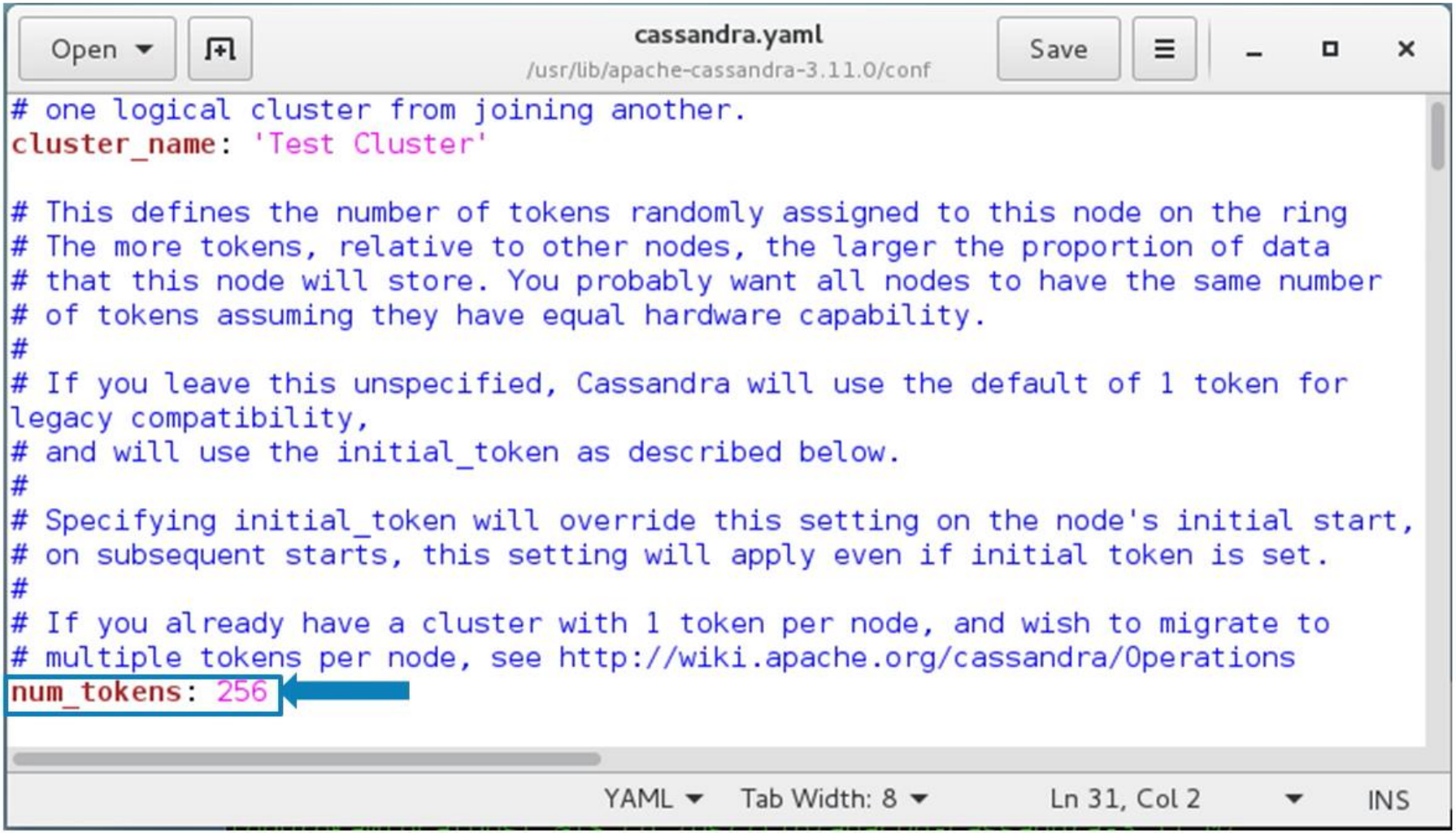


Virtual Nodes Token Allocation

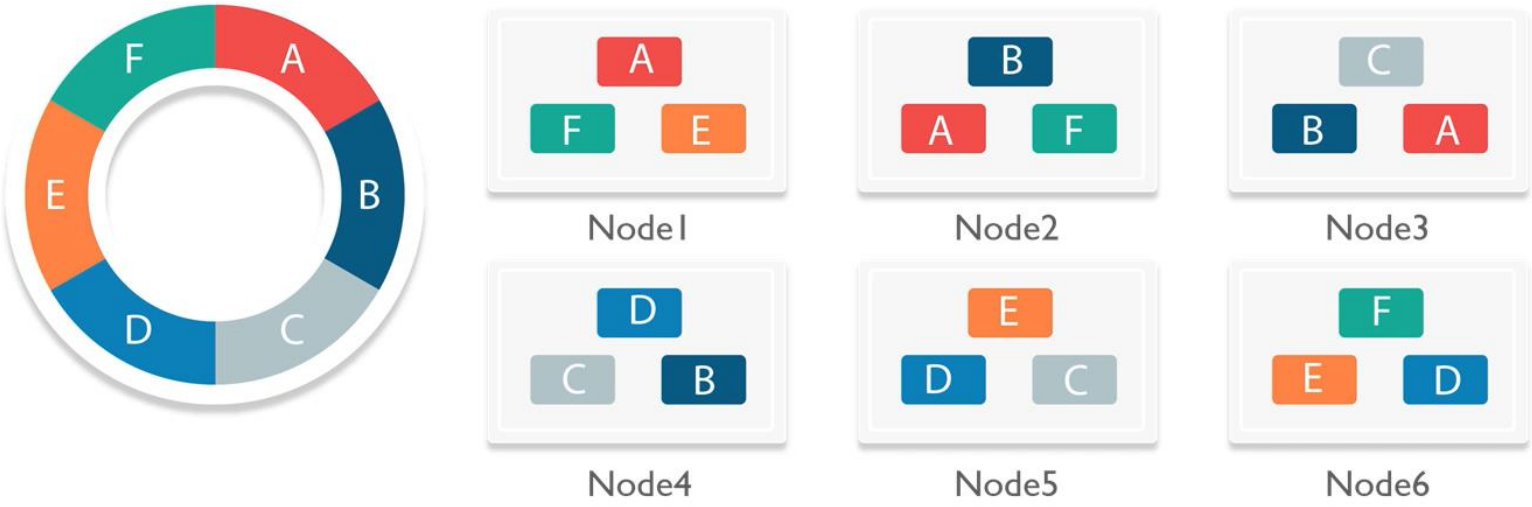
- The token range from -8 to 7 is distributed among four nodes as shown in the figure
- Here, node 1 has tokens with values greater than or equal to -8 and less than -4
- Node 2 has tokens for a range greater than or equal to -4 and less than 0
- Node 3 has tokens with value greater than or equal to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



Cassandra.yaml File

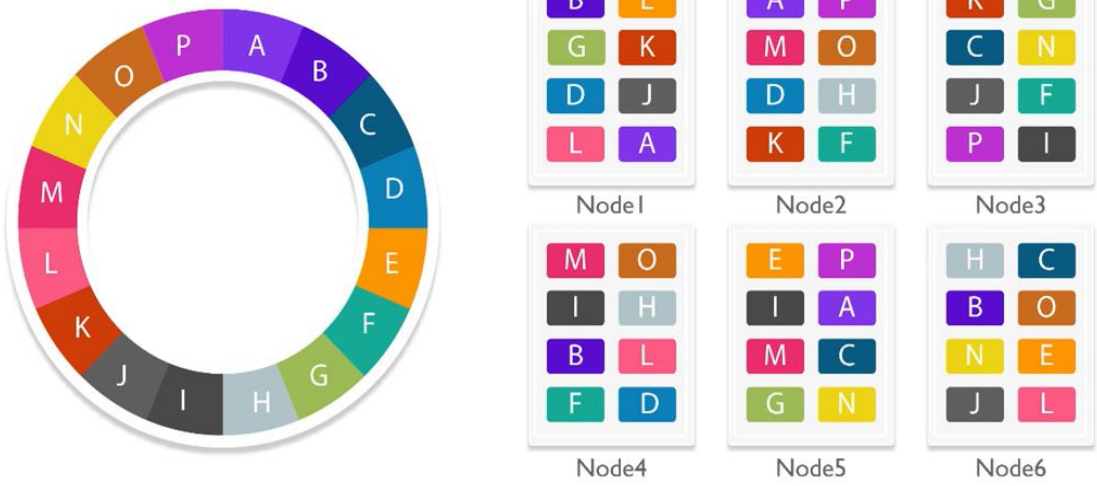


Ring Without VNodes



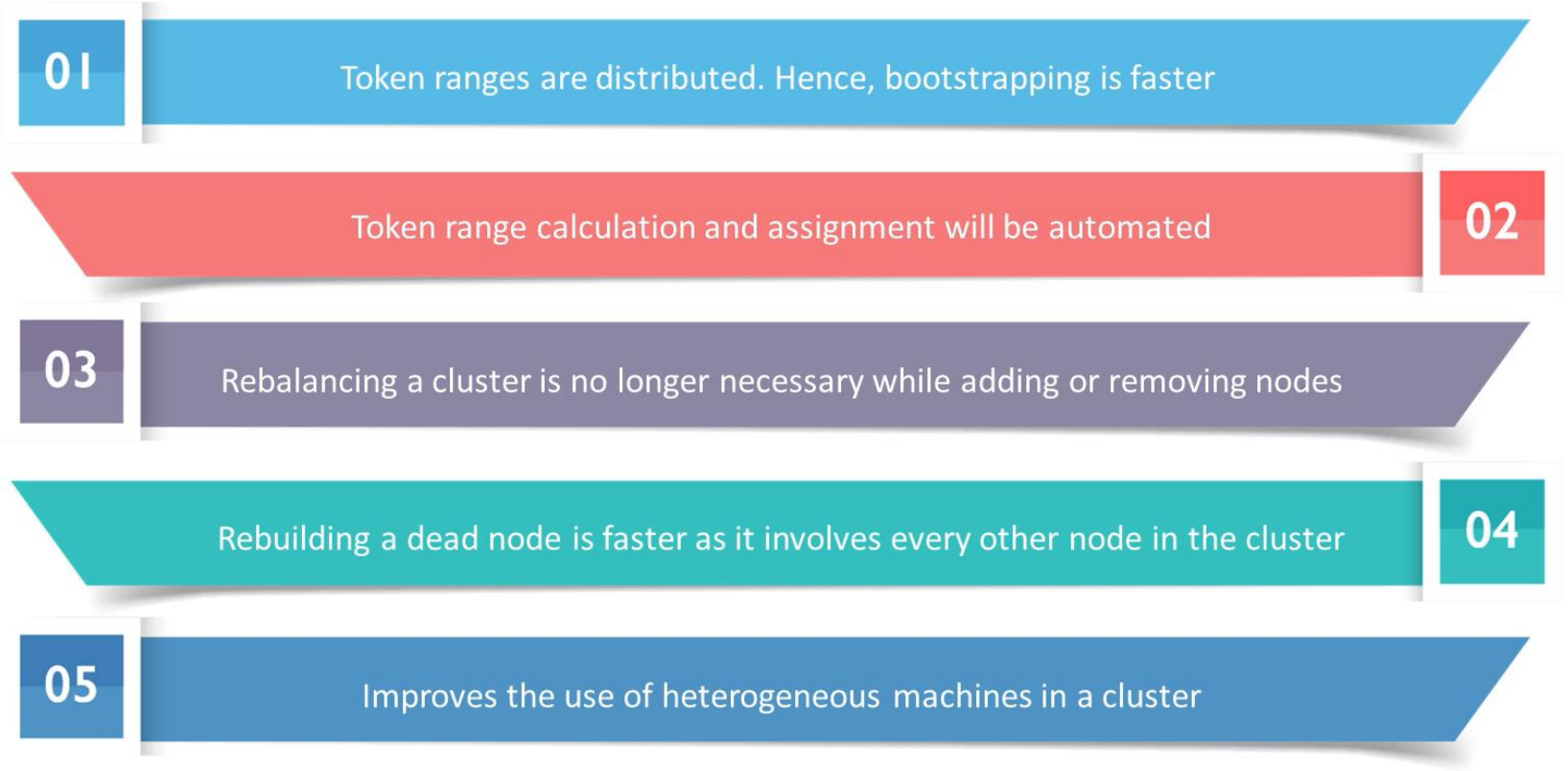
- Notice that a node owns exactly one contiguous partition range in the ring space
- Each node stores data determined by mapping the partition key to their token value
- The token value lies within a range from the previous node to its assigned value
- Each node also contains copies of each row from other nodes in the cluster

Ring Without VNodes



- Within a cluster, virtual nodes are randomly selected and non-contiguous
- The placement of a row is determined by the hash of the partition key within many smaller partition ranges belonging to each node

How are Virtual Nodes Helpful

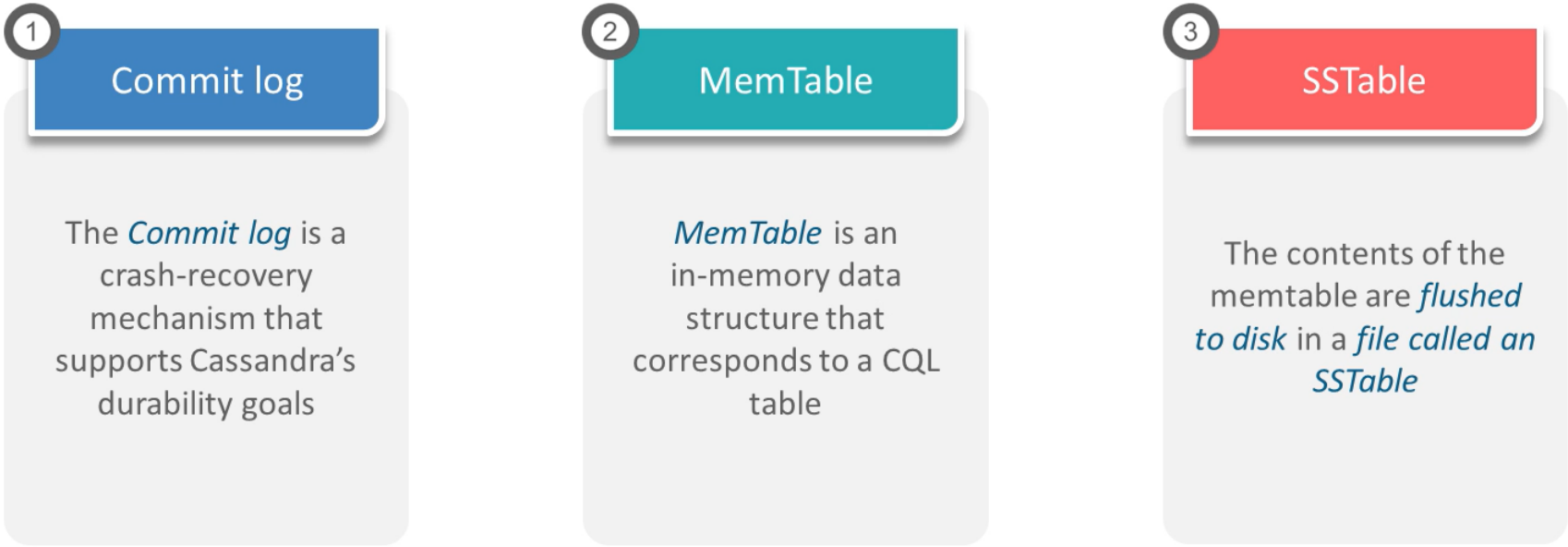


Write Consistency Levels

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



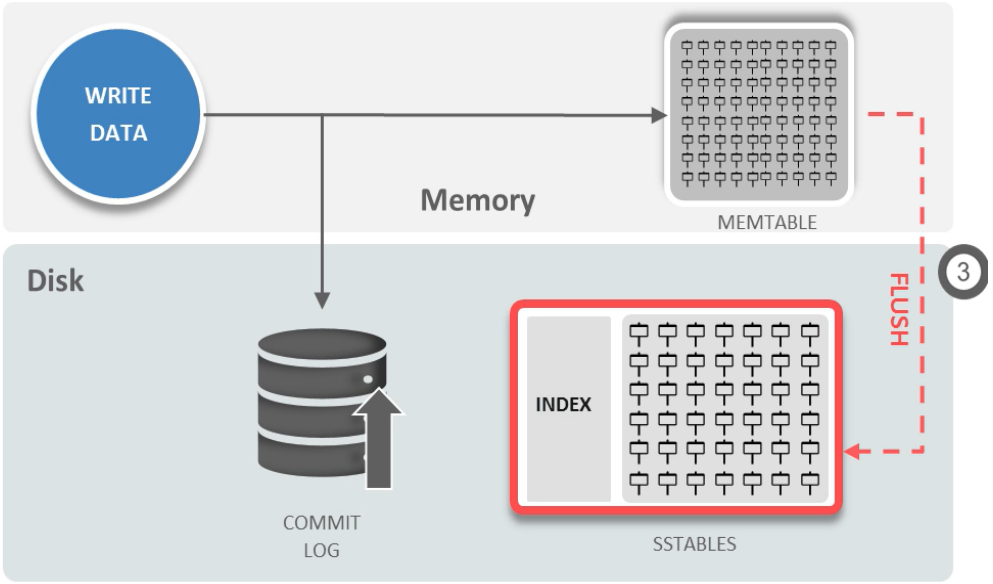
Write Path Process

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

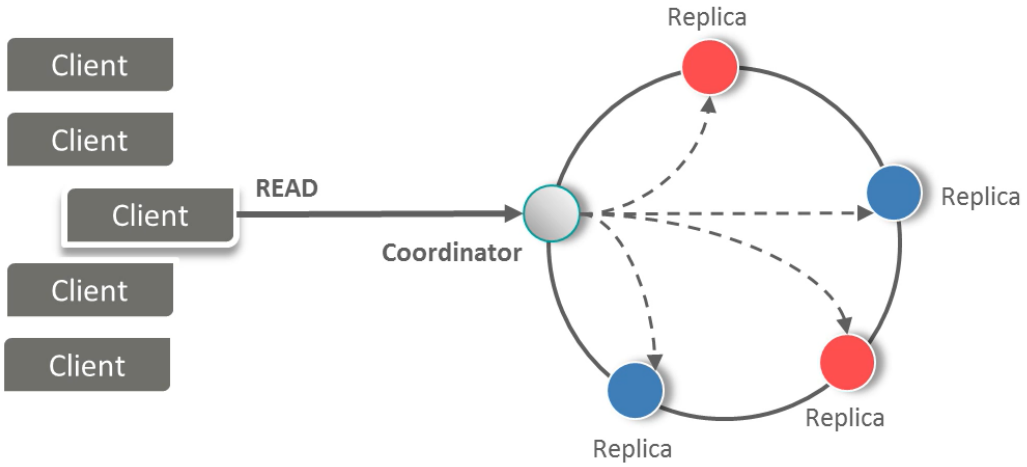
Cassandra periodically consolidates the *SSTables*, discarding unnecessary data



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



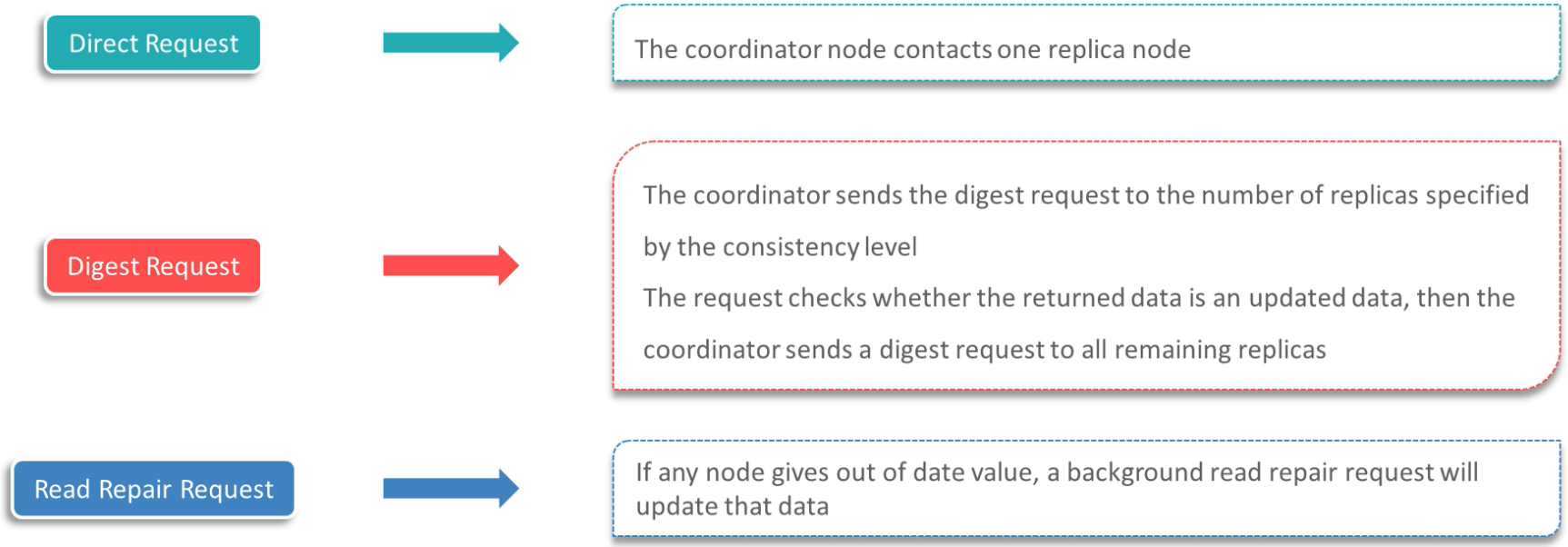
Read Consistency Level

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.

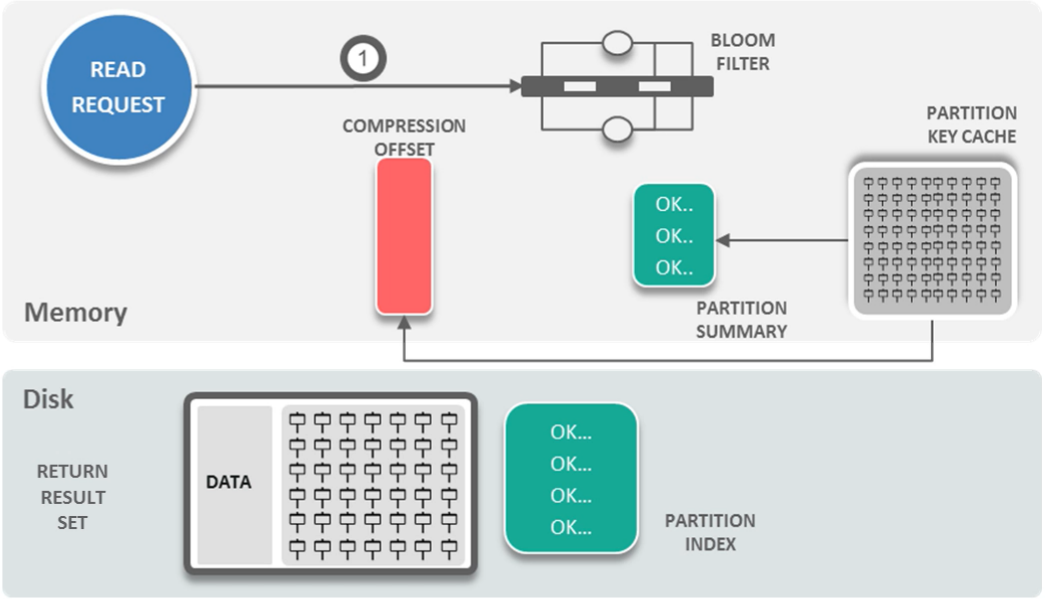
Read Consistency Level

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

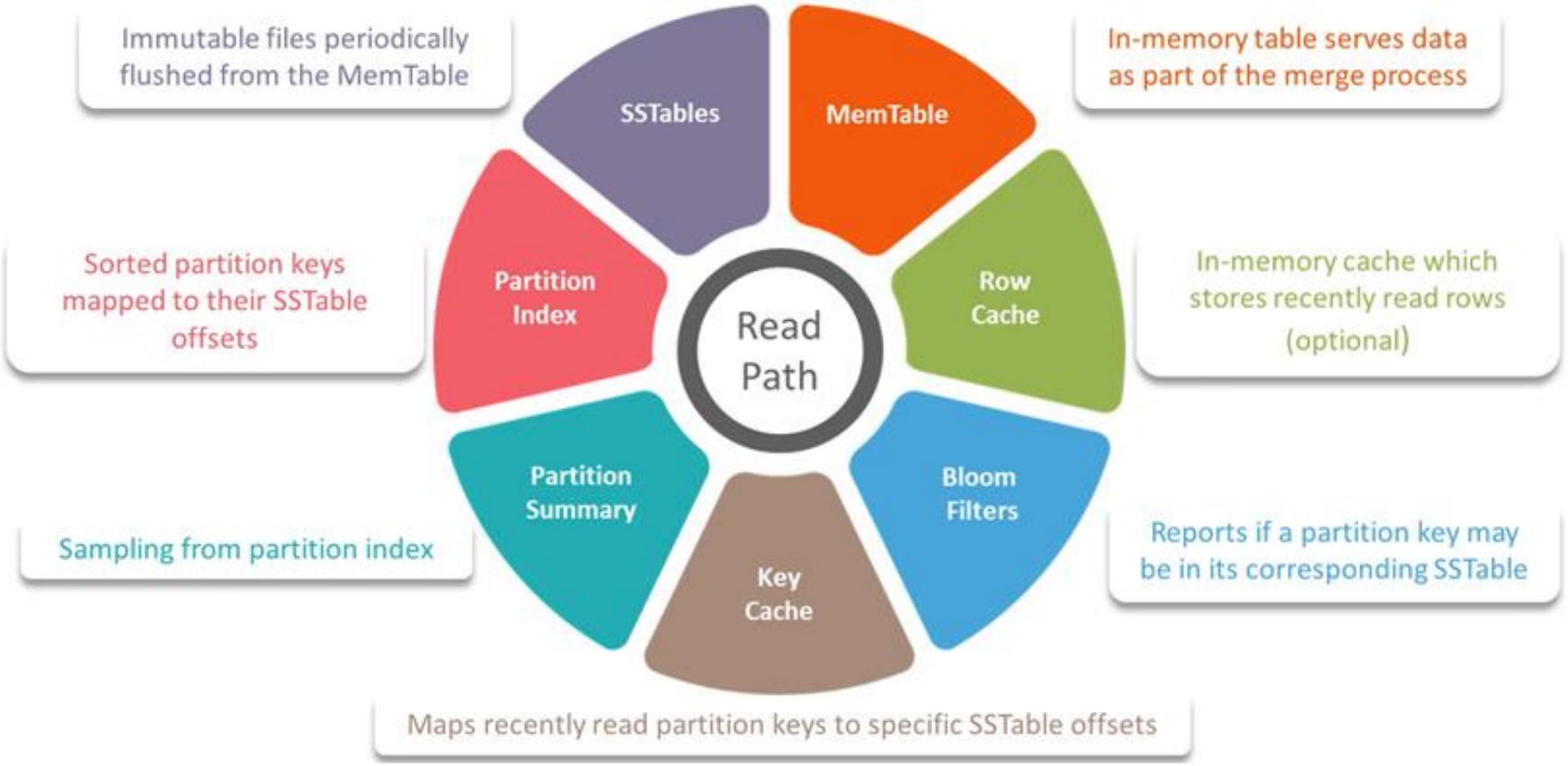


Cassandra - Read

- 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

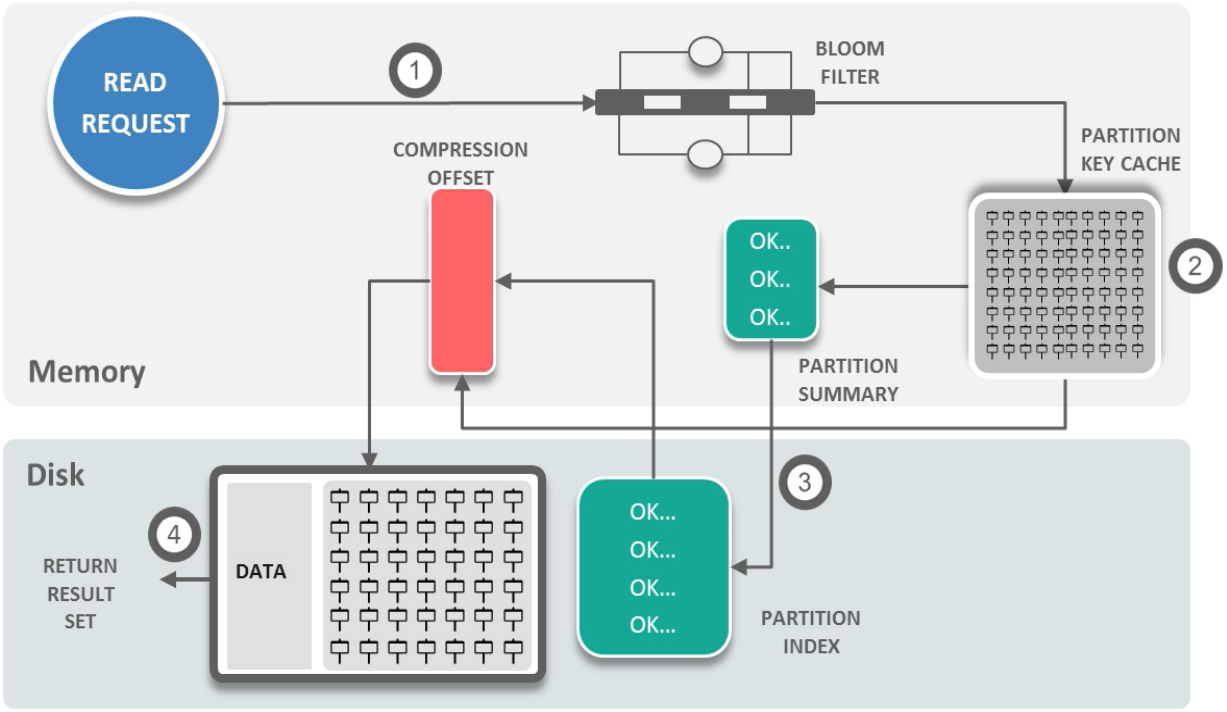


Key Elements – Read Path



Read Path - Process

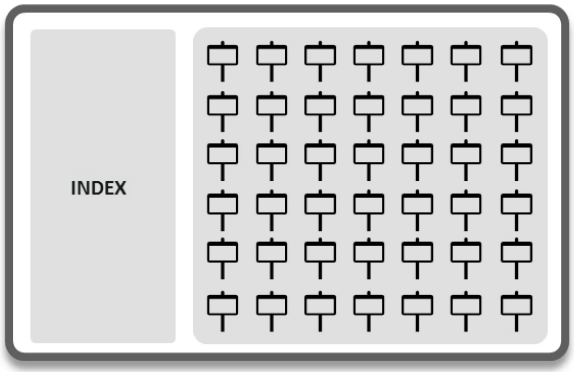
1. Cassandra consults a bloom filter that checks the probability of a table having the needed data.
2. If the probability is good, Cassandra checks a memory cache that contains row keys and finds the needed key in the cache
3. Locates the needed key in Partition Index and finds corresponding data on 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 exe
cute this query as it might involve data filtering and thus may have unpredictab
le performance. If you want to execute this query despite the performance unpred
ictability, 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
```

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

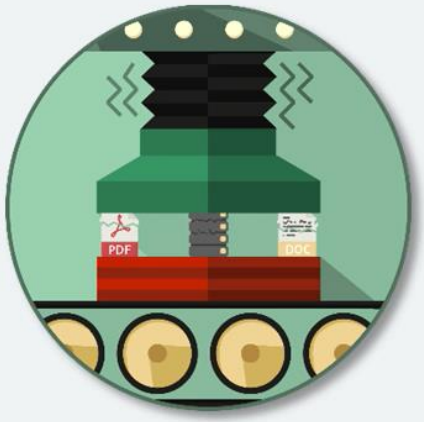
- On high-cardinality columns
- In tables that use a counter column
- On a frequently updated or deleted column
- To look for a row in a large partition unless narrowly queried

Compaction

It is the process of *freeing up space* by *merging large accumulated datafiles*

On compaction:

- the merged data is sorted,
- a new index is created over the sorted data,
- and the freshly merged, sorted, and indexed data is written to a single new SSTable



It *improves performance* by *reducing* the number of required *seeks*

Compaction Types

There are *different types of compaction* in Cassandra:

Major Compaction

1

It is triggered in two ways:

- *Via a node probe*
- *Automatically*

A node probe sends a TreeRequest message to the nodes that neighbour the target.

Read-Only Compaction

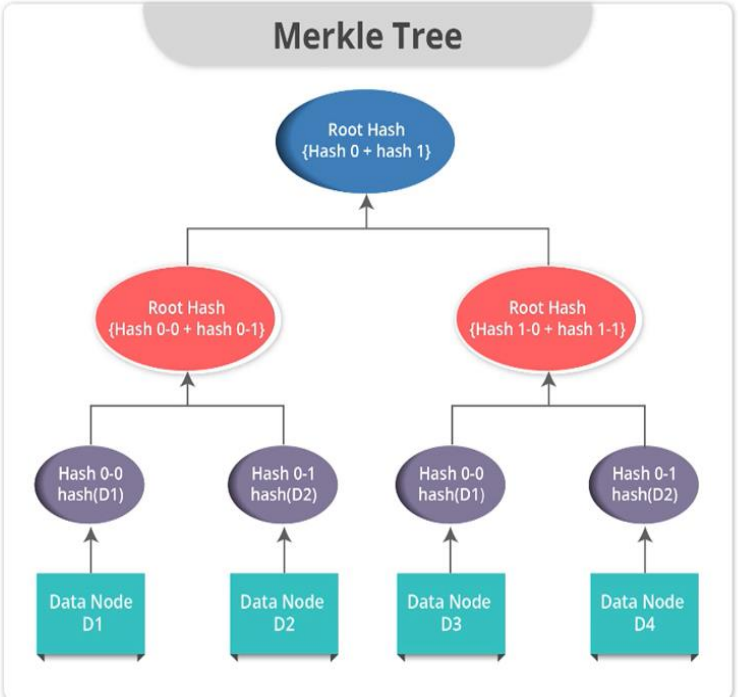
2

- Get the *key distribution* from the column family
- The column family needs to be validated, it will create *Merkle tree created in anti-entropy process* & broadcast it to the neighbouring nodes.
- The Merkle trees are brought together in a "rendezvous" as a list of differencers
- The comparison is executed by the StageManager class

Anti-Entropy

Anti-entropy is the *replica synchronization mechanism*, ensuring that data on different nodes is updated to the newest version

- *Cassandra* uses *Merkle tree* for *anti-entropy repair*
- *Merkel Tree* is a hash tree where leaves are hashes of the values of individual keys
- Each *column family* has its *own Merkle tree*
- It is *created as a snapshot* during a major compaction operation and is kept until is required to send it to the neighbouring nodes on the ring
- The *advantage* of this implementation is that it *reduces disk I/O*



Tombstone

A tombstone is an *indicator/mark* on the *data* that it has been *deleted but not removed entirely* yet

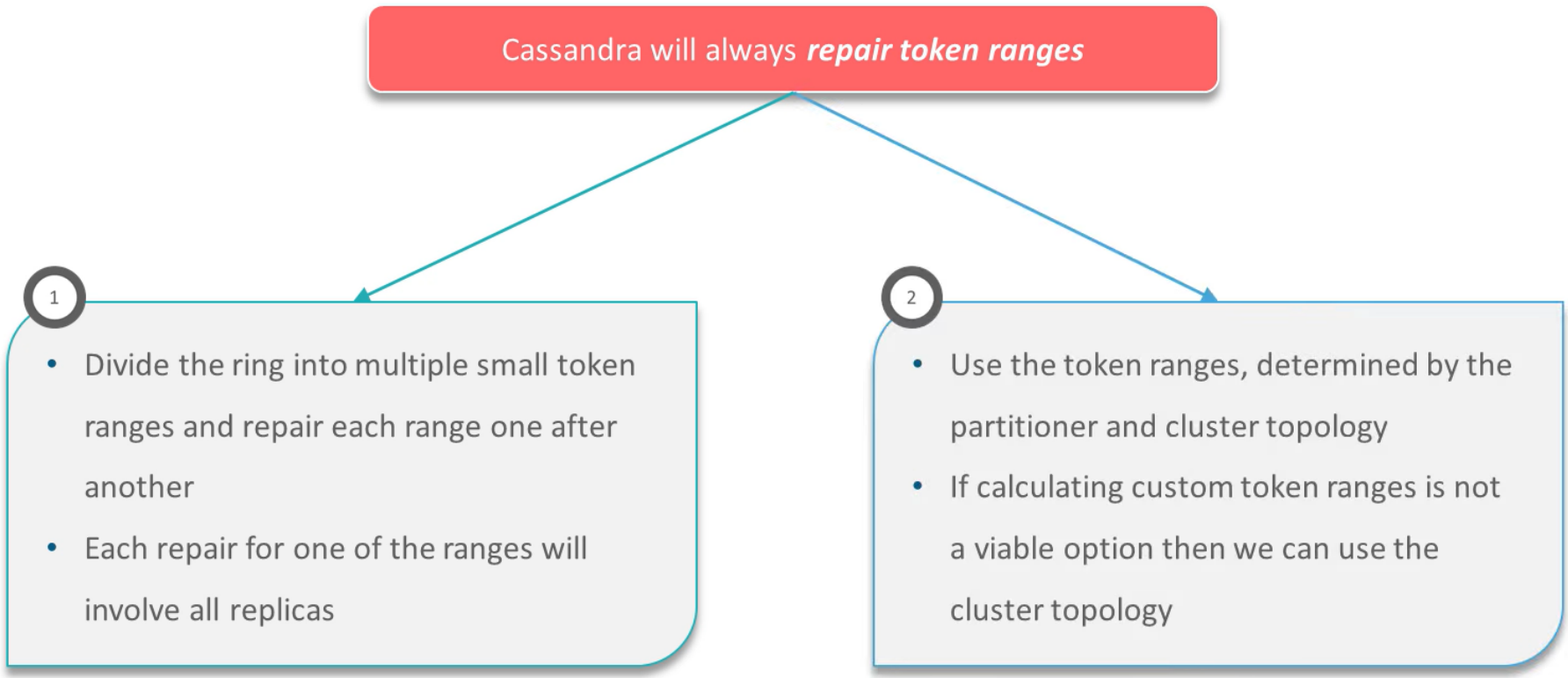
- When a *delete operation* is *executed*, the data is *not immediately deleted*
- It's treated as an *update operation* that places a tombstone on the value
- A tombstone is a *Deletion Marker* that is required to *suppress older data in SSTables* until compaction can run
- *The Tombstone* can be *propagated* to other *replicas*
- Tombstones are *discarded* on *Major Compaction*

Node for Repair

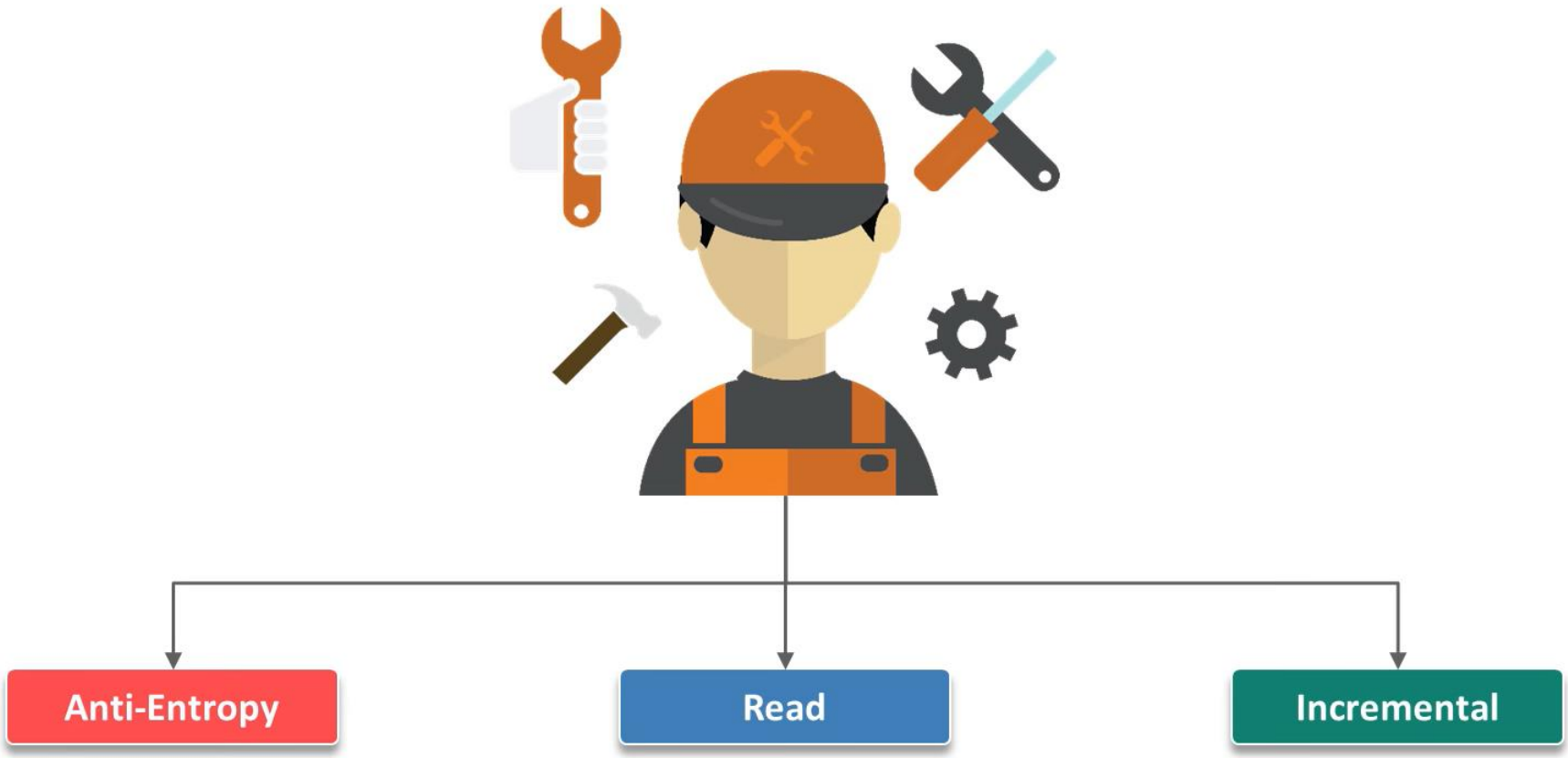
- Nodes can become unreachable as they experience:
  - hardware failures
  - get cut off the network
  - will shut down for system maintenance
  - fail to respond under heavy load
  - run out of disk space

- 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

Node for Repair

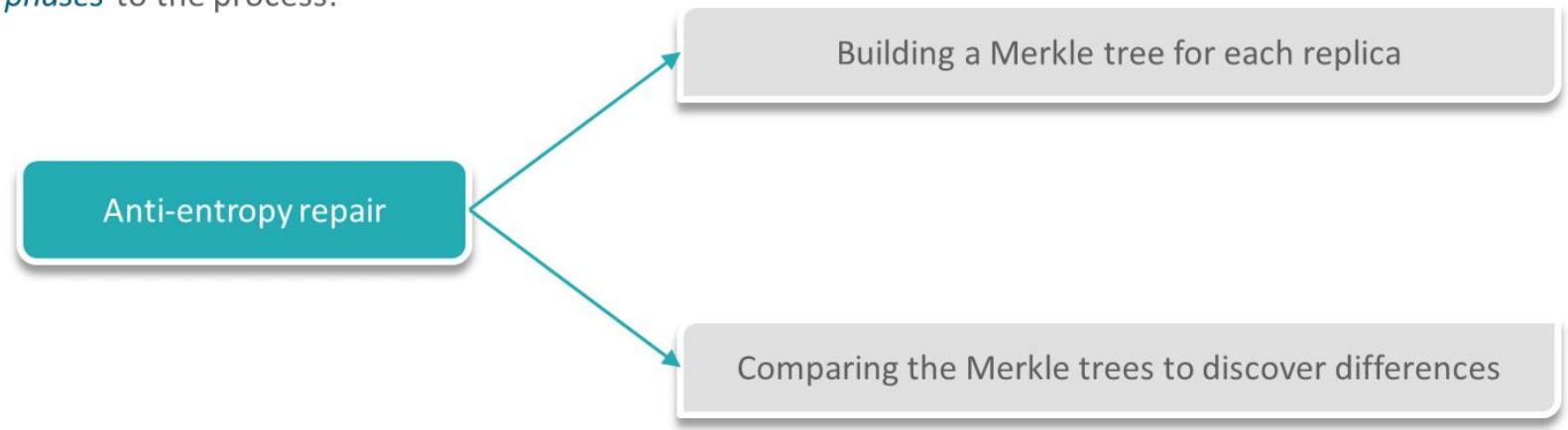


Types of Repair



Anti-Entropy Repair

- *Anti-entropy repair* is *triggered manually*
- It has *two phases* to the process:



- Anti-entropy repair is very useful and is often recommended to *run periodically to keep data in sync*

Read Repair

Read Repair implies that Cassandra will *send a write request* to the nodes with *stale data* to get them up to date with the *newer data* returned from the original read operation

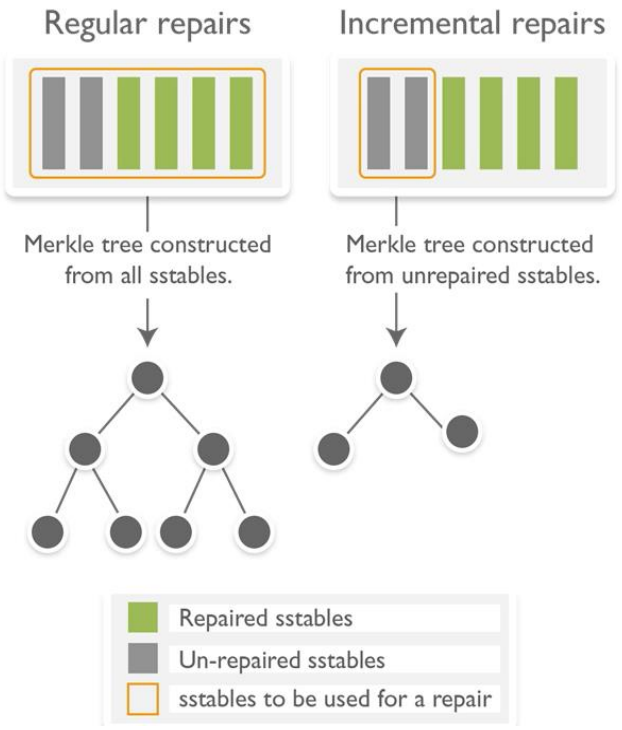
- In a *read operation*, if some *nodes respond* with data that is *inconsistent* with the *response of newer nodes*, a *Read Repair* is performed on the old nodes
- It ensures *consistency* throughout the *node ring*
- Done by *pulling all of the data* from the node and *performing a merge*, and then *writing it* back to the nodes that were *out of sync*
- The *detection of inconsistent data* is made by comparing *timestamps* and *checksums*
- The *Read Repair stage manages* the task of *keeping the data fresh* in the background

Incremental Repair

Instead of building a Merkle tree out of all SSTables (repaired or not), it builds the tree out of un-repaired SSTables(only)

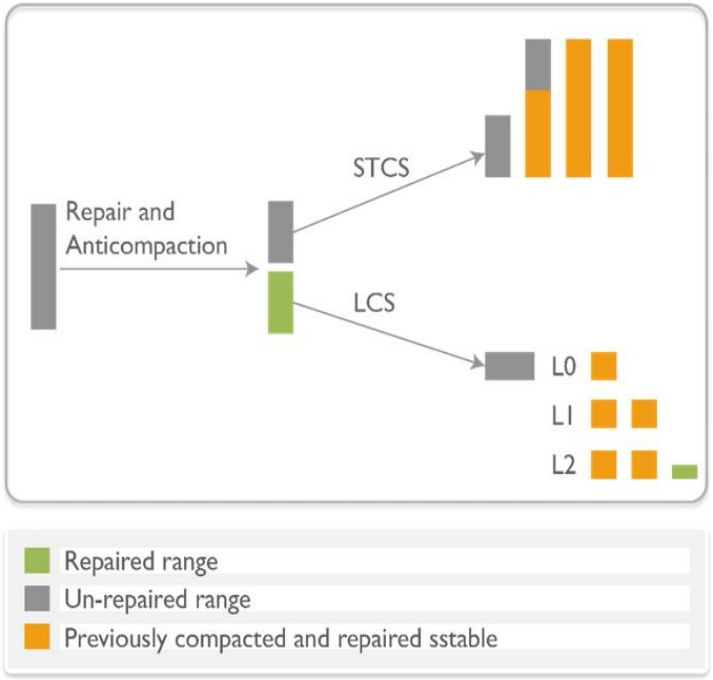
Therefore,

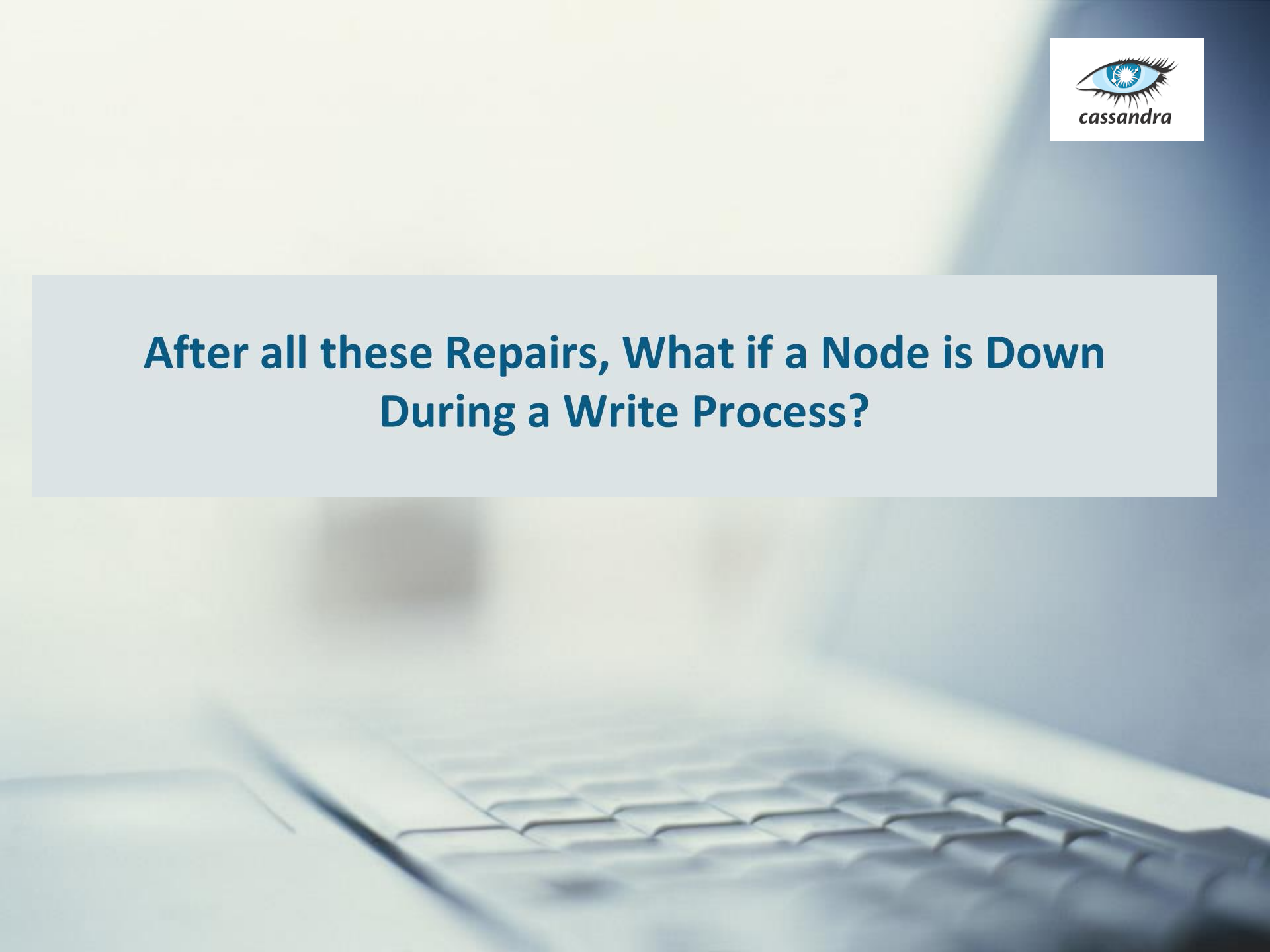
- The size of the Merkle tree to be built is smaller and requires much less computing resources
- When doing Merkle tree comparison, less data will be compared and potentially streamed over the network
- Unrepaired SSTables are determined by a *repairedAt* property in its metadata
- An anti-compaction occurs after the incremental repair to separate the repaired and unrepaired ranges



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

In order to *ensure general availability of the ring* in such situations, Cassandra implements a feature called *Hinted Handoff*

Hinted Handoff

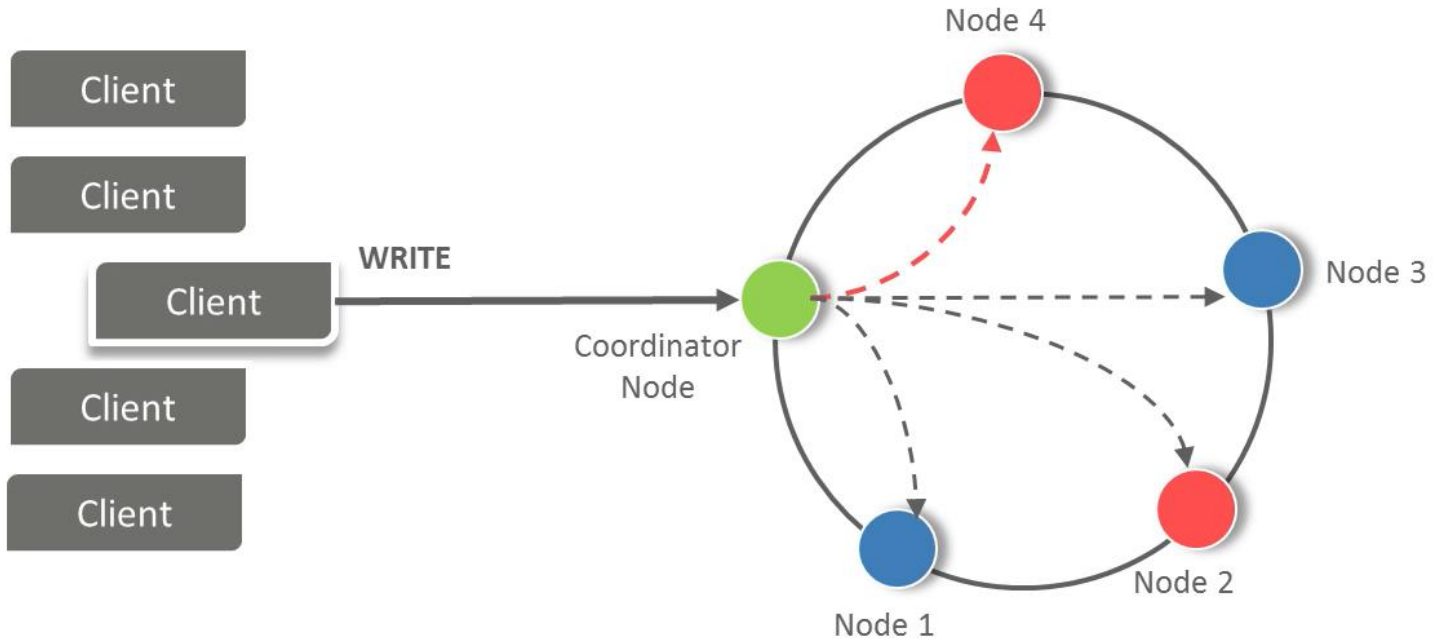
Hinted Handoff is a mechanism to *ensure availability*, *fault-tolerance* and *graceful degradation* in Cassandra

- It is comprised of:
  - Target node location, which is down
  - Partition which requires a replay
  - Data to be written
- The node that receives the hint will know when the unavailable node comes back online again, because of Gossip
- Coordinator stores the hint in its "*system.hints* " table

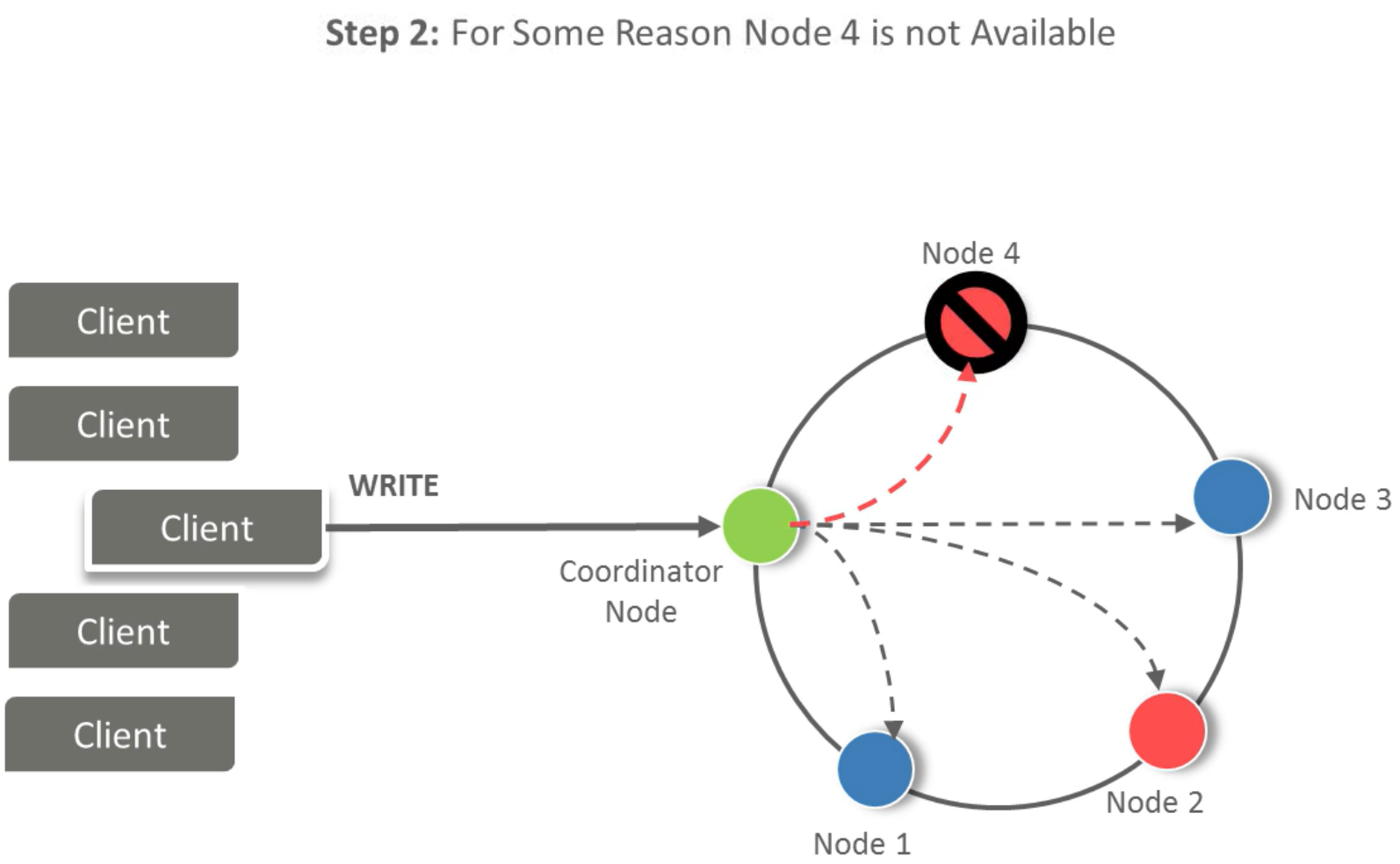
Let’s see how Hinted Handoff solves this Problem?

Hinted Handoff : Process

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

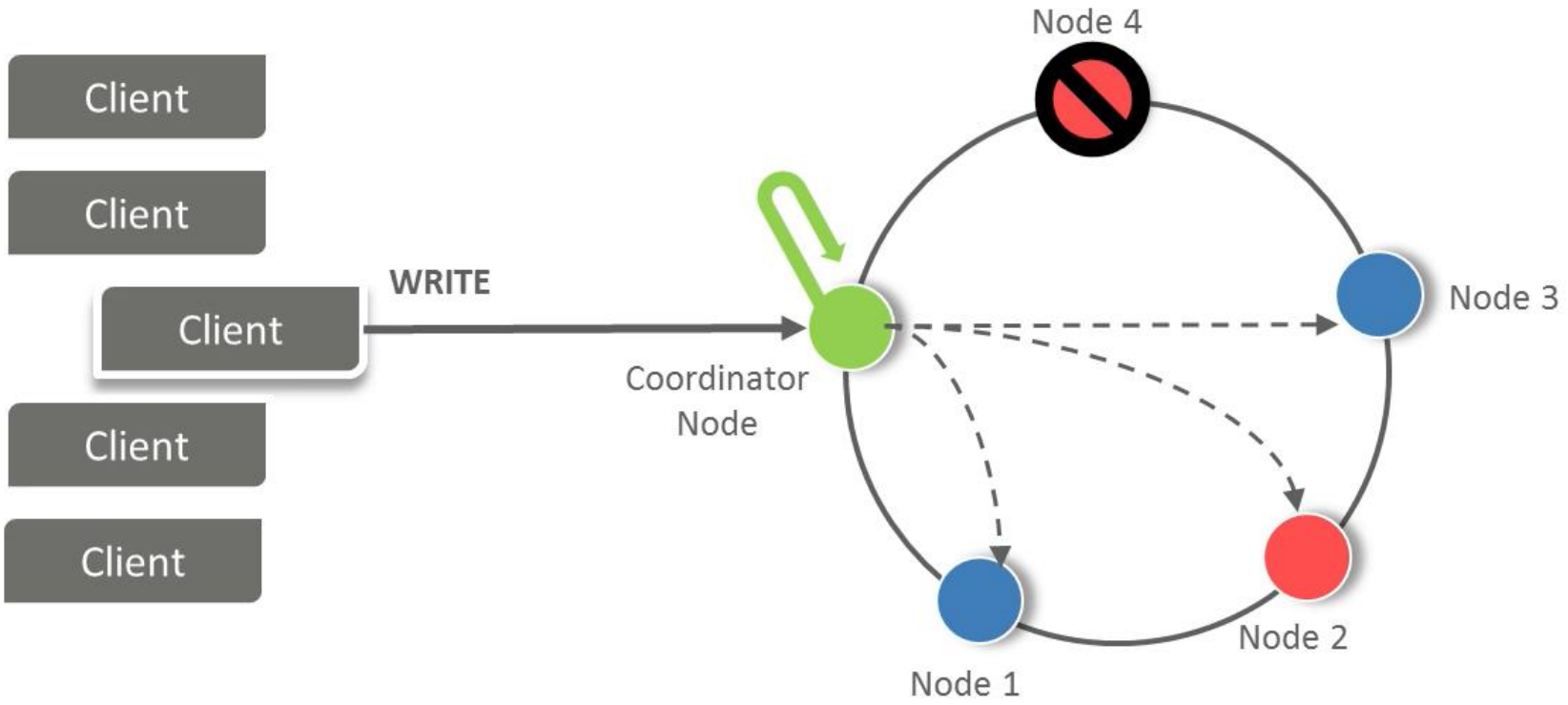


Hinted Handoff : Process



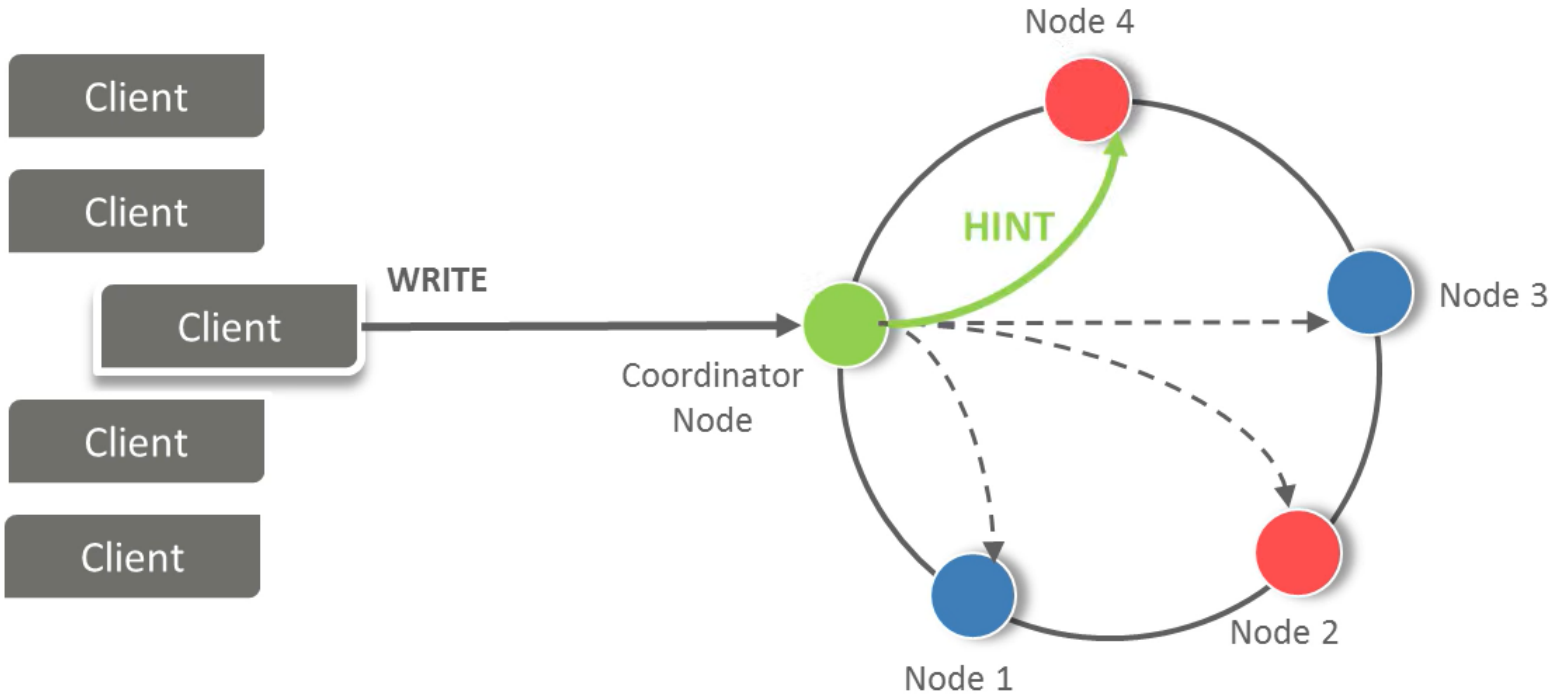
Hinted Handoff : Process

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

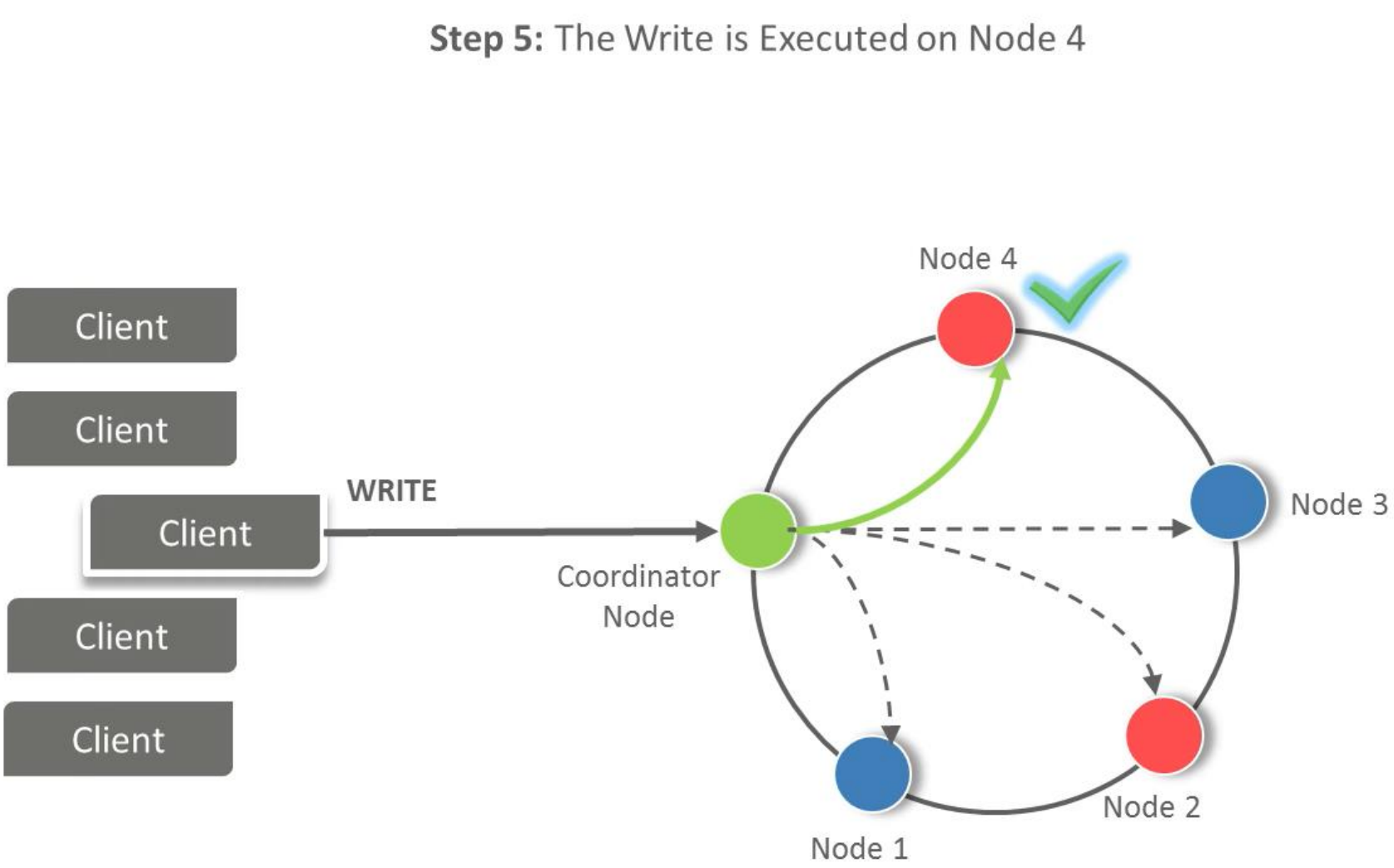


Hinted Handoff : Process

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



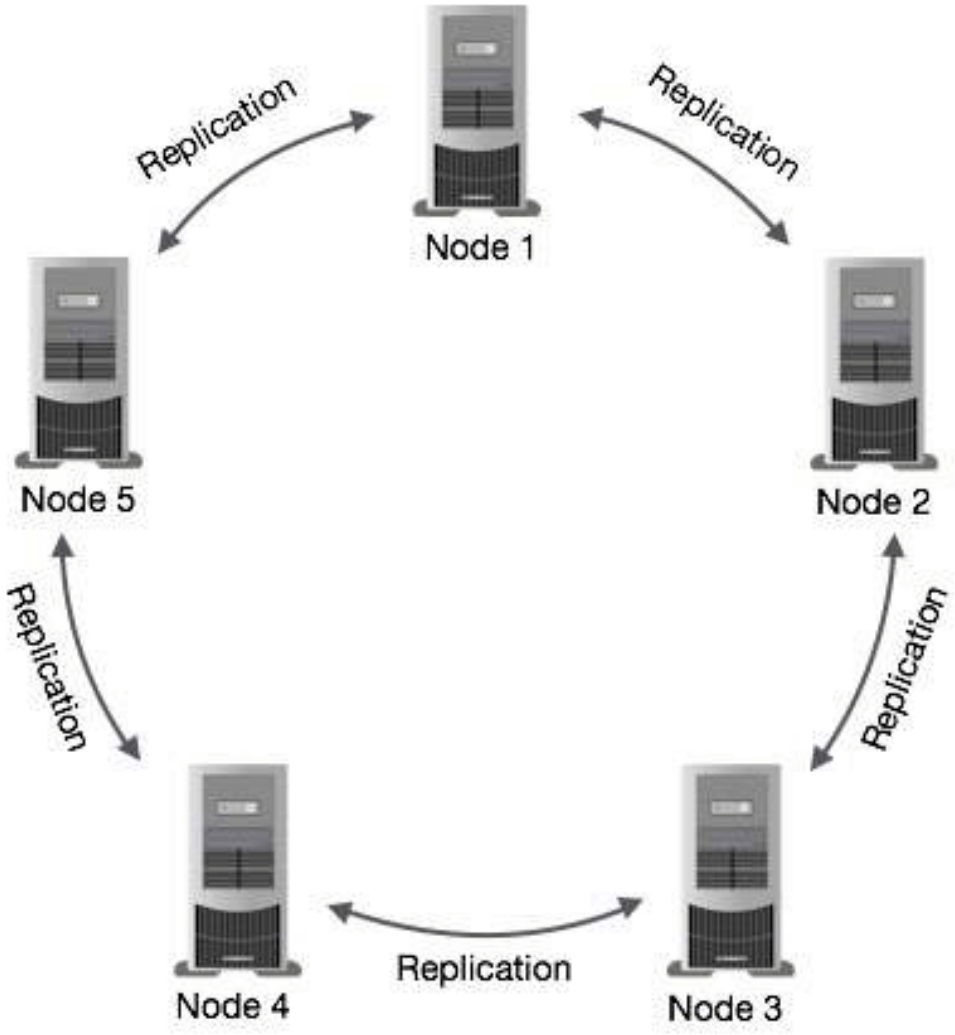
Hinted Handoff : Process



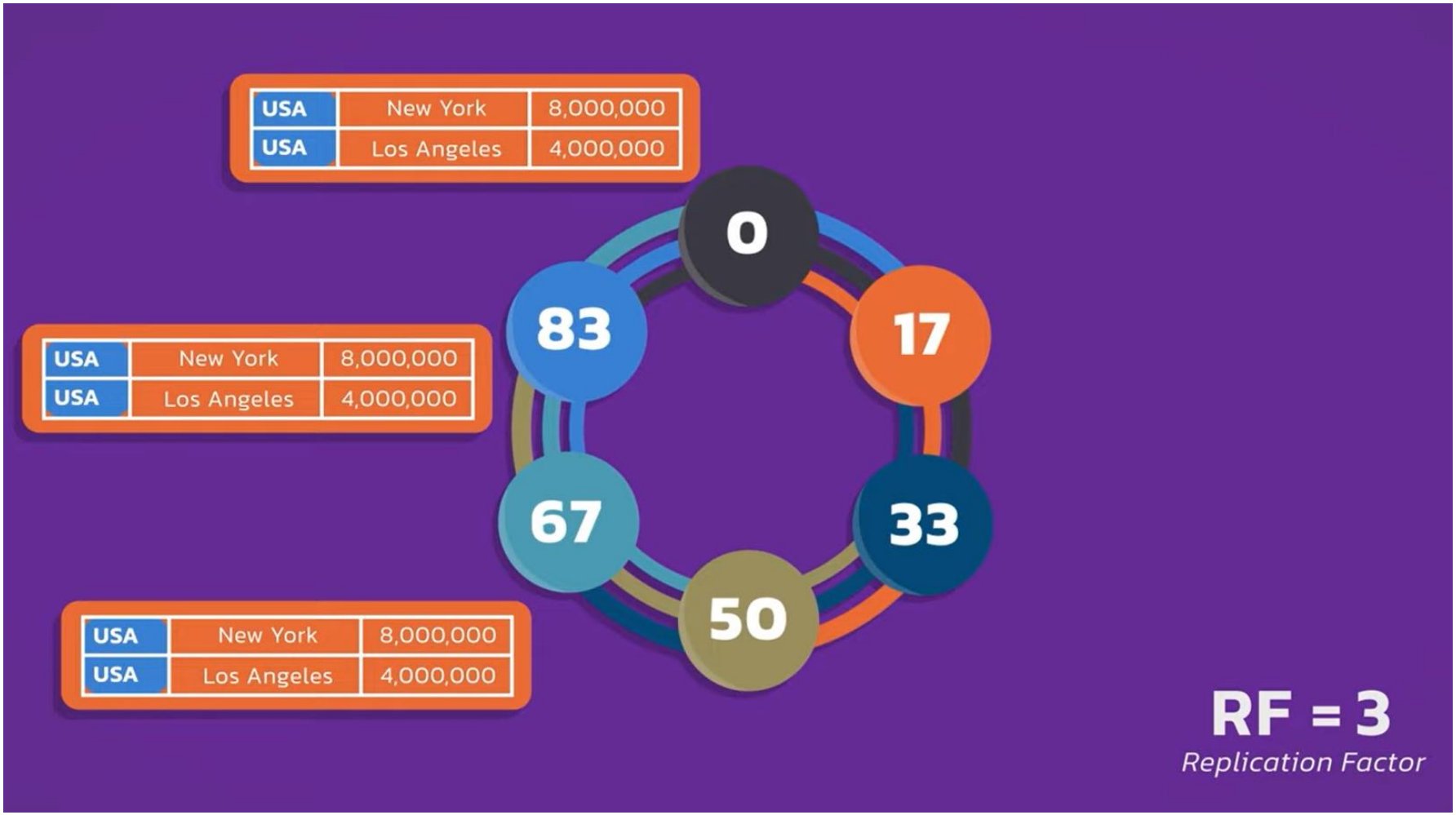
Data Replication in Cassandra

- 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.

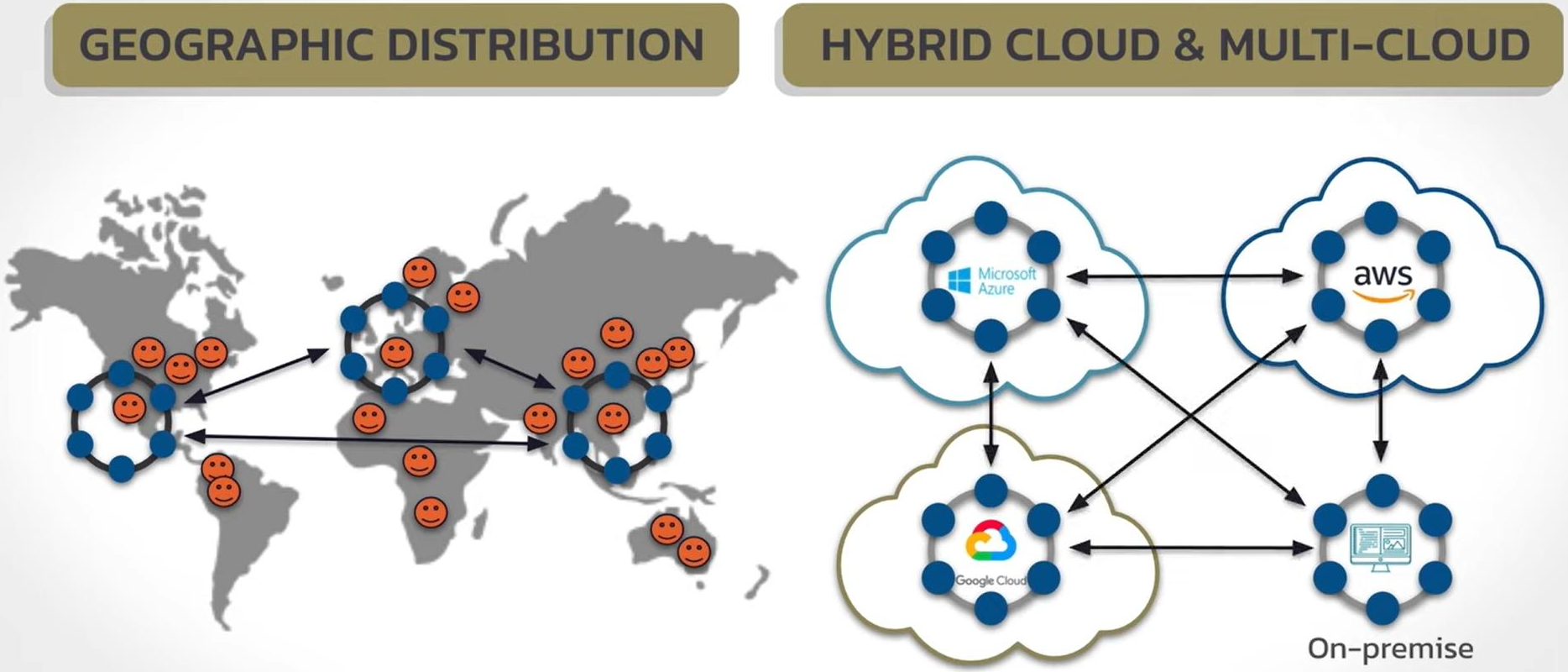
Data Replication in Cassandra



Data Replication in Cassandra



Distribution



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

CQL Type	Constants	Description
ascii	Strings	US-ascii character string
bigint	Integers	64-bit signed long
blob	blobs	Arbitrary bytes in hexadecimal
boolean	Booleans	True or False
counter	Integers	Distributed counter values 64 bit
decimal	Integers, Floats	Variable precision decimal
double	Integers, Floats	64-bit floating point
float	Integers, Floats	32-bit floating point
frozen	Tuples, collections, user defined types	stores cassandra types
inet	Strings	IP address in ipv4 or ipv6 format
int	Integers	32 bit signed integer

Cassandra Data Types

list		Collection of elements
map		JSON style collection of elements
set		Collection of elements
text	strings	UTF-8 encoded strings
timestamp	Integers, Strings	ID generated with date plus time
timeuuid	uuids	Type 1 uuid
tuple		A group of 2,3 fields
uuid	uuids	Standard uuid
varchar	strings	UTF-8 encoded string
varint	Integers	Arbitrary precision integer

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 species 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.

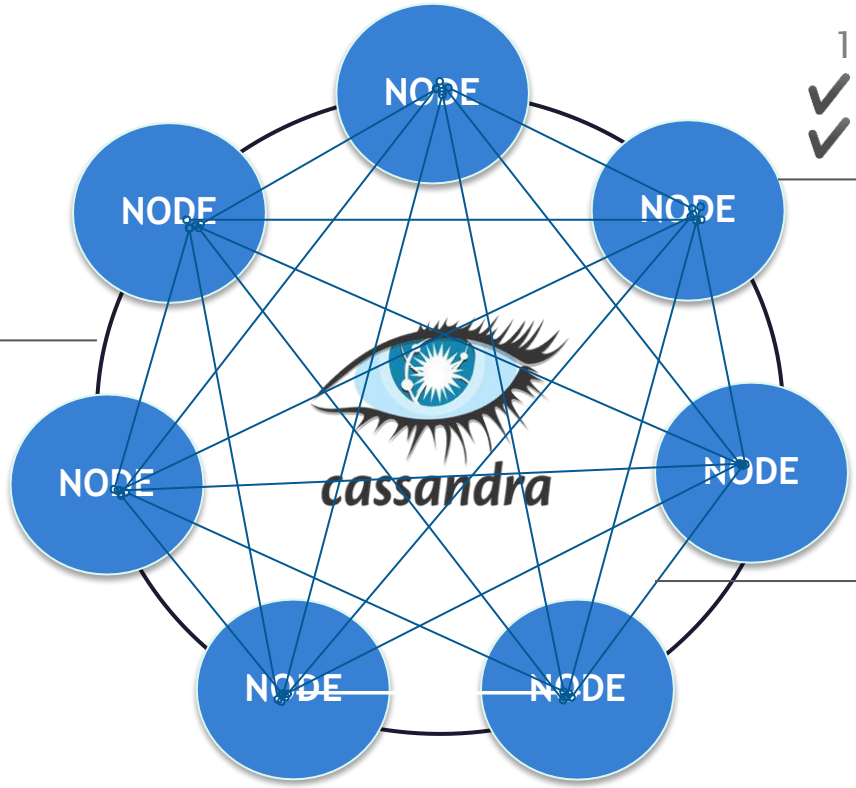
Wrong usecases for cassandra

- 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. LSD, Sulphuric or any other kind. If you think you need it go elsewhere. Many times people think they do need it when they don't.
- Aggregates: Cassandra does not support aggregates, if you need to do a lot of them, think another database.
- 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



1Installation =1NODE
- ✔ Capacity =~ 2–4TB
- ✔ Throughput =LOTS Tx/sec/core

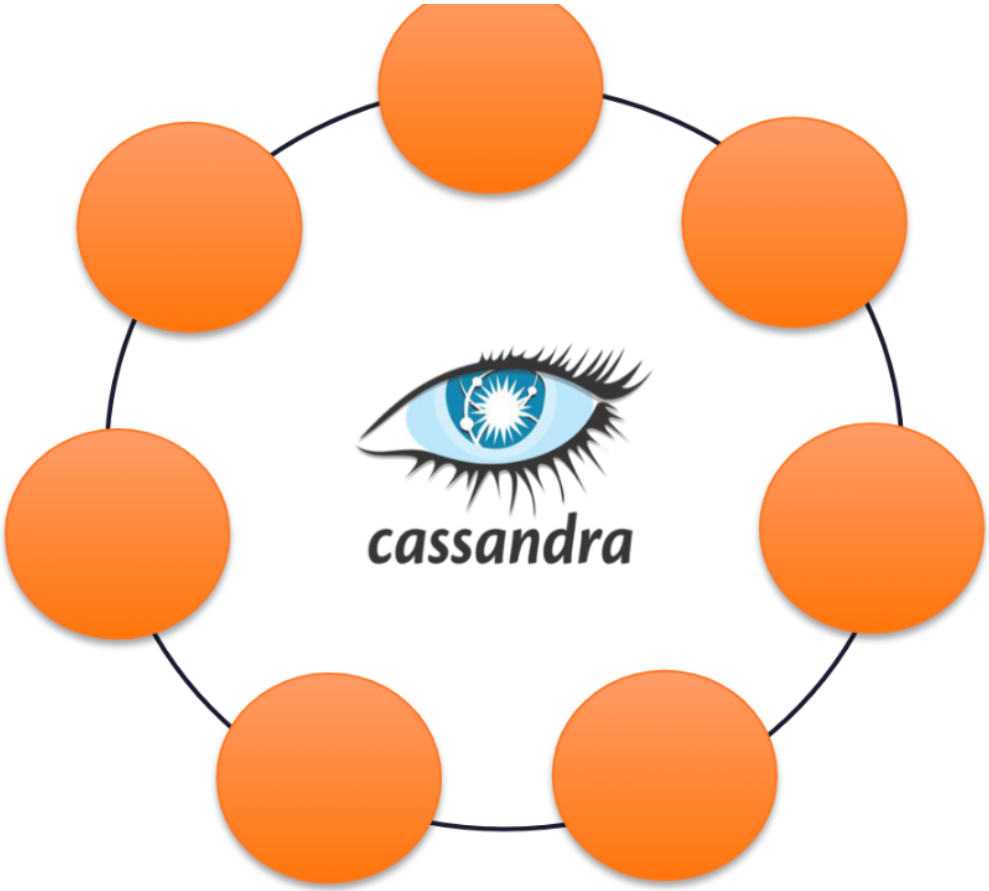
DataCenter | Ring

Communication:
- ✔ Gossiping

Apache Cassandra™ = NoSQL Distributed Database

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

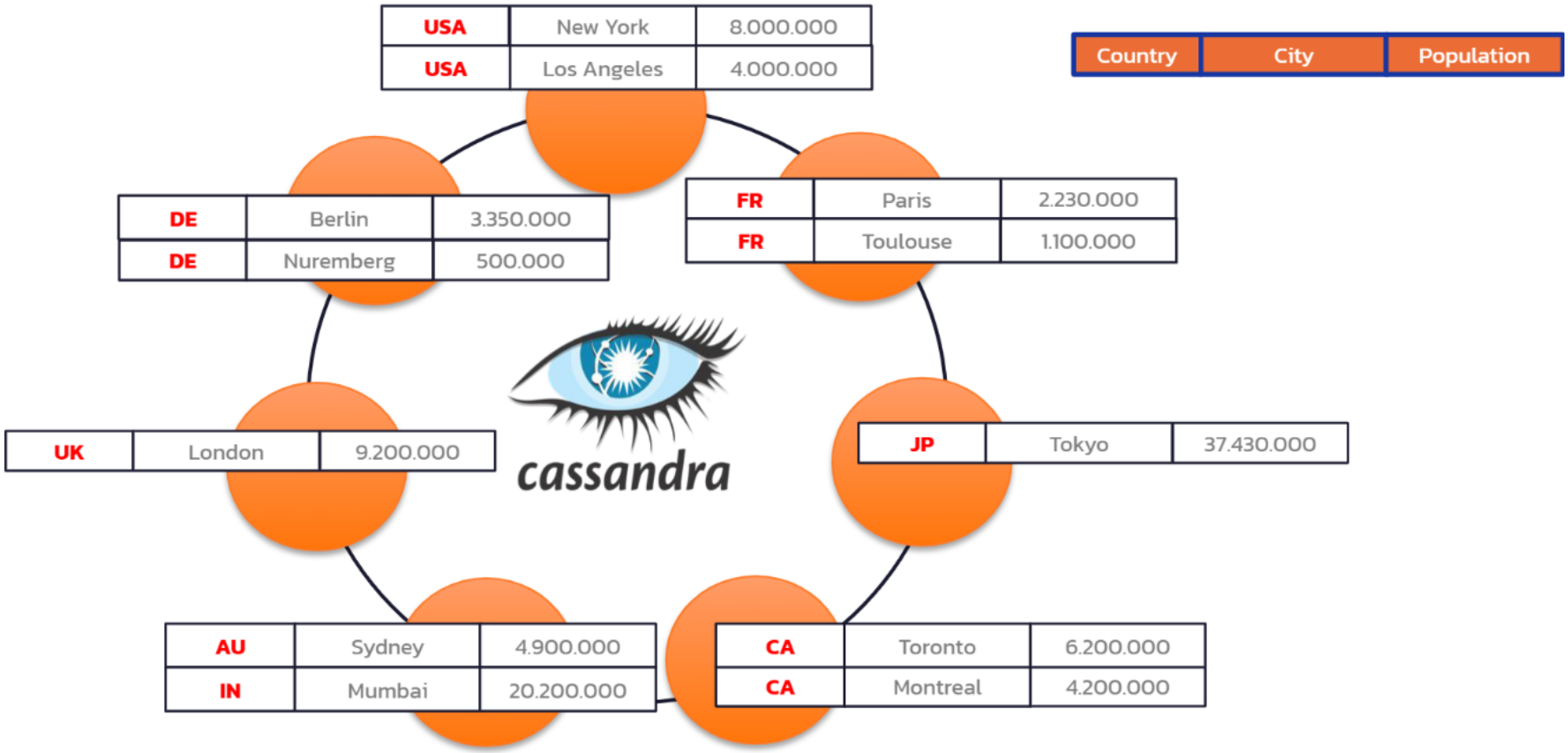
Data Distributed



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

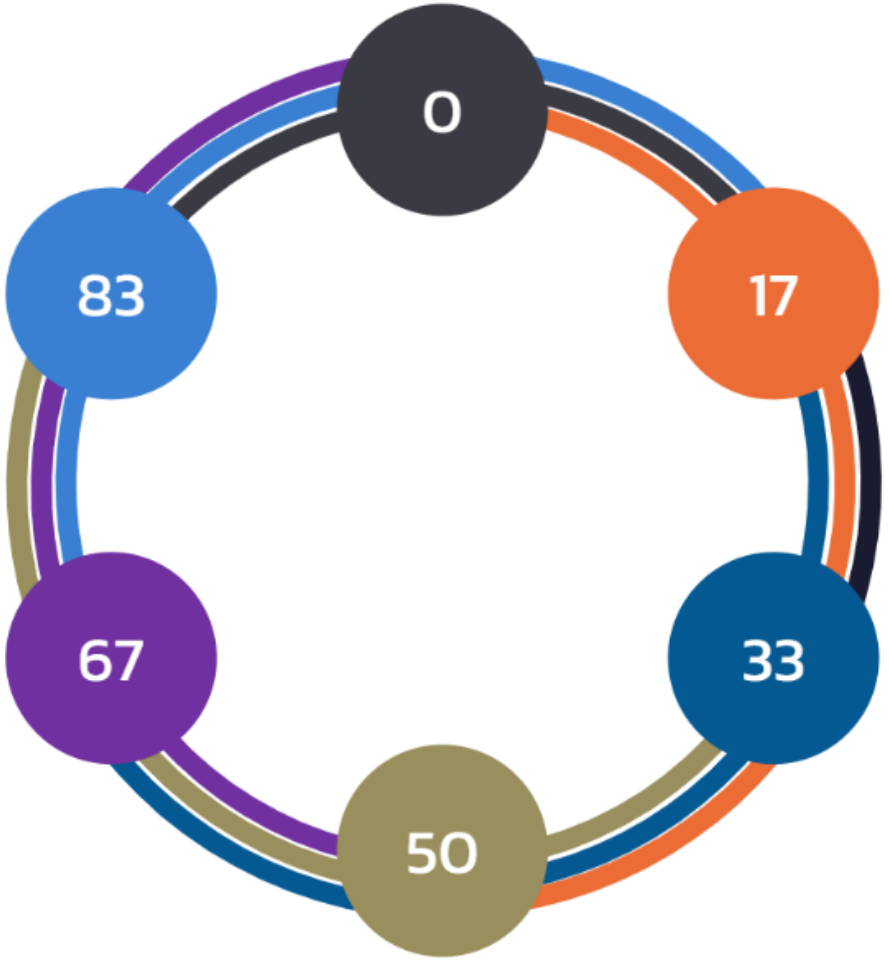
Data Distributed



Data is Replicated

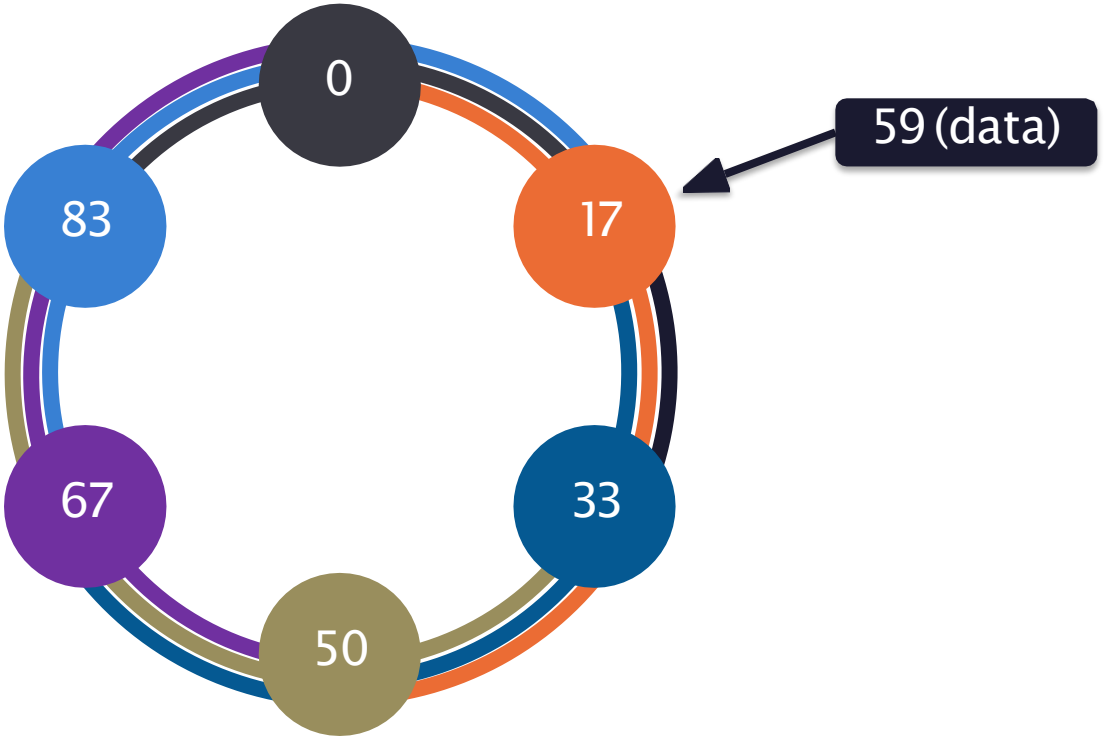
RF = 3

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



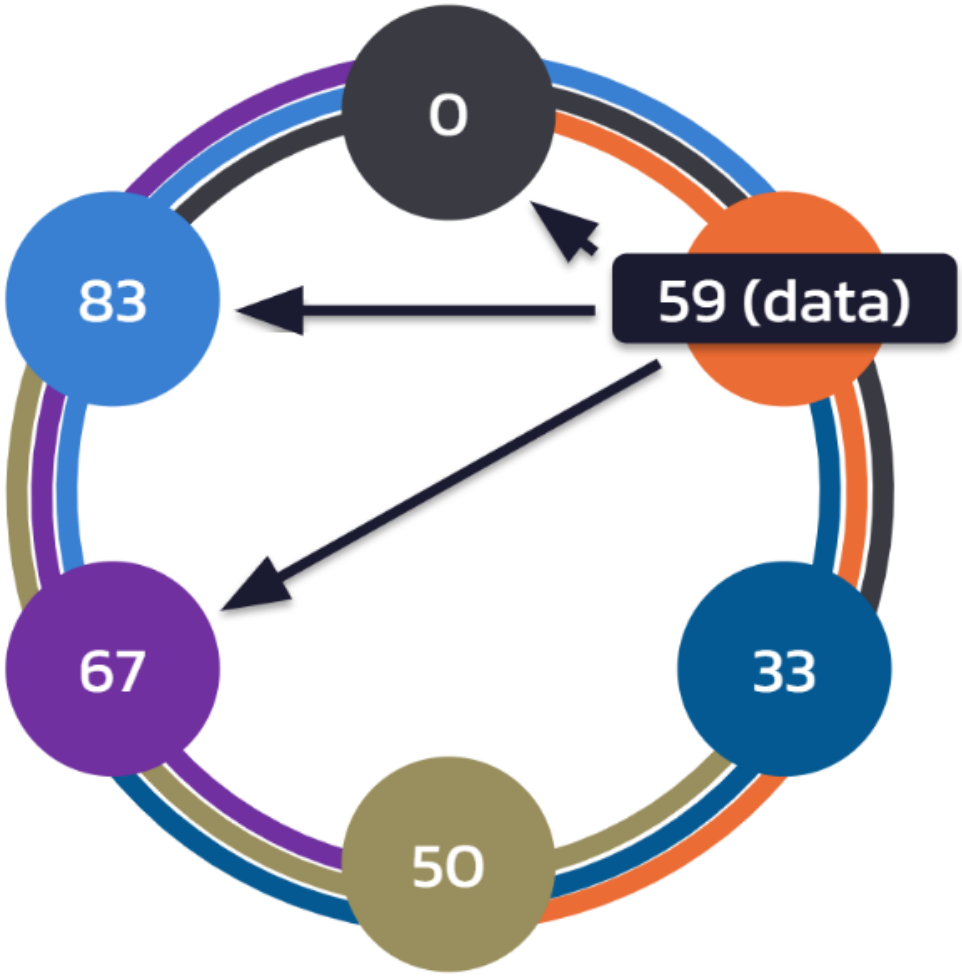
Replication within the Ring

RF =3



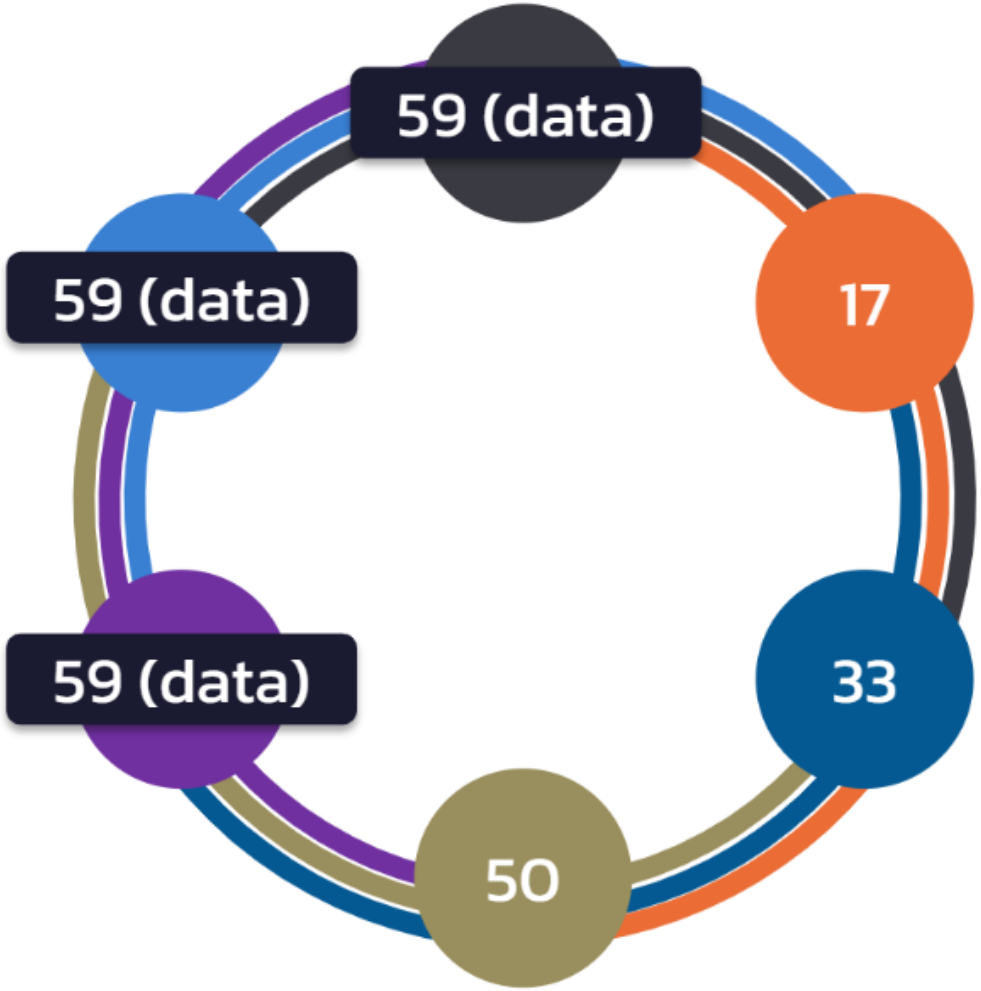
Replication within the Ring

RF = 3



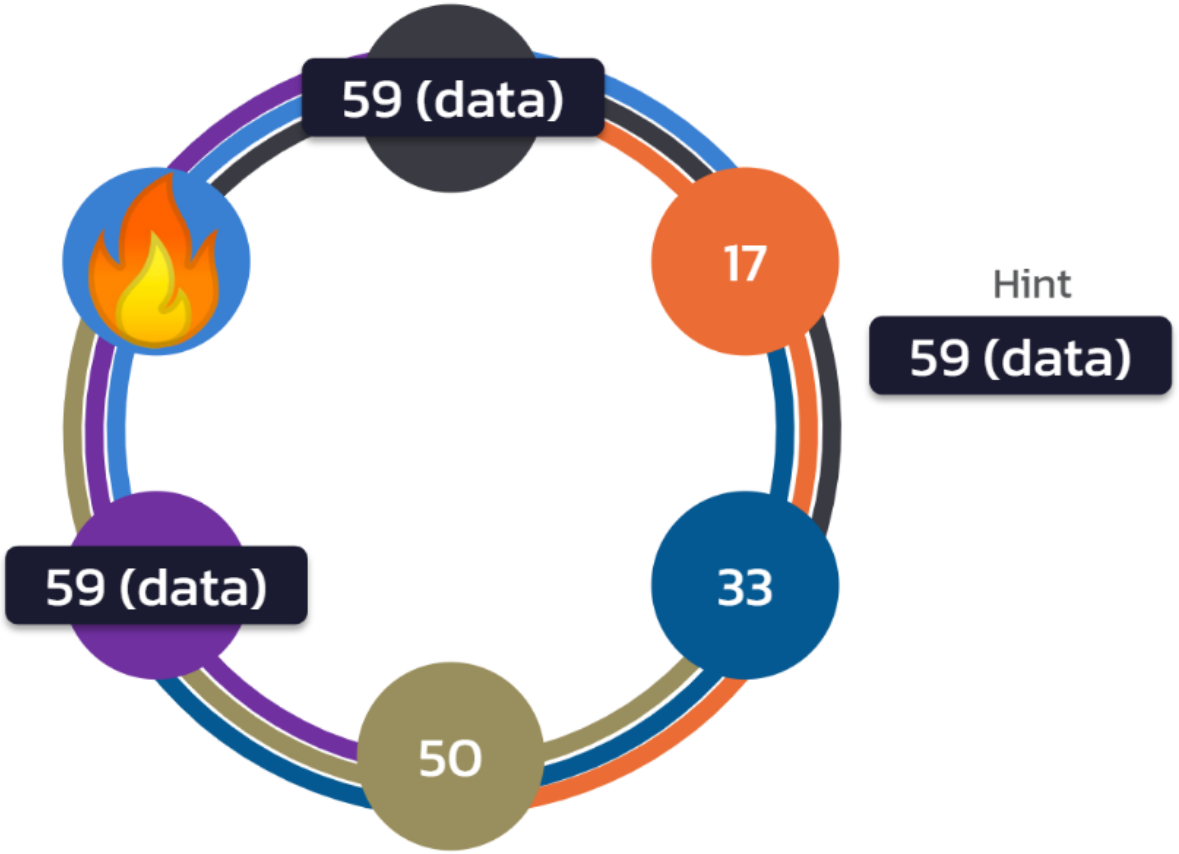
Replication within the Ring

RF = 3

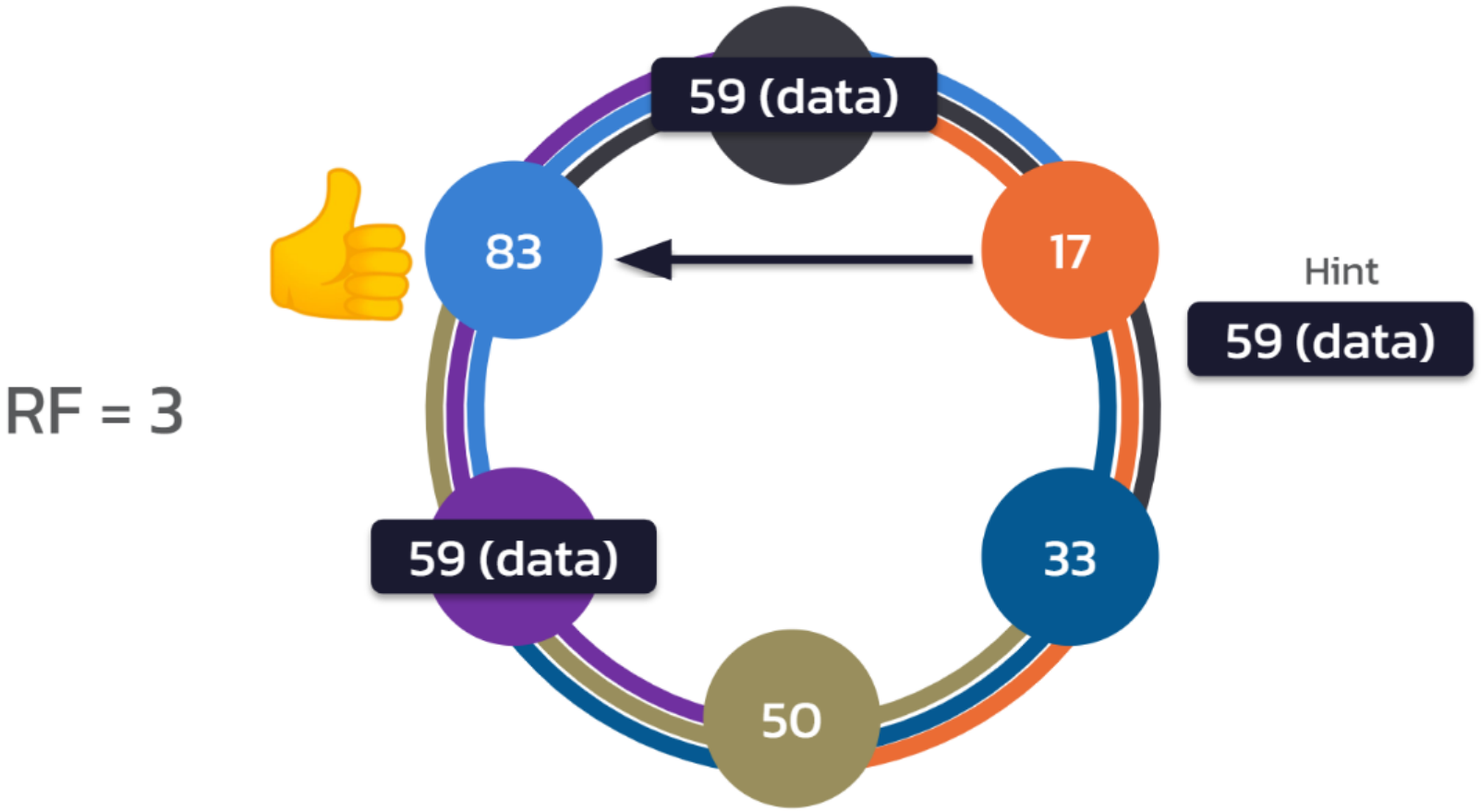


Node Failure

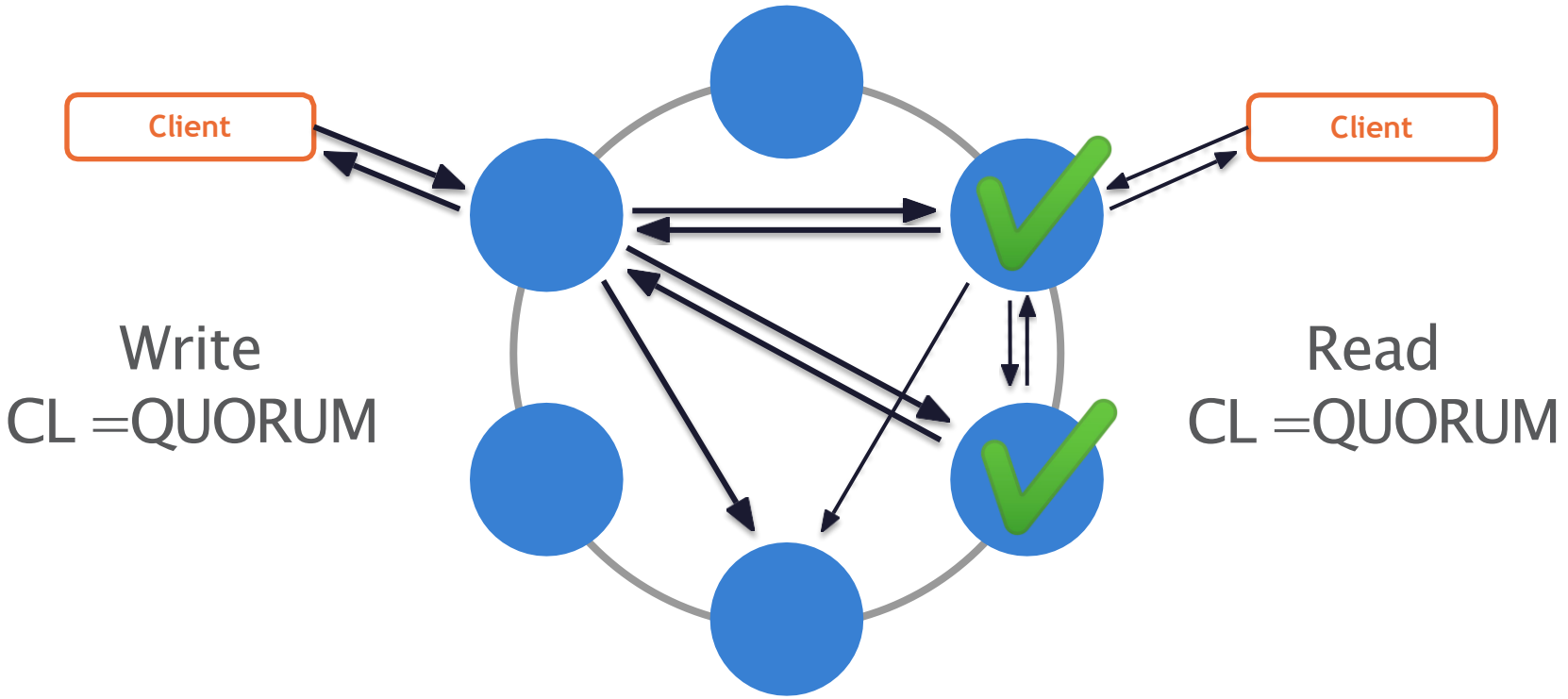
RF = 3



Node Failure Recovered



Immediate Consistency – A Better Way



Intro to Cassandra for Developers

1. Tables, Partitions
2. The Art of Data Modelling
3. What's NEXT?

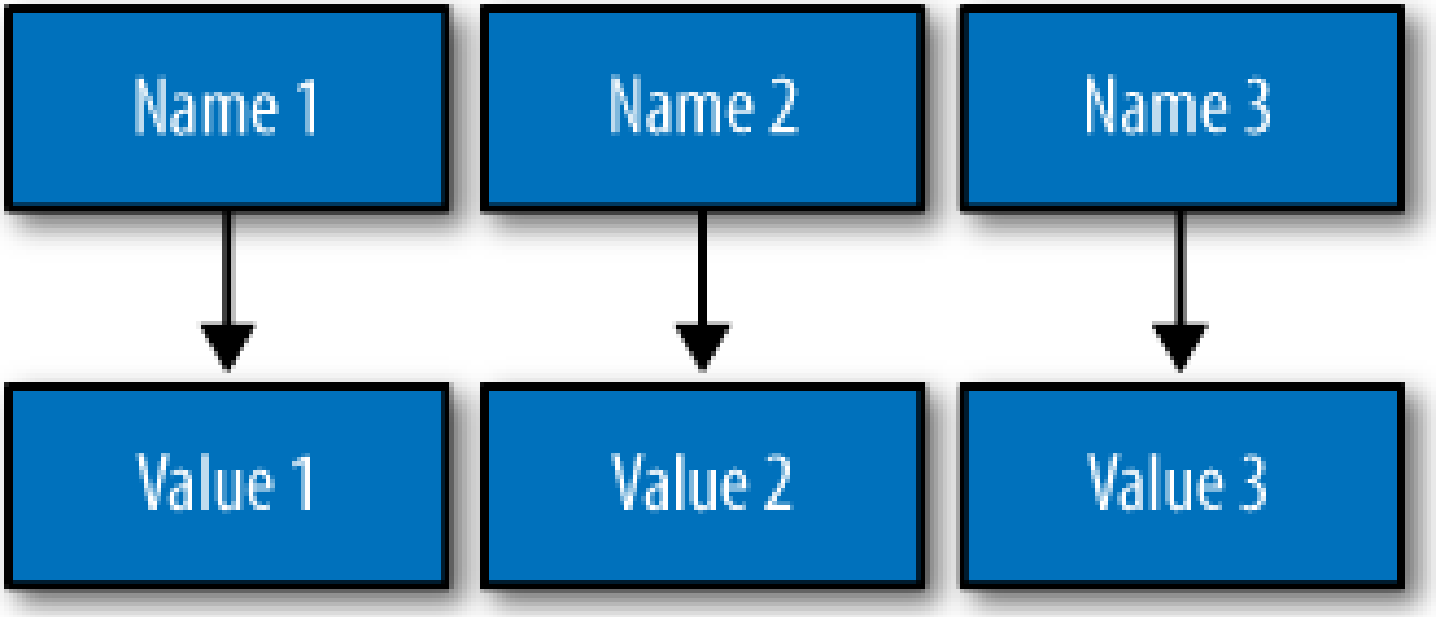
Data Structure: a Cell

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

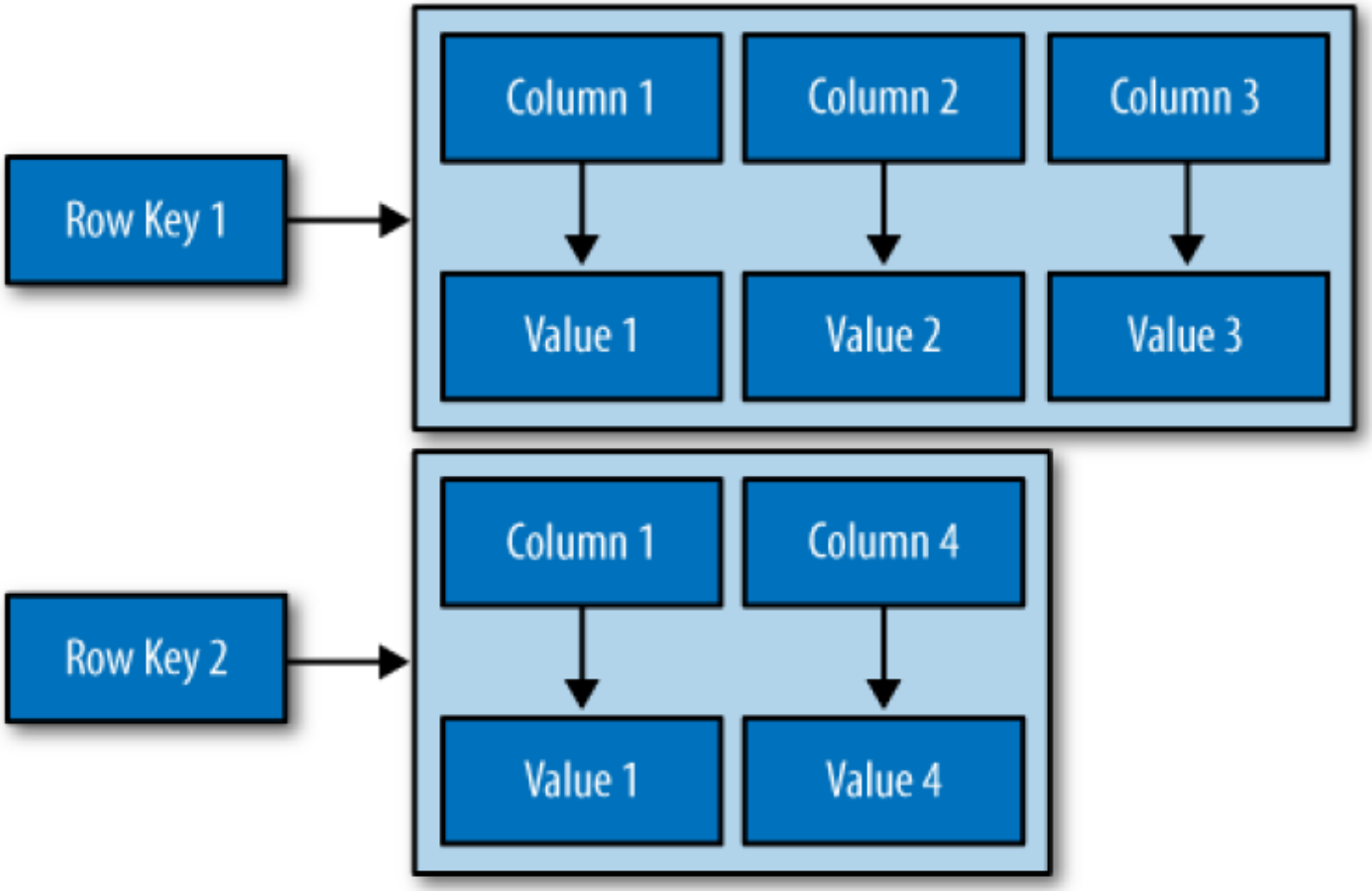
A list of values

Data Structure: a Cell

A map of name/value pairs



Column Family



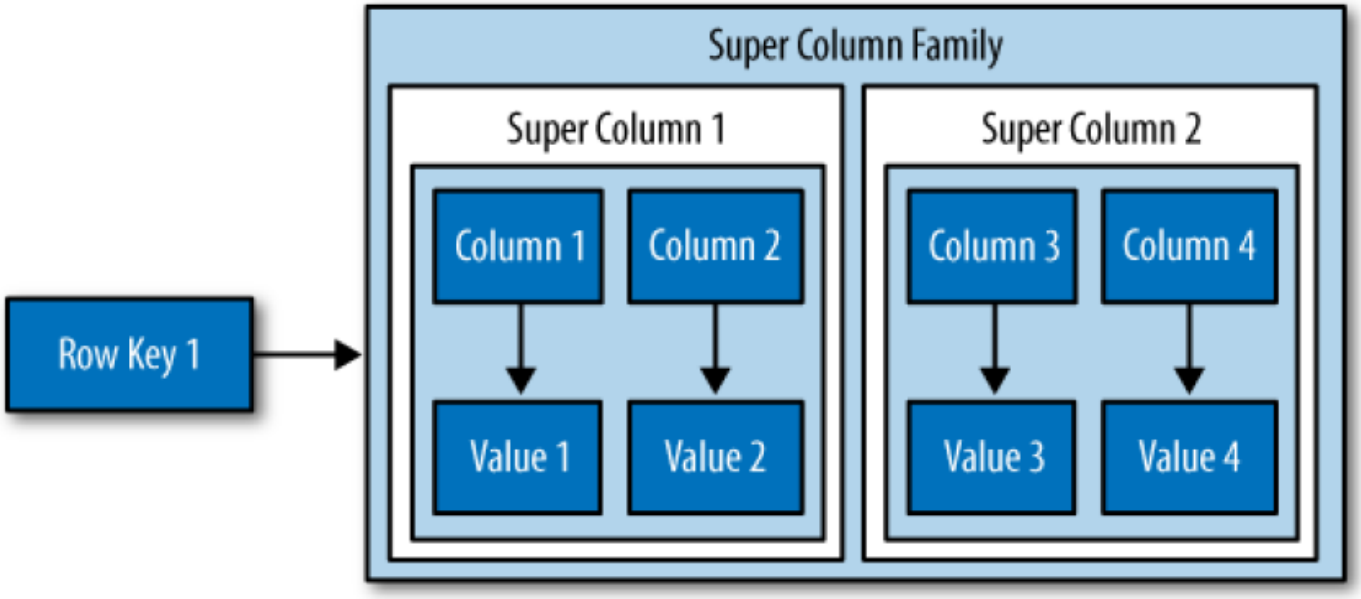
Column Family

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.

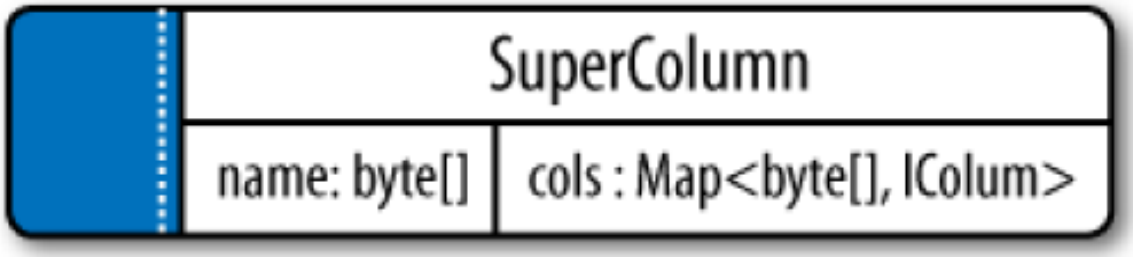
Column Sorting

- 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.

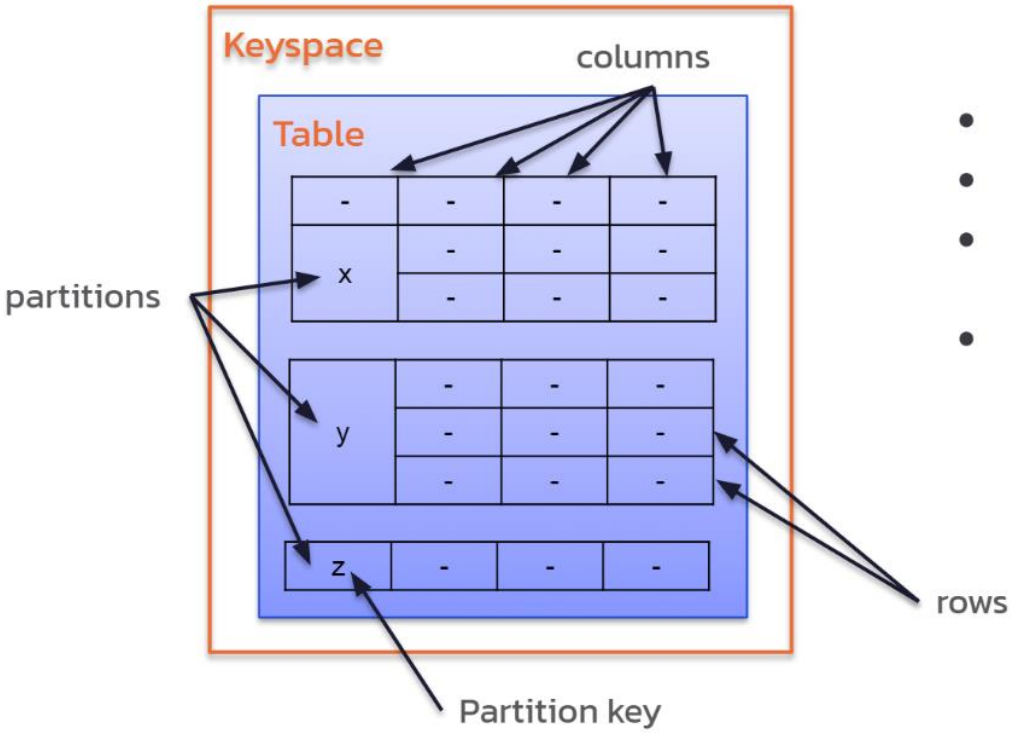
ID	First Name	Last Name	Department
1	John	Doe	Wizardry
399	Marisha	Chapez	Wizardry
415	Maximus	Flavius	Wizardry

Data Structure: a Table

A group of columns and rows storing partitions.

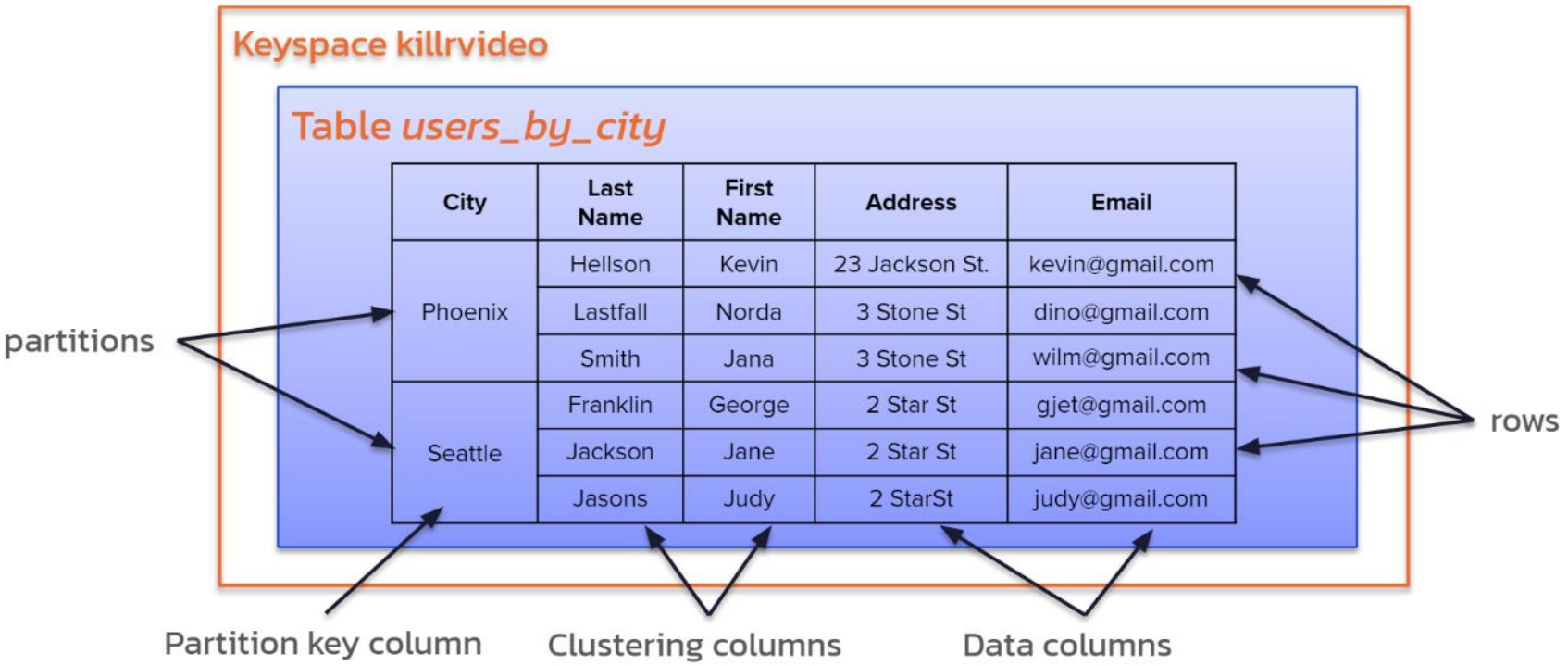
ID	First Name	Last Name	Department
1	John	Doe	Wizardry
2	Mary	Smith	Dark Magic
3	Patrick	McFadin	DevRel

Data Structure: Overall



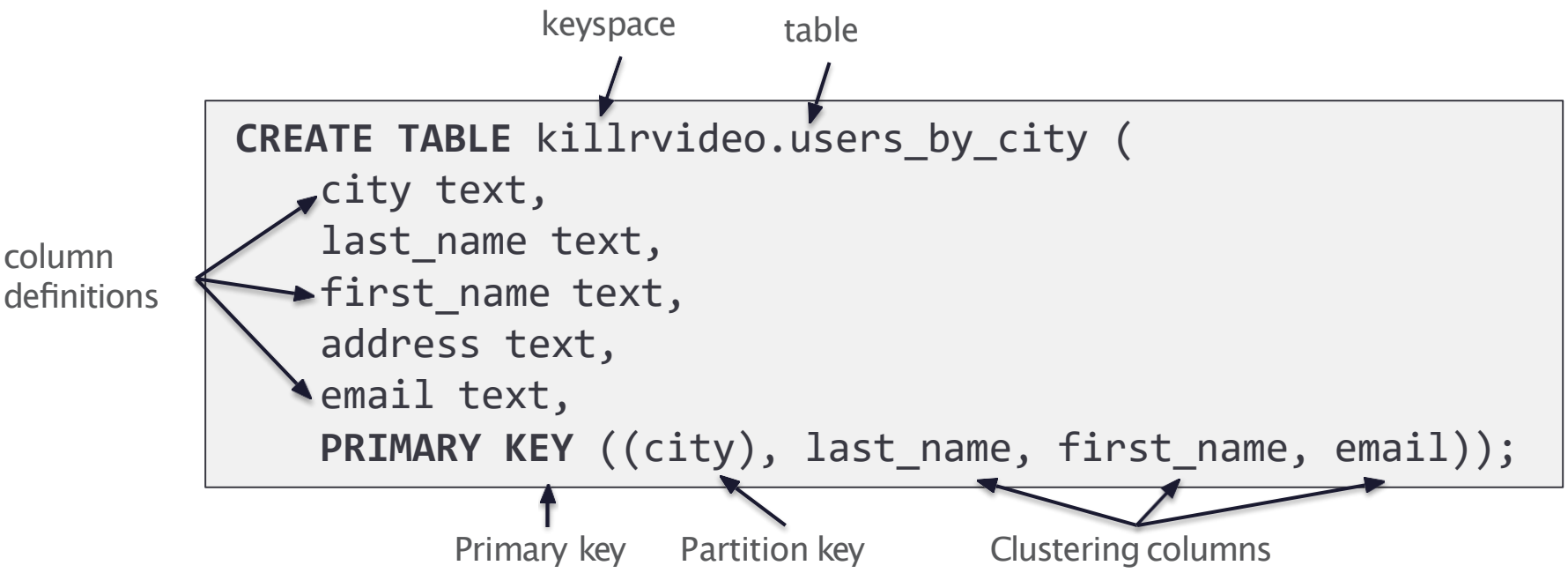
- Tabular data model, with one twist
- *Tables* are organized in *rows* and *columns*
- Groups of related rows called *partitions* are stored together on the same node (or nodes)
- Each row contains a *partition key*
  - One or more columns that are hashed to determine which node(s) store that data

Example Data: Users Organized By City



City	Last Name	First Name	Address	Email
Phoenix	Hellson	Kevin	23 Jackson St.	kevin@gmail.com
	Lastfall	Norda	3 Stone St	dino@gmail.com
	Smith	Jana	3 Stone St	wilm@gmail.com
Seattle	Franklin	George	2 Star St	gjet@gmail.com
	Jackson	Jane	2 Star St	jane@gmail.com
	Jasons	Judy	2 StarSt	judy@gmail.com

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));
```

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));
```

Partition key

Clustering columns

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.

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

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

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

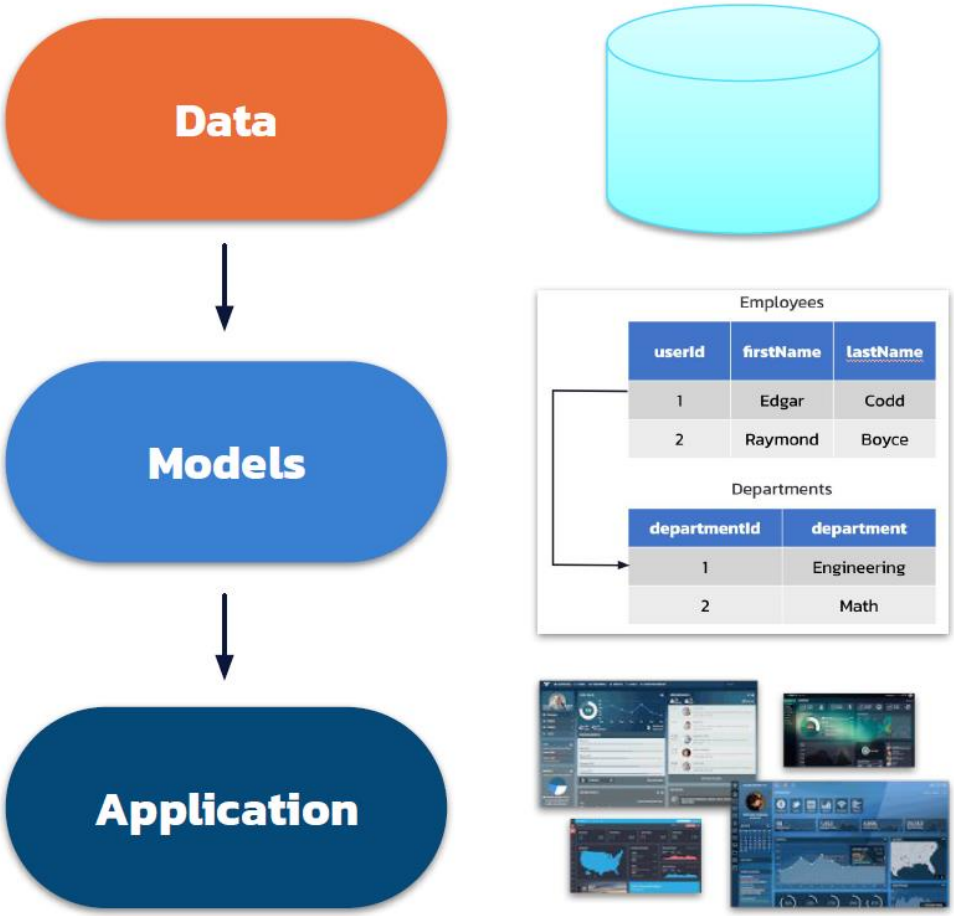
userId	deptId	firstName	lastName
1	1	Edgar	Codd
2	1	Raymond	Boyce

Departments

departmentId	department
1	Engineering
2	Math

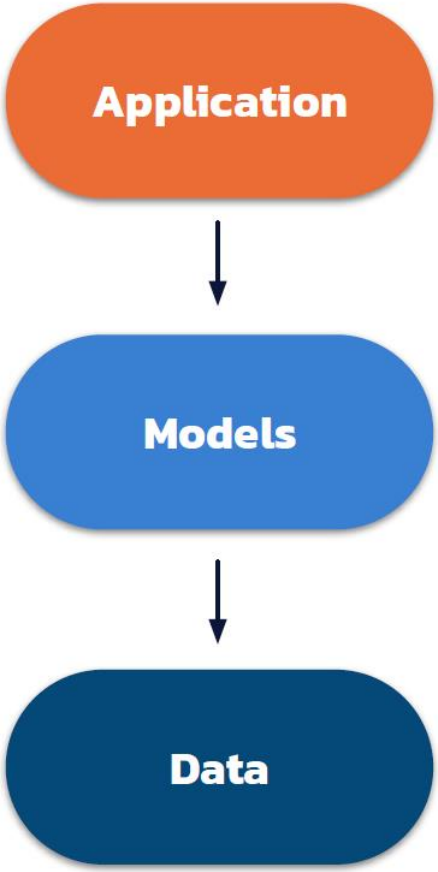
Relational Data Modeling

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



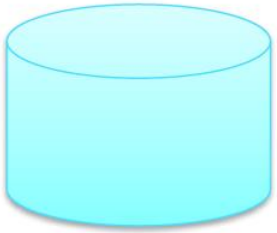
NoSQL Data Modelling

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

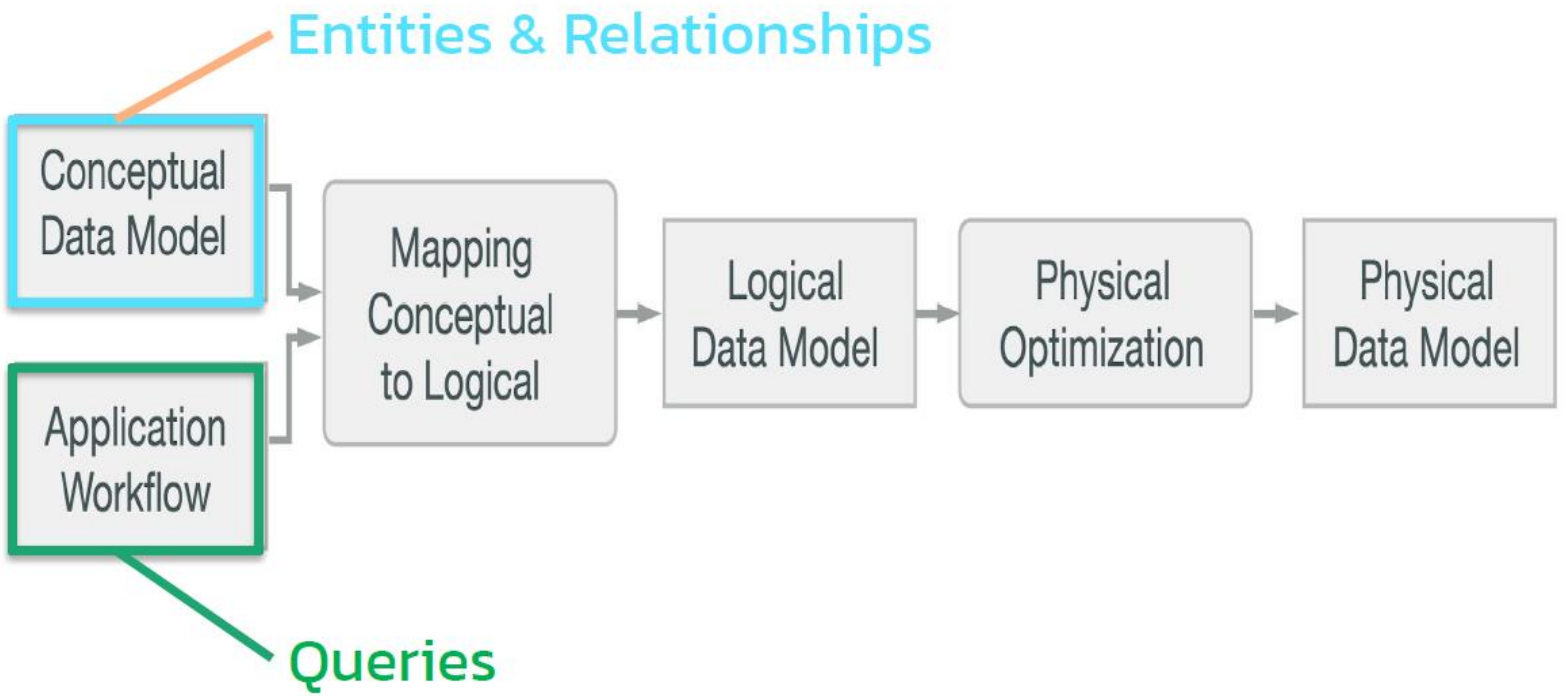


Employees

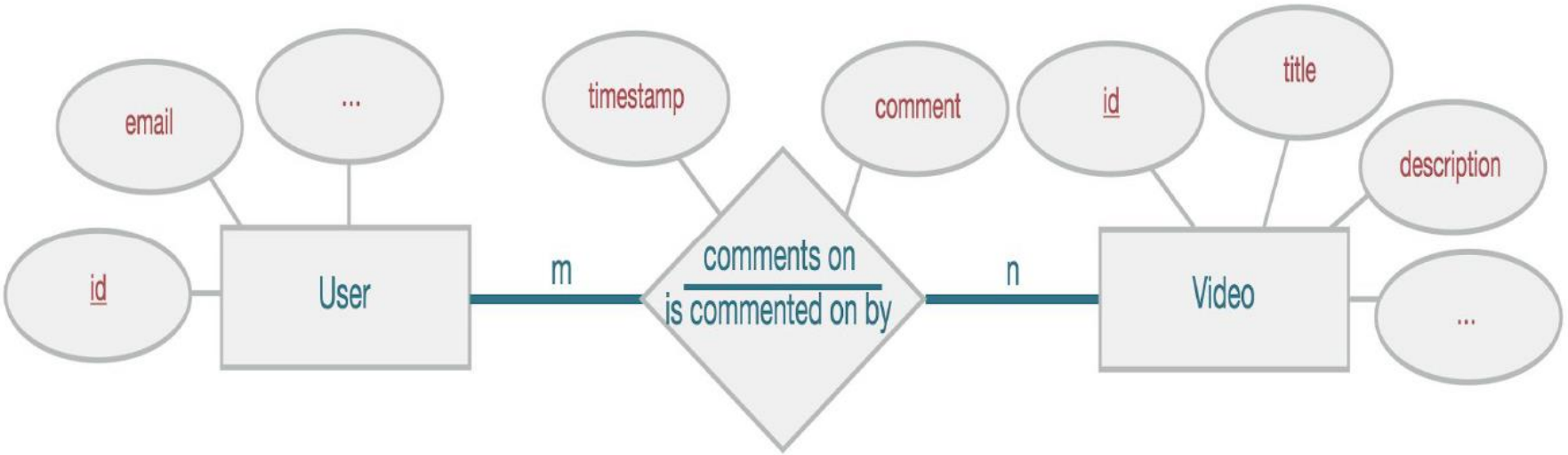
userId	firstName	lastName	department
1	Edgar	Codd	Engineering
2	Raymond	Boyce	Math
3	Sage	Lahja	Math
4	Juniper	Jones	Botany



Designing Process Step By Step



Designing Process Conceptual Data Model



Designing Process Application Work Flow

Use-Case I:

- A User opens a Profile

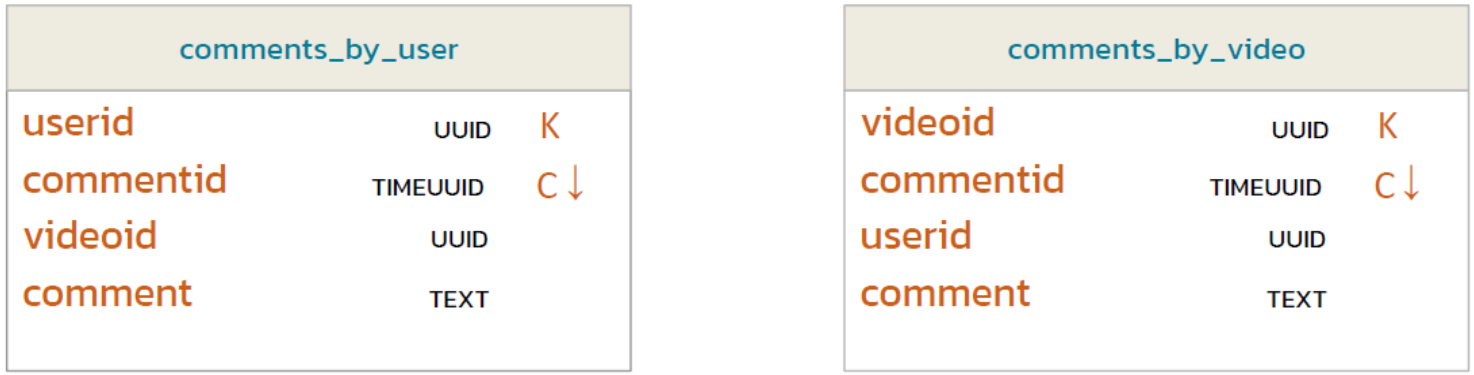
WF2: Find comments related to target user using its identifier, get most recent first

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

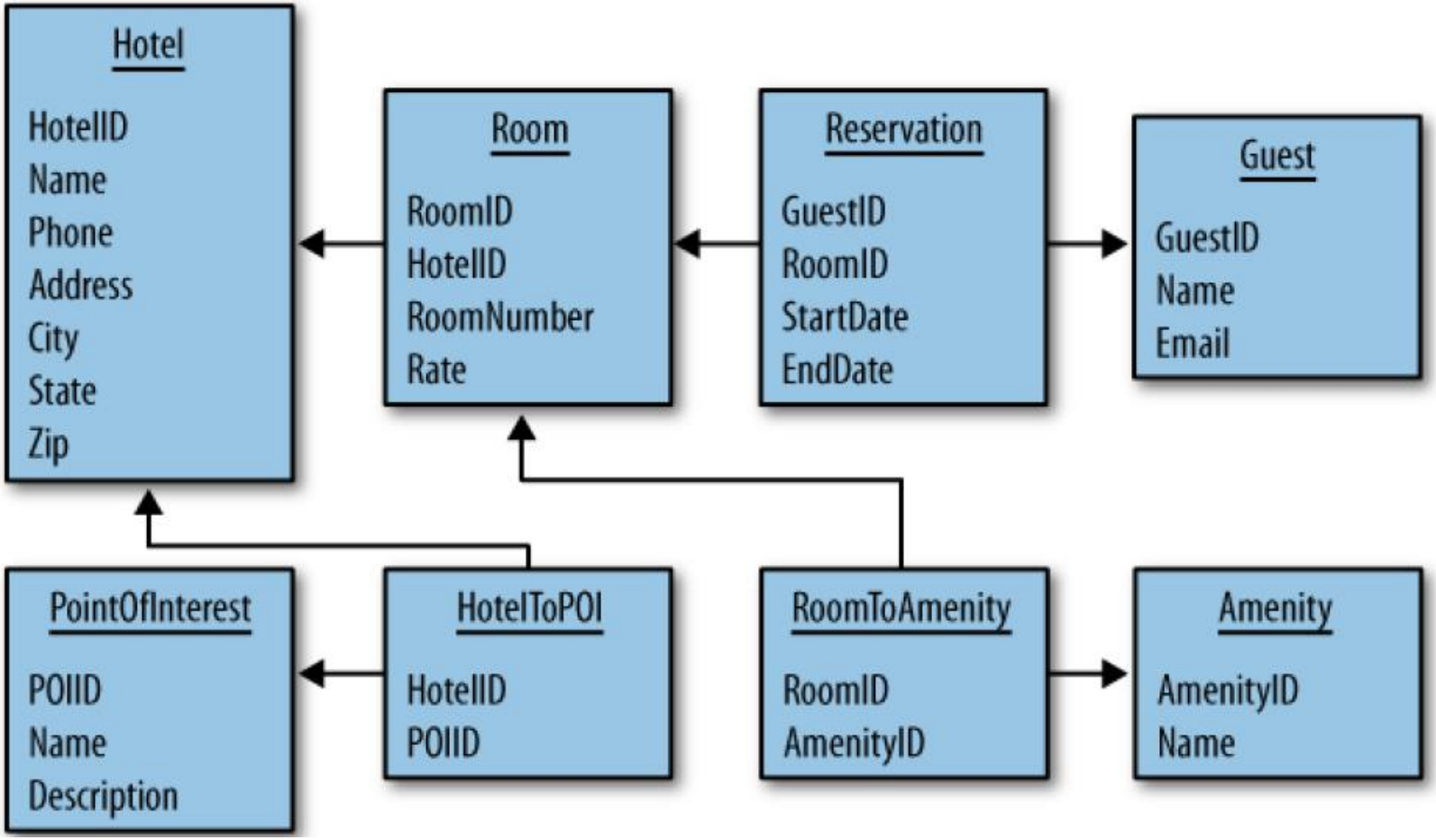


Designing Process Schema DDL

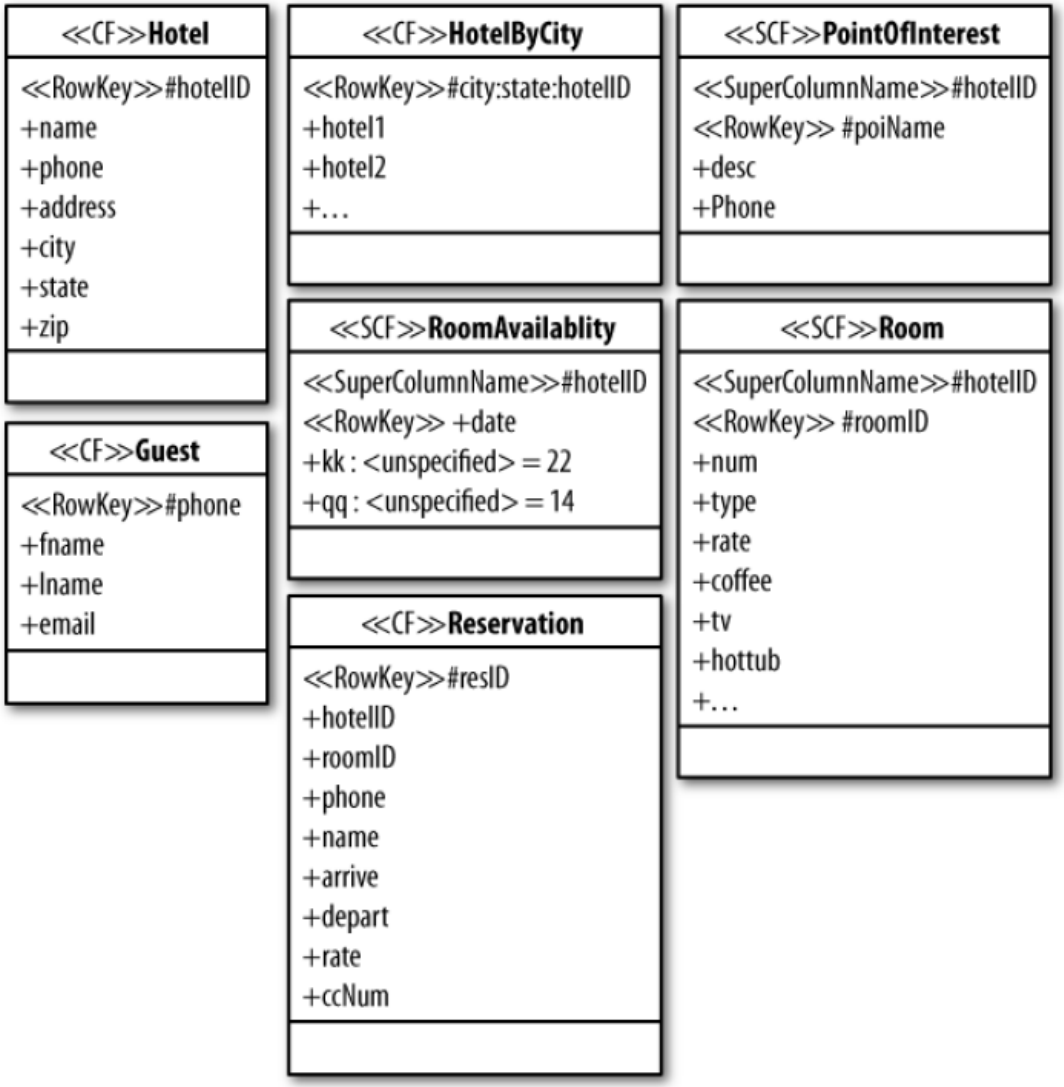
```
CREATE TABLE IF NOT EXISTS comments_by_user (
    userid uuid,
    commentid timeuuid,
    videoid uuid,
    comment text,
    PRIMARY KEY ((userid), commentid)
) WITH CLUSTERING ORDER BY (commentid DESC);

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


Hotel
HotelID
Name
Phone
Address
City
State
Zip
Room
RoomID
HotelID
RoomNumber
Rate
Reservation
GuestID
RoomID
StartDate
EndDate
Guest
GuestID
Name
Email
PointOfInterest
POIID
Name
Description
HotelToPOI
HotelID
POIID
RoomToAmenity
RoomID
AmenityID
Amenity
AmenityID
Name

Use Case


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

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.

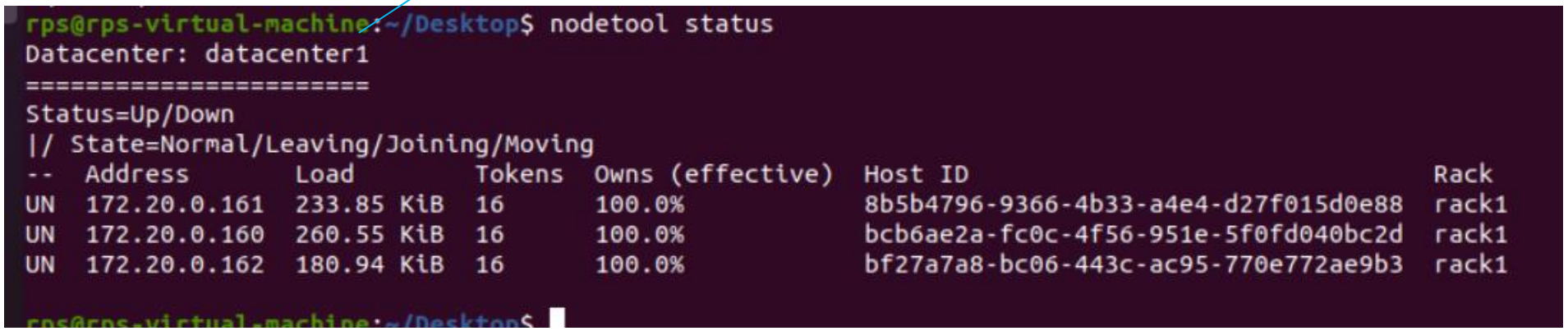
Different Components of Keyspaces

- 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 vebspace 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



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 mutliple 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 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
- https://docs.datastax.com/en/cql-oss/3.3/cql/cql_reference/cqlCreateTable.html#cqlCreateTable