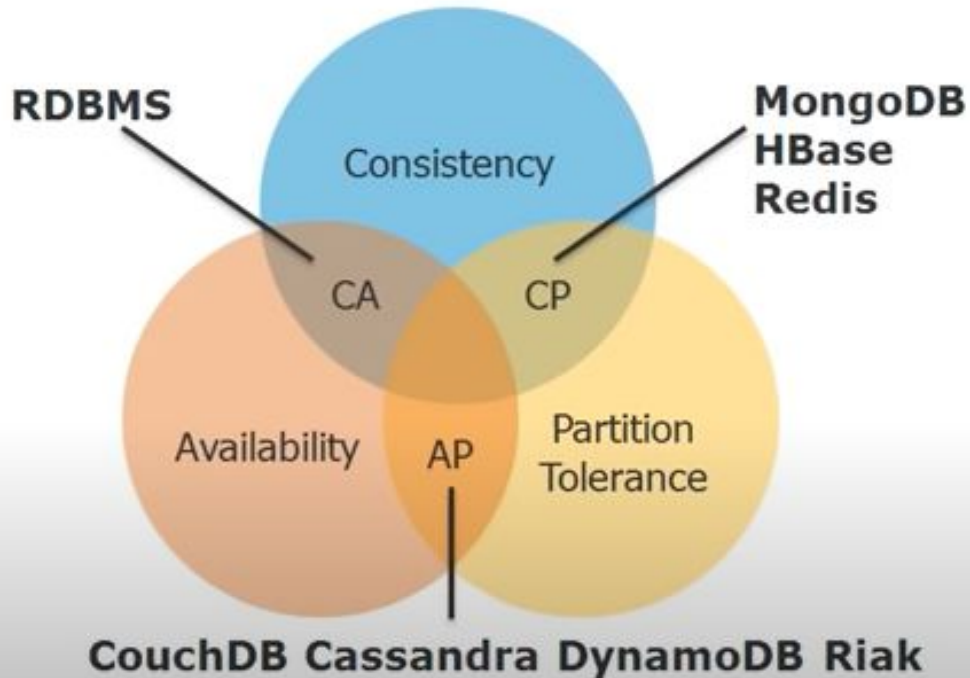


Cassandra

Prepared By : Prof. Shital Pathar

Prepared For :- DDU CE Semester-7

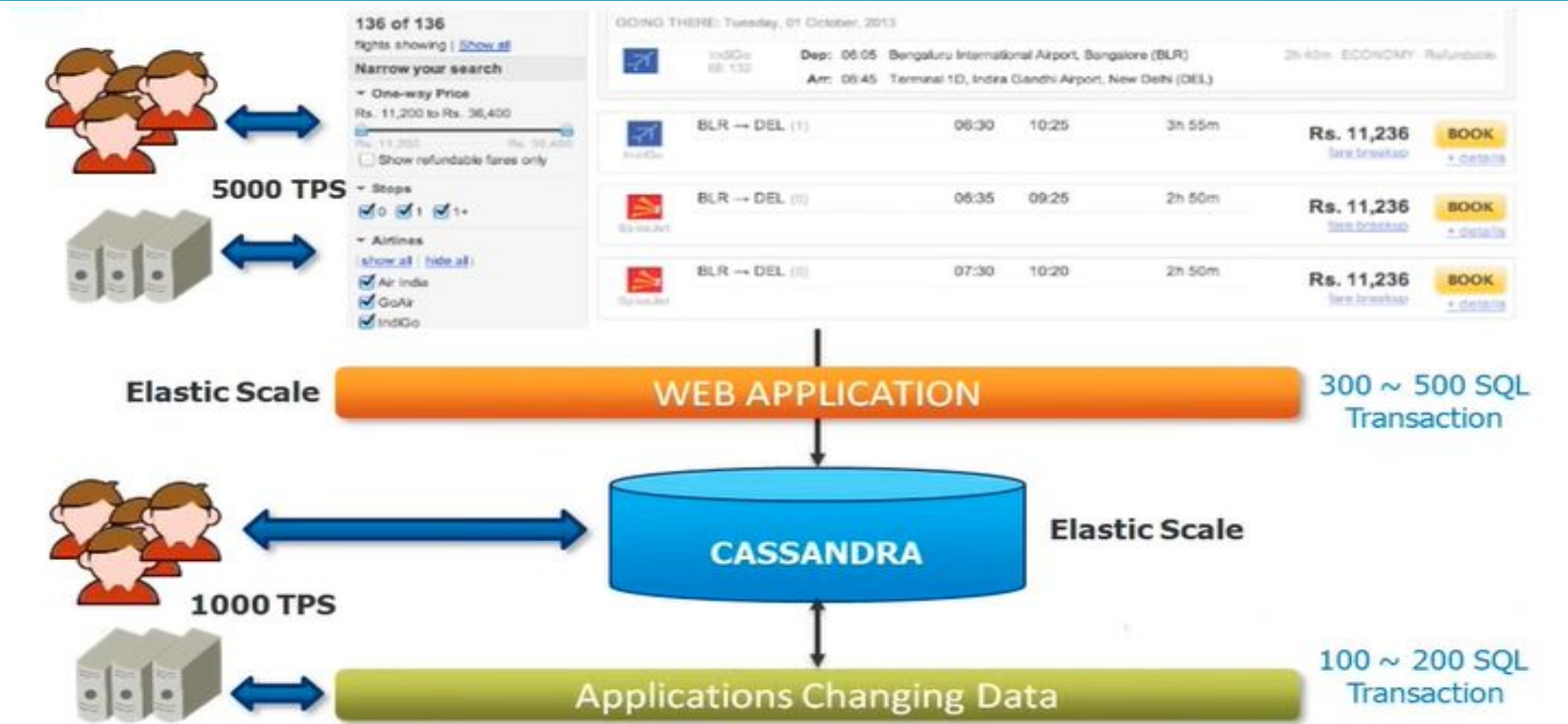
Brewers CAP Theorem



Why Cassandra and not RDBMS?



Why Cassandra and not RDBMS?



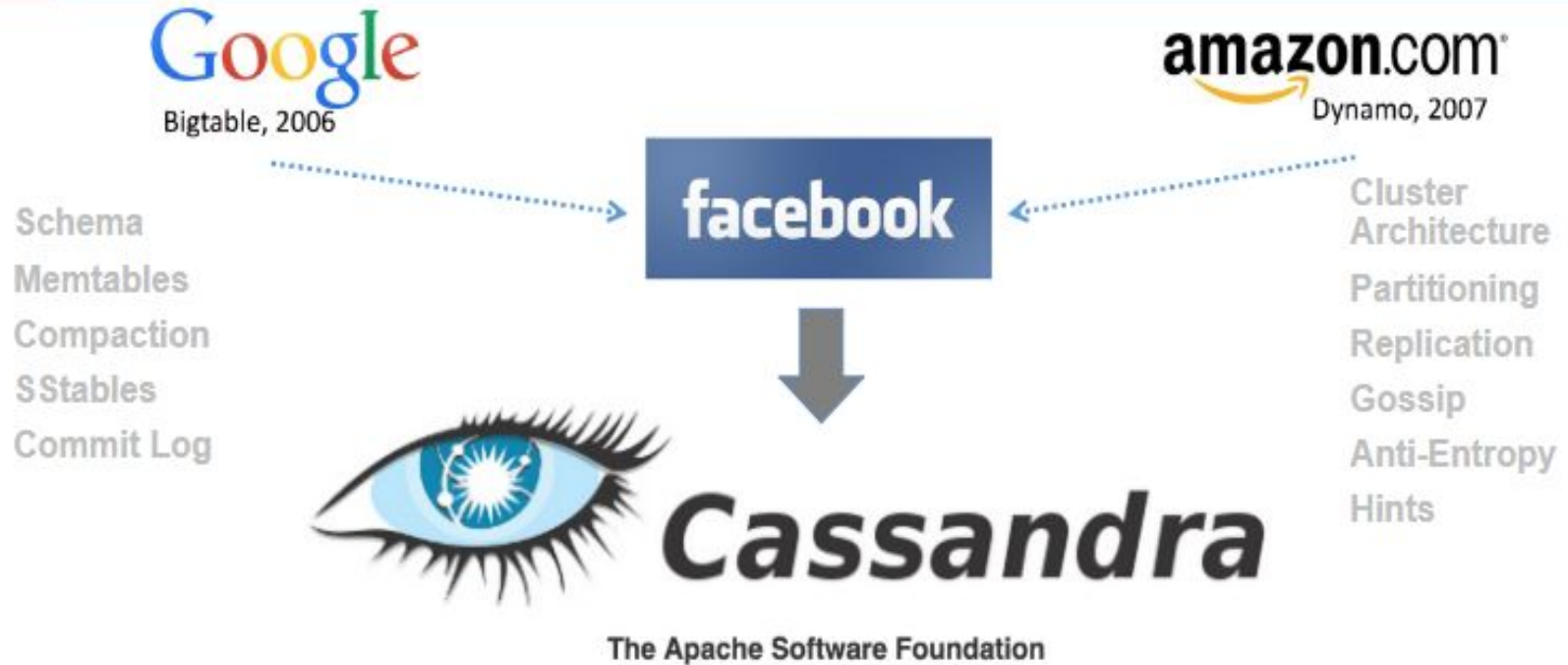
Introduction to Cassandra

- **Apache Cassandra** is a free open source, distributed, column-oriented NoSQL database.
- Cassandra was designed to handle huge amount of data across servers, providing high availability, without a single point of failure.
- COLUMN-ORIENTED – Represents a 2D table, of columns and rows. Data is stored against a row key and a column key, can call it as a cell.



History of Cassandra

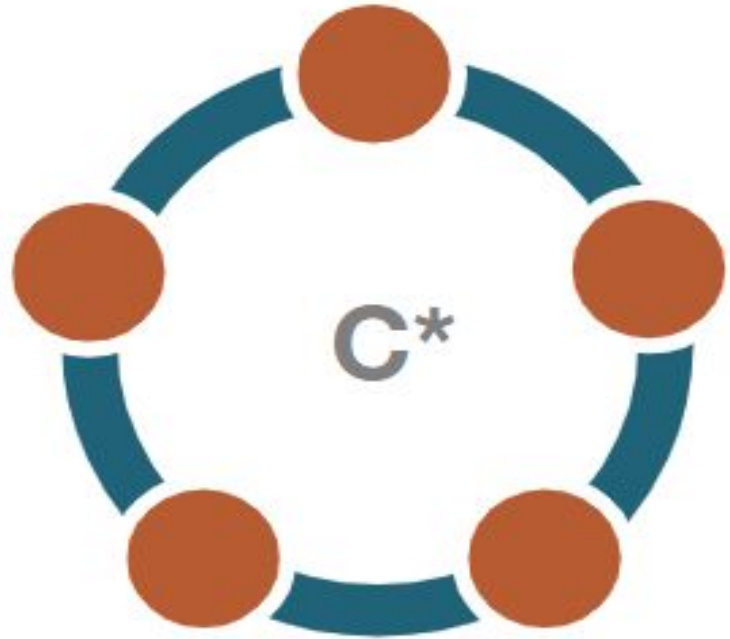
5



What is Cassandra?

Distributed Database

- ❑ Individual nodes
- ❑ Working in a cluster



Who uses Cassandra?

Google

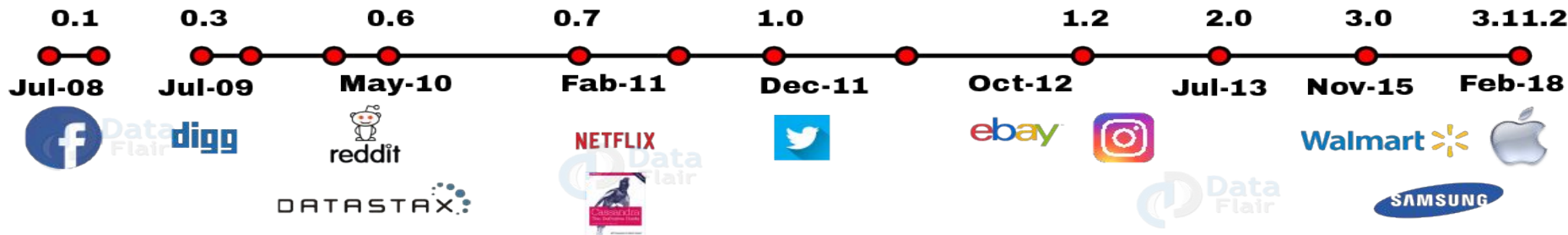
Bigtable, 2006

amazon.com

Dynamo, 2007

facebook.

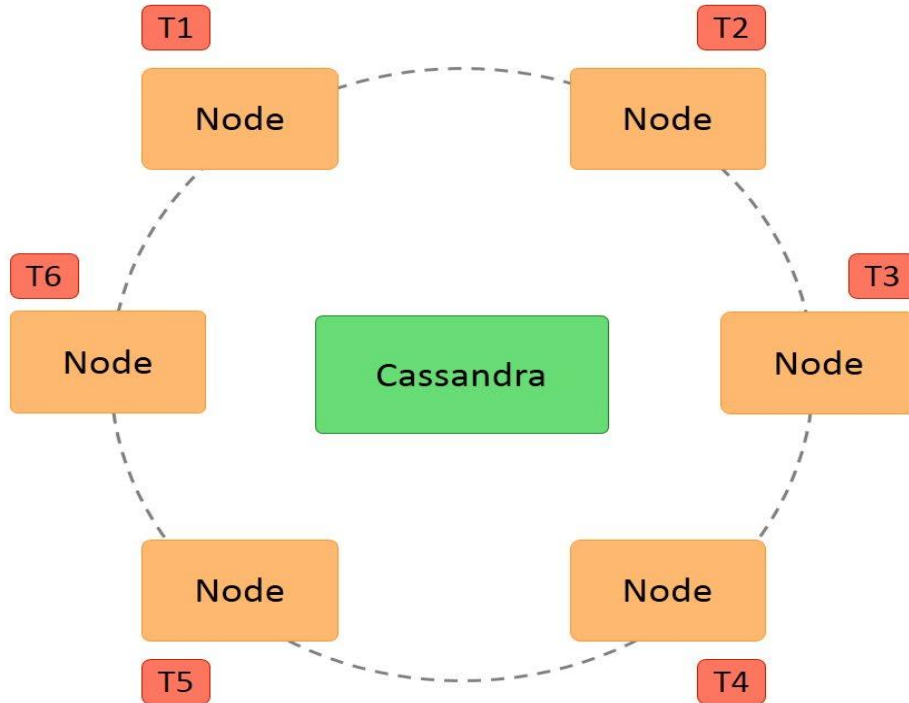
OpenSource, 2008



Design Objectives of Cassandra

- Full **multi-master** database replication
- **Global availability** at low latency
- **Scaling** out on commodity hardware
- Linear **throughput increase** with each additional processor
- Online **load balancing** and cluster growth
- **Partitioned** key-oriented queries
- Flexible **schema**

ARCHITECTURE OF CASSANDRA

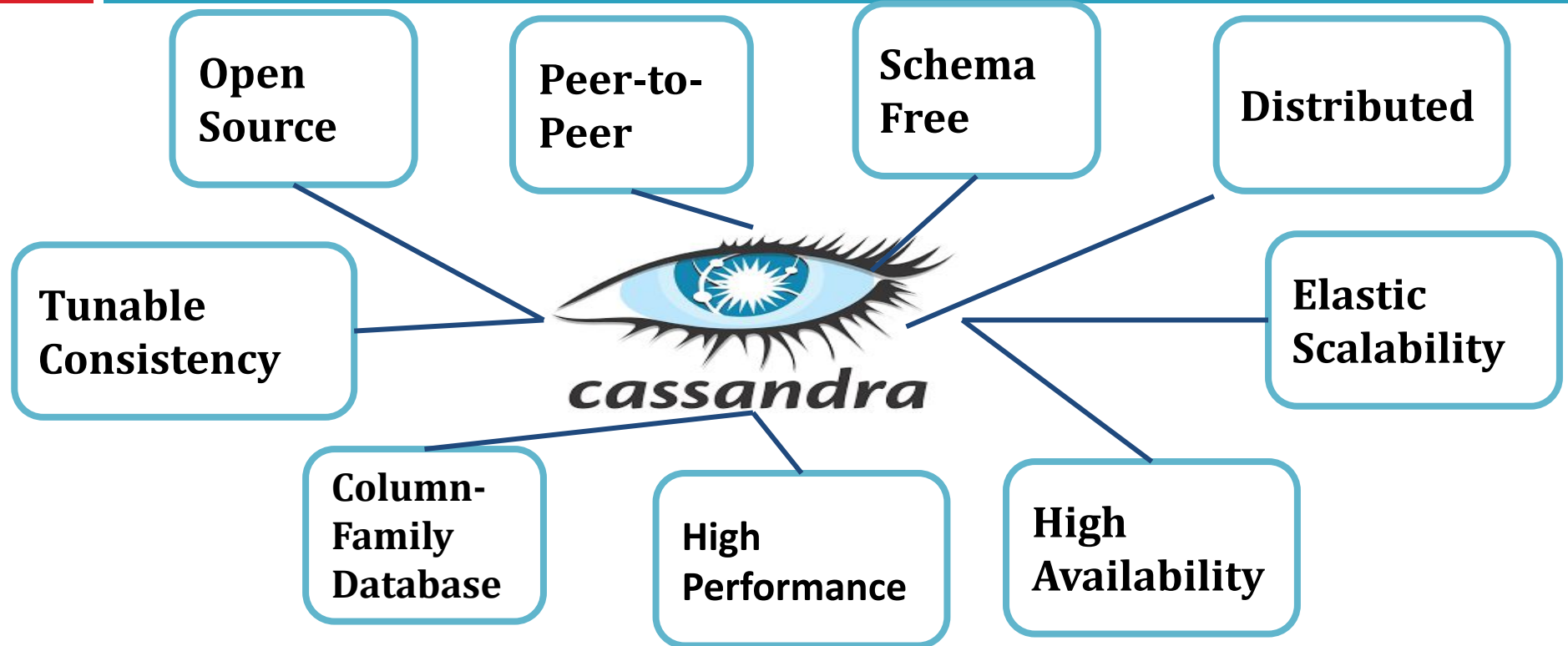


ARCHITECTURE OF CASSANDRA



- 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.

Features of Cassandra

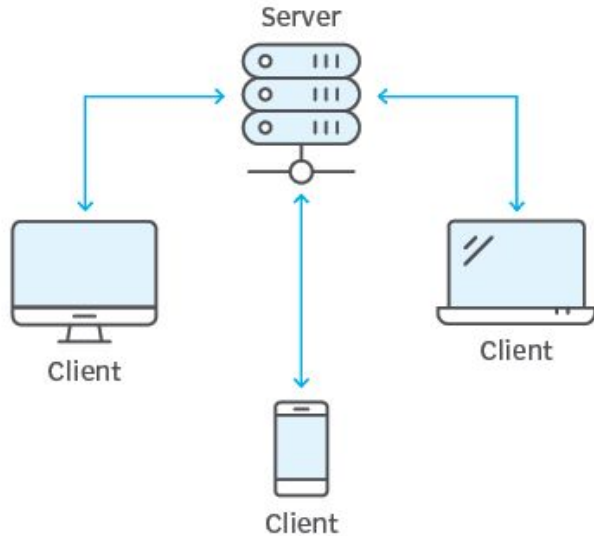


Open Source

- Cassandra, though it is very powerful and reliable, is FREE!
- It is an open source project by Apache.
- Because of the open source feature, it gave birth to a huge Cassandra Community, where people discuss their queries and views.
- Possibility of integrating Cassandra with other Apache Open-source projects like Hadoop, Apache Hive , Apache pig etc.

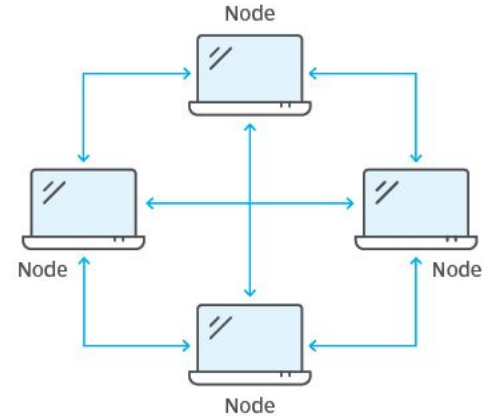
Peer-to-Peer Architecture

Client Server



→ server is having privileges to receive ,
clients are having privileges for sending.

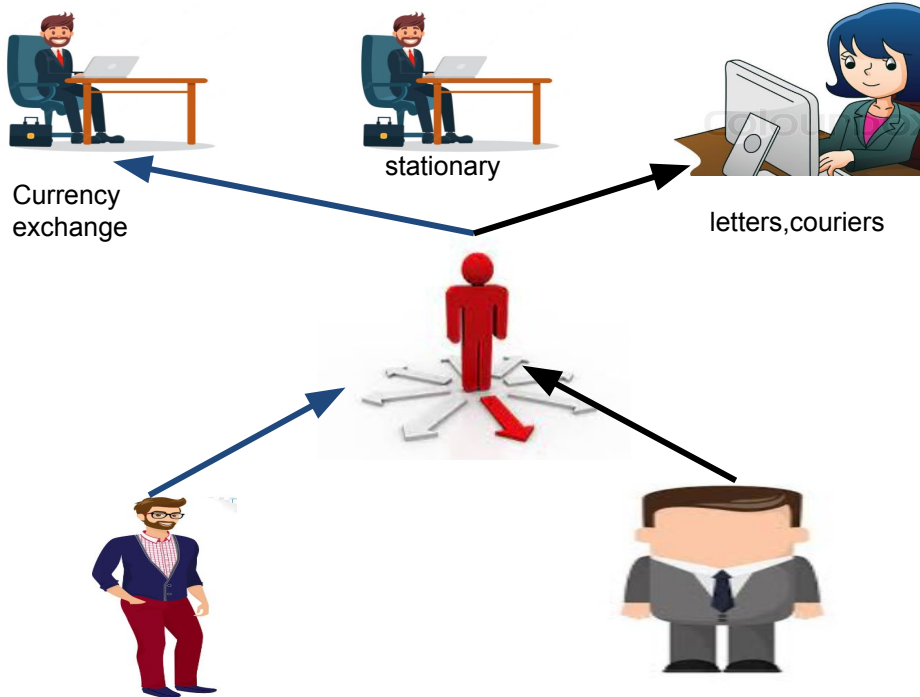
peer to peer



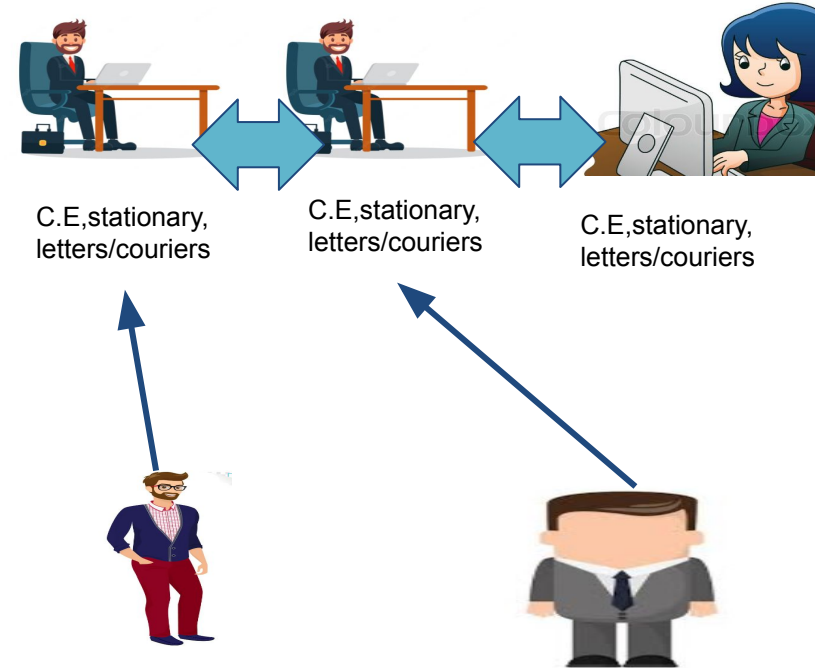
→ all hosts are equally privileged

Distributed

Centralized



Distributed



Post Office Scenario

Distributed



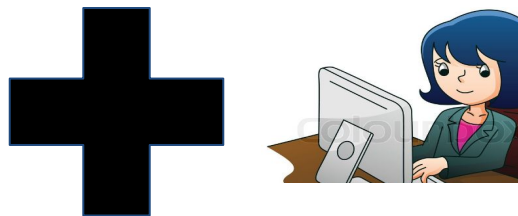
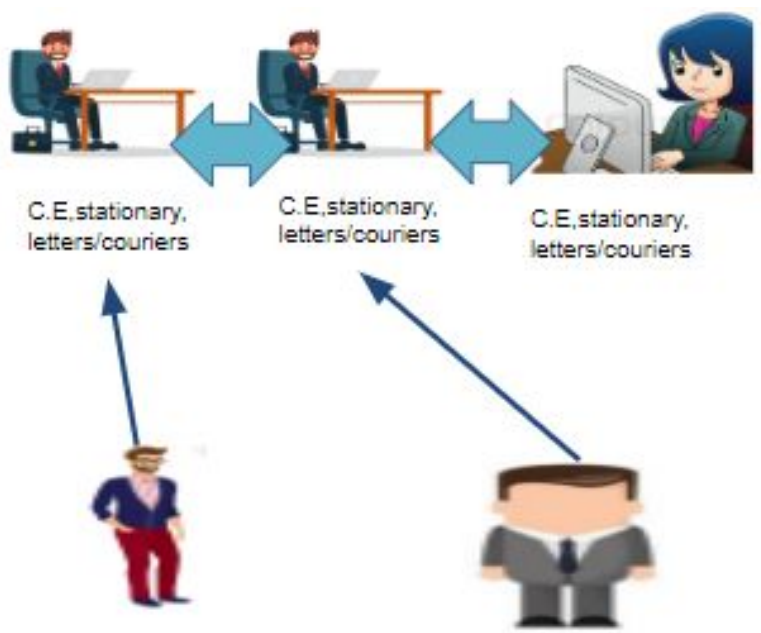
- Every node is identical
- No single point of failure
- No special host to coordinate the activities.
- Peer to peer protocol and gossip protocol
- easy to operate and maintain because all nodes are identical

Types of Scalability



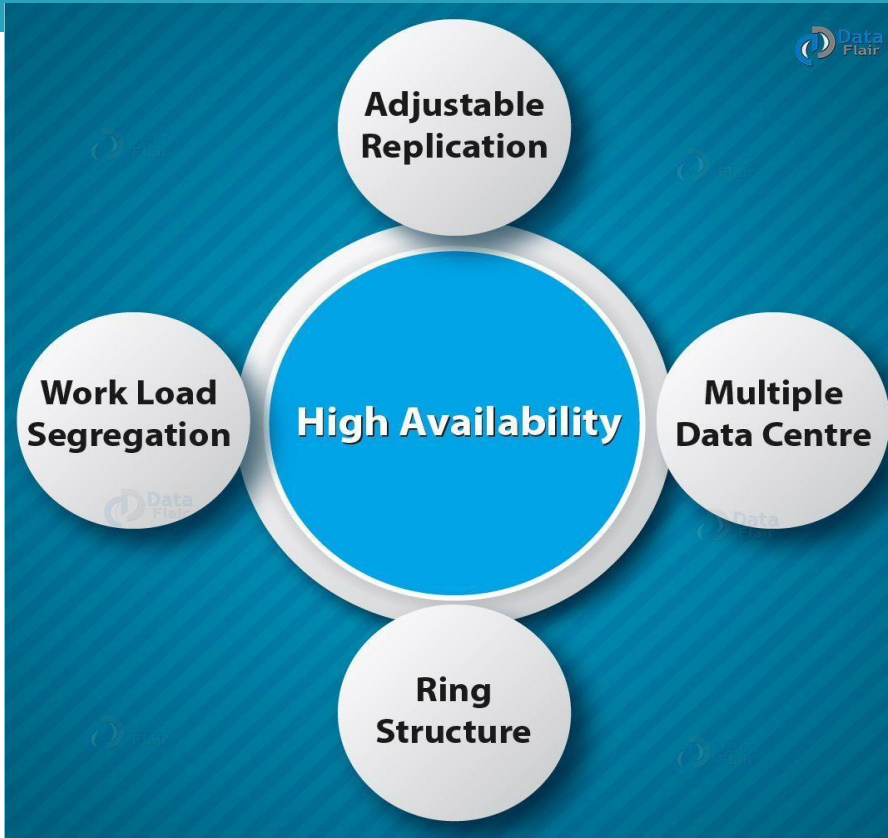
Horizontal Scalability
vertical scalability

Elastic Scalability



- easily **scale-up or scale-down** the cluster in Cassandra.
- Flexibility for **adding or deleting any number of nodes** from the cluster without disturbances - no need of restarting the cluster while scaling up or scaling down.
- **very high throughput** for the highest number of nodes.
- Moreover, there is **zero downtime** .

High Availability and Fault Tolerance



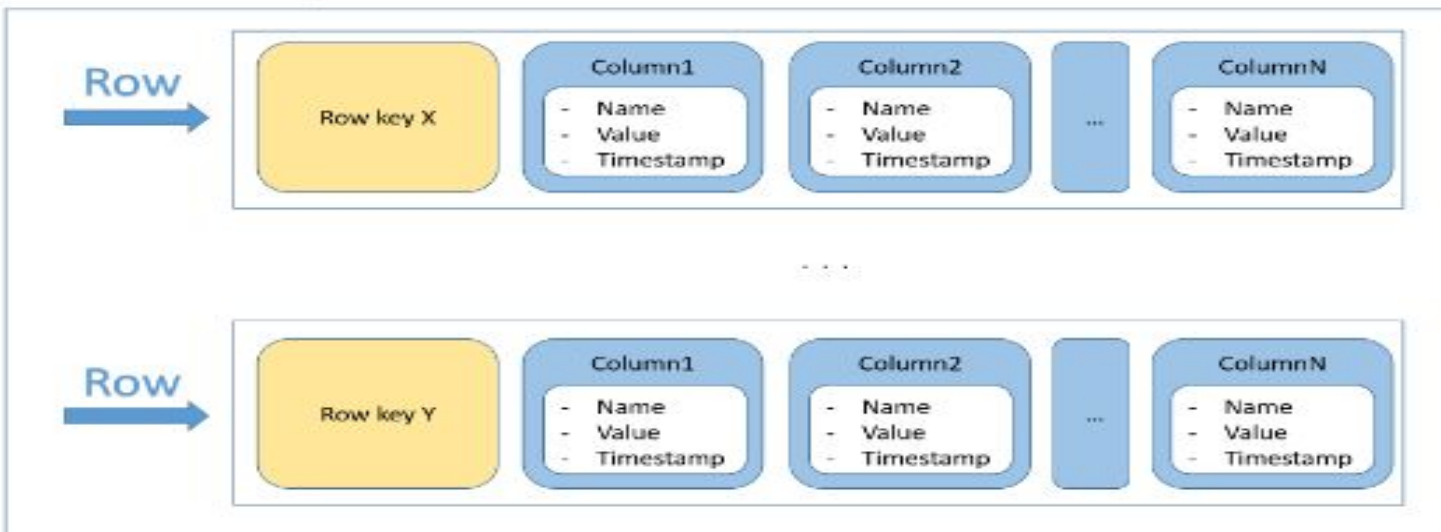
Replication Factor

- Determines no. of copies of data across the cluster
- Ideally, it should be more than one and less than the no. of nodes in cluster
- Two replication strategies:
 - SimpleStrategy
 - NetworkTopologyStrategy

High Performance

- best performance as compared to other NoSQL database.
- Developers wanted to utilize the capabilities of many multi-core machines. This is the base of development of Cassandra.
- Cassandra has proven itself to be excellently reliable when it comes to a large set of data.
- Therefore, Cassandra is used by a lot of organizations which deal with huge amount of data on a daily basis. Furthermore, they are ensured about the data, as they cannot afford to lose the data.

Column-Family Database



- In Cassandra, columns are stored based on column names. We use row key and column family name to address a column family.

Example

512	first name Carol	last name Harper	email carol@datazy.com	address 123 Sesame St, NY
513	first name Benjamin	last name Lieberman	email ben@datazy.com	
514	first name Peter	last name Stallings	email peter@datazy.com	date of birth 07/04/1984

Employee Table

Id	Name	Age	Gender	Car
1	{Name : Brian}	{Age : 21}	{Gender : M}	{Car : BMW}
2	{Name : John}	{Age : 43}	{Gender : M}	{Car : BMW}
3	{Name : Bob}	{Age : 45}	{Gender : M}	{Car : BMW}
4	{Name : Frank}	{Age : 23}	{Gender : M}	{Car : Audi}
5	{Name : Olivia}	{Age : 35}	{Gender : F}	{Car : Audi}
6	{Name : Emma}	{Age : 32}	{Gender : F}	{Car : Audi}
7	{Name : Sophia}	{Age : 45}	{Gender : F}	
8	{Name : Mia}	{Age : 23}	{Gender : F}	

Id	Name	Age	Gender	Car
1	{Name : Brian}	{Age : 21}	{Gender : M} * 4	{Car : BMW}
2	{Name : John}	{Age : 43}		{Car : BMW}
3	{Name : Bob}	{Age : 45}		{Car : BMW}
4	{Name : Frank}	{Age : 23}		{Car : Audi}
5	{Name : Olivia}	{Age : 35}	{Gender : F} * 4	{Car : Audi}
6	{Name : Emma}	{Age : 32}		{Car : Audi}
7	{Name : Sophia}	{Age : 45}		
8	{Name : Mia}	{Age : 23}		



Select first_name from emp where ssn=666

Row Oriented Database

rowid	id	first_name	last_name	ssn	salary	dob	title	joined
1001	1	John	Smith	111	101,000	1/1/1991	eng	1/1/2011
1002	2	Kary	White	222	102,000	2/2/1992	mgr	2/1/2012
1003	3	Norman	Freeman	333	103,000	3/3/1993	mkt	3/1/2013
1004	4	Nole	Smith	444	104,000	4/4/1994	adm	4/1/2014
1005	5	Dar	Sol	555	105,000	5/5/1995	adm	5/1/2015
1006	6	Yan	Thee	666	106,000	6/6/1996	mkt	6/1/2016
1007	7	Hasan	Ali	777	107,000	7/7/1997	acc	7/1/2017
1008	8	Ali	Bilal	888	108,000	8/8/1998	acc	8/1/2018

Logical Storage

1001, 1, John, Smith, 111, 101,000, 1/1/1991, eng, 1/1/2011 ||||
1002, 2, Kary, White, 222, 102,000, 2/2/1992, mgr, 2/1/2012

1003, 3, Norman, Freeman, 333, 103,000, 3/3/1993, mkt, 3/1/2013 ||||
1004, 4, Nole, Smith, 444, 104,000, 4/4/1994, adm, 4/1/2014

1005, 5, Dar, Sol, 555, 105,000, 5/5/1995, adm, 5/1/2015 ||||
1006, 6, Yan, Thee, 666, 106,000, 6/6/1996, mkt, 6/1/2016

1007, 7, Hasan, Ali, 777, 107,000, 7/7/1997, acc, 7/1/2017 ||||
1008, 8, Ali, Bilal, 888, 108,000, 8/8/1998, acc, 8/1/2018

Column Oriented

1:1001, 2:1002, 3:1003, 4:1004, 5:1005, 6:1006, 7:1007, 8:1008

John:1001, Kary:1002, Norman:1003, Nole:1004, Dar:1005, Yan:1006, Hasan:1007, Ali:1008

Smith:1001, White:1002, Freeman:1003, Sol:1004 Thee:1005, Ali:1006, Bilal:1007, Ali:1008

111:1001, 222:1002, 333:1003, 444:1004, 555:1005, 666:1006, 777:1007, 888:1008

101000:1001, 102000:1002, 103000:1003, 104000:1004, 105000:1005, 106000:1006, 107000:1007, 108000:1008

1/1/1991:1001, 2/2/1992:1002, 3/3/1993:1003, 4/4/1994:1004, 5/5/1995:1005, 6/6/1996:1006, 7/7/1997:1007, 8/8/1998:1008

eng:1001, mgr:1002, mkt:1003, adm:1004, adm:1005, mkt:1006, acc:1007, acc:1008

1/1/2011:1001, 2/1/2012:1002, 3/1/2013:1003, 4/1/2014:1004, 5/1/2015:1005, 6/1/2016:1006, 7/1/2017:1007, 8/1/2018:1008



Select sum(salary) from emp

Schema-Free

- There is a flexibility in Cassandra to create columns within the rows. That is, Cassandra is known as the schema-optional data model.
- Since each row may not have the same set of columns, there is no need to show all the columns needed by the application at the surface.
- Therefore, Schema-less/Schema-free database in a column family is one of the most important Cassandra features.

Cassandra Consistency level

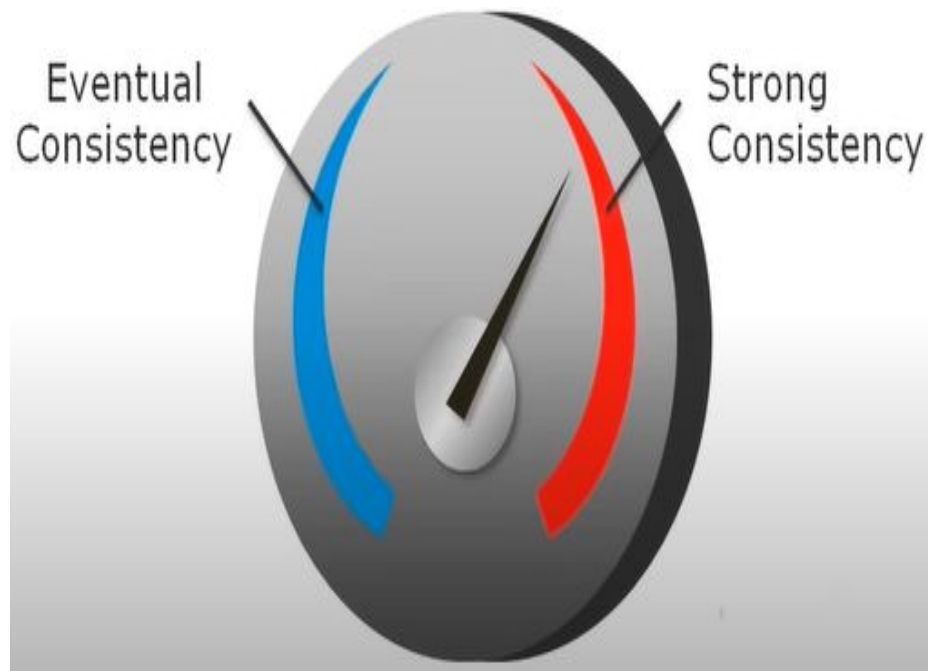


- The **Cassandra consistency level** is defined as the minimum number of **Cassandra** nodes that must acknowledge a read or write operation before the operation can be considered successful.
- How many nodes will be read before responding to the client is based on the consistency level specified by the client. If client-specified consistency level is not met, there is possibility that few nodes may respond with out-of-date copies, in which case read operation blocks.

Tunable Consistency

Cassandra enables us to tune the consistency based on the application requirements.

- **Eventual consistency** makes sure that the client is acknowledged as soon as a part of the cluster acknowledges the write. (Used when performance matters)
- Whereas, **Strong consistency** make sure that any update is broadcasted to all the nodes or machines where the particular data is suited.
- Cassandra can cash in on any of them depending on requirements.



DataBase Model(Top to down)

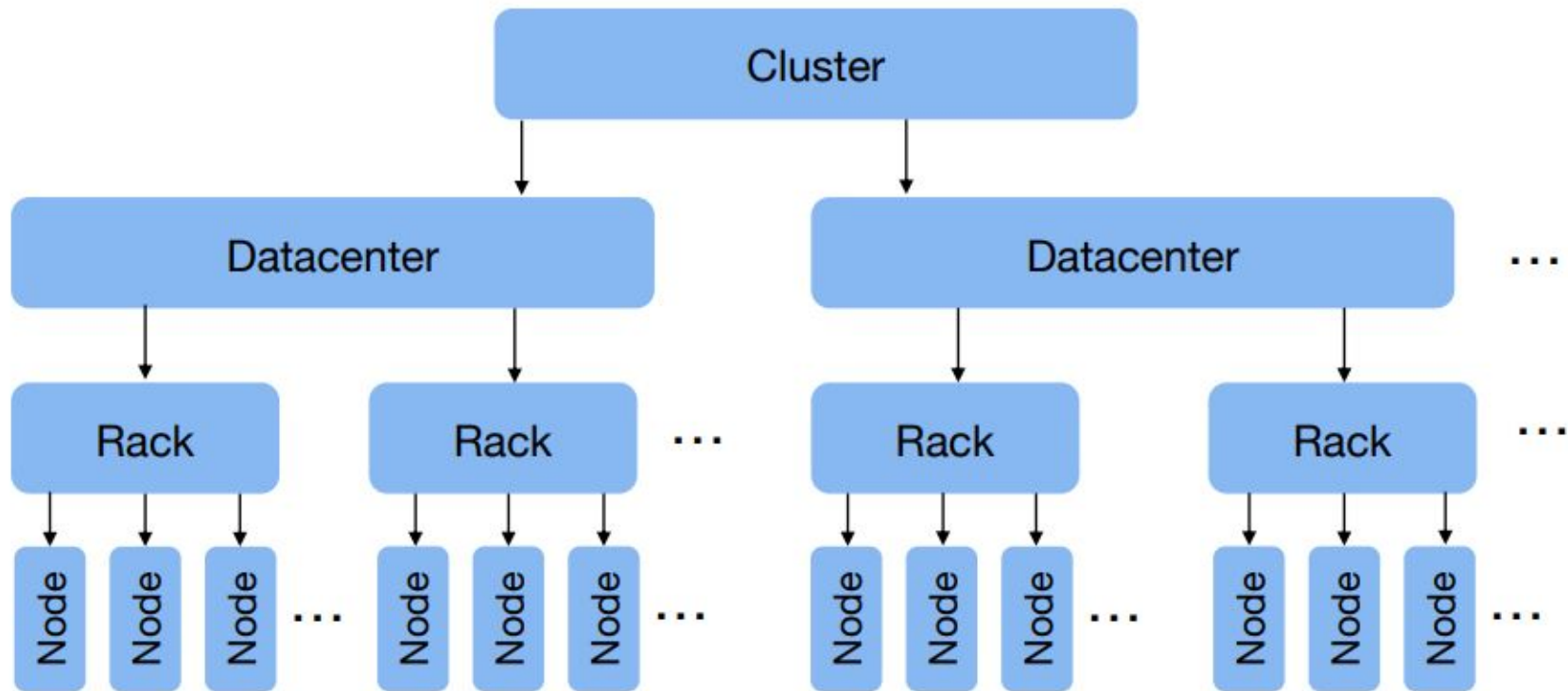
RDBMS	Cassandra
Database Server	Cluster
Database	Keyspace
Table	Column Family
Rows and Columns	Rows and Columns
Primary Key	
Indexes	Indexes (Primary & Secondary)
Stored Procedure / Functions	-
Trigger	*
Inbuilt Functions	-
SQL	CQL
JOINS / UNION / SUB QUERIES	-
Sorting	*
Grouping	-
Commit / Rollback	-
-	Collections
Partitions	Partitions

Design goals of cassandra



- High availability – no single point of failure -
- No central coordinator
- Low latency
- Run on commodity hardware
- Linear performance increase with additional nodes
- Tunable consistency: Define replication and policy
- Key-oriented queries
- Flexible data model: row can have different columns with different data types
- SQL-like query language (CQL - Cassandra Query Language)

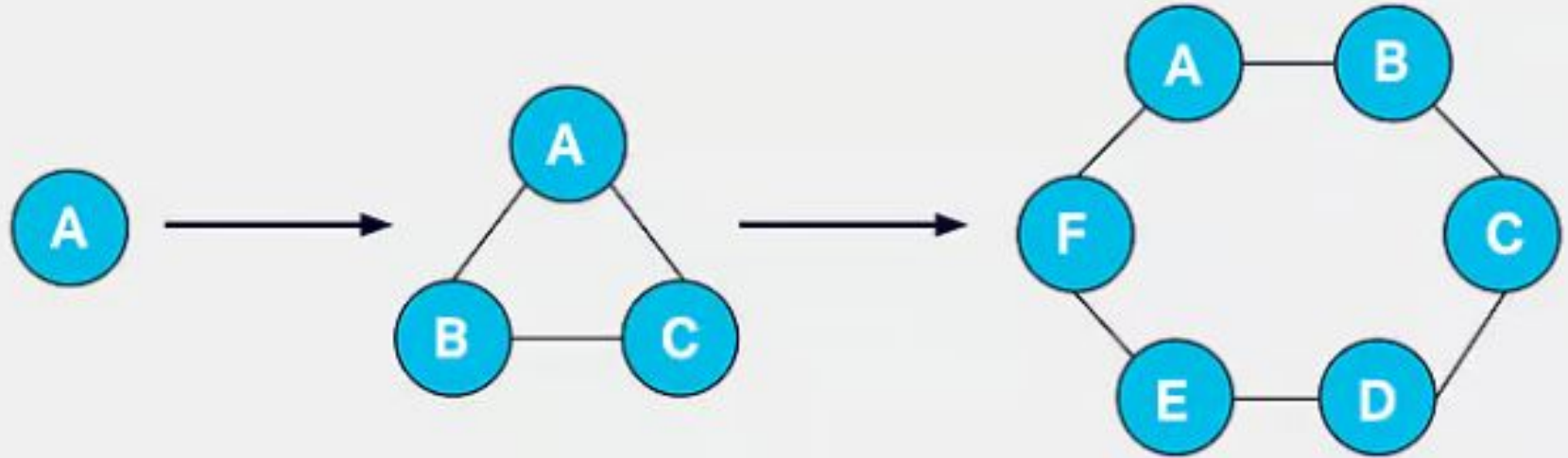
Architecture of Cassandra



Terms in Cassandra

- A **Cluster** is a collection of Data Centers.
- A **Data Center** is a collection of Racks.
- A **Rack** is a collection of Servers.
- A **Server** contains 256 virtual nodes (or vnodes) by default.
- A **vnode** is the data storage layer within a server. Node is the place where data is stored. It is the basic component of Cassandra.

Cluster of Cassandra

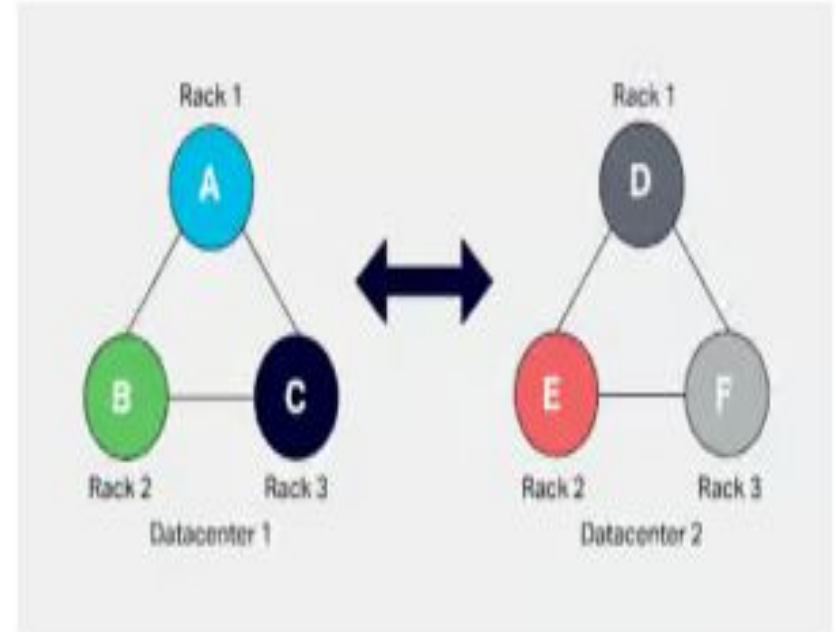
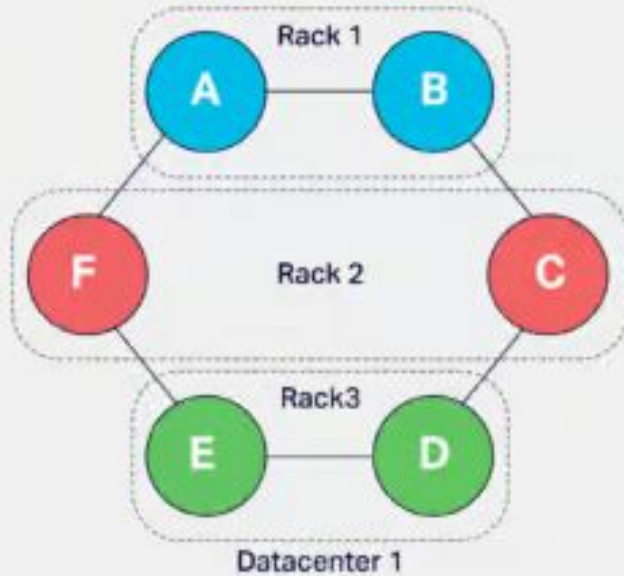



Cluster of Cassandra



- Cassandra works with **peer to peer architecture**, with each node connected to all other nodes.
- **Each Cassandra node performs all database operations** and can serve client requests without the need for a master node.
- A Cassandra cluster does not have a single point of failure as a result of the peer-to-peer distributed architecture.

Rack and Datacenter



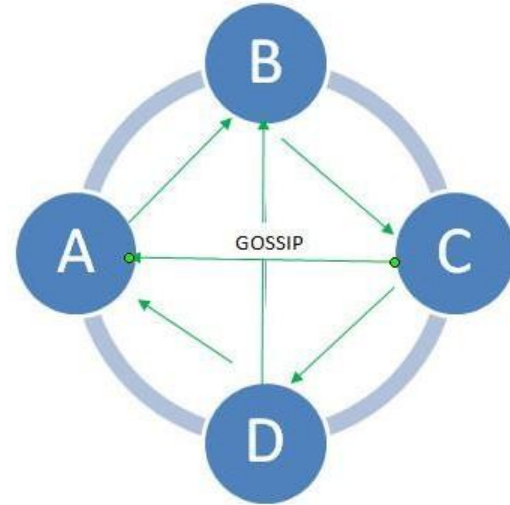
- 
- A cluster is subdivided into racks and data centers.
 - These terminologies are Cassandra's representation of a real-world rack and data center.
 - A physical rack is a group of bare-metal servers sharing resources like a network switch, power supply etc.
 - In Cassandra, the nodes can be grouped in racks and data centers with snitch configuration.
 - Ideally, the node placement should follow the node placement in actual data centers and racks. Data replication and placement depends on the rack and data center configuration.

Gossip Protocol

- Gossip is the protocol used by Cassandra nodes for peer-to-peer communication.
- The gossip informs a node about the state of all other nodes.
- A node performs gossip with up to three other nodes every second.
- The gossip messages follow specific format and version numbers to make efficient communication.

Gossip and Failure Detection

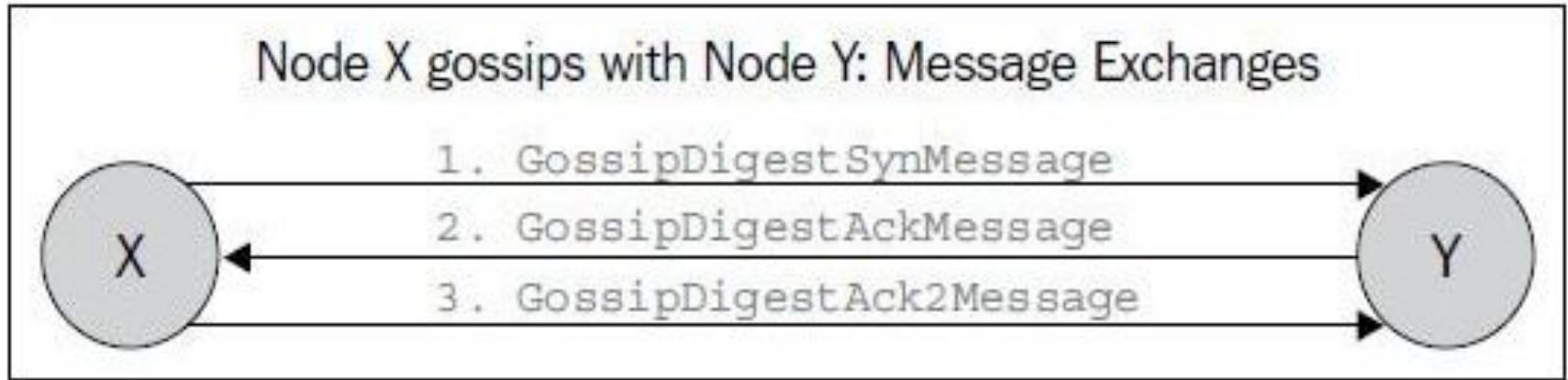
- Gossip is used for intra-ring communication
- The gossip process runs every second for every node and exchange state messages with up to three other nodes in the cluster.
- Each node independently will always select one to three peers to gossip with. It will always select a live peer (if any) in the cluster.
- Eases the discovery and sharing of location and state information with other nodes in the cluster



PEER TO PEER DISTRIBUTION
MODEL OF CASSANDRA

Gossip messaging

•



Partitioner

- Partitioner
 - takes a call on how to distribute data on the various nodes in a cluster.
 - also determines the node on which to place the very first copy of data
- It is a **hash function** to compute the token of the partition key which helps to identify a row uniquely.
- Both the **Murmur3Partitioner** and **RandomPartitioner** use tokens(it's hash) to help assign equal portions of data to each node and evenly distribute data from all the tables throughout the ring or other grouping, such as a keyspace.

Relational Tables

Que : How many employee of a company drive BMW?

Employee				
ID	FirstName	Surname	CompanyCarId	Salary
1	Elvis	Presley	1	\$30000
2	David	Bowie	2	\$40000
3	Kylie	Jenner	3	\$60000
4	Elton	John	1	\$20000
5	Mariah	Carey	2	\$30000
6	Justin	Bieber	4	\$30000
7	Selena	Gomez	5	\$50000

Company Car				
ID	Make	Model	Cost	Engine
1	BMW	5 Series	\$50000	1.8
2	Audi	A6	\$55000	1.6
3	Mercedes	C-Class	\$60000	1.6
4	Mercedes	A-Class	\$30000	1.4
5	BMW	3 Series	\$35000	1.6

Joins

- Joins – Joining two tables
- Not possible in distributed databases
- Solution – Query First Approach (Queries are reflected in tables)
- Design tables for queries
- May end up having duplicate data in multiple tables, but that is the only efficient approach for distributed databases, also called denormalization

Query First Approach-Data Model

Employee By Car Make				
Make	EmployeeID	Employee Firstname	Employee Surname	Salary
BMW	1	Elvis	Presley	\$30000
BMW	4	Elton	John	\$20000
BMW	7	Selena	Gomez	\$30000
Audi	2	David	Bowie	\$40000
Audi	5	Mariah	Carey	\$30000
Mercedes	3	Kylie	Jenner	\$60000
Mercedes	6	Justin	Bieber	\$50000

Company Car By ID				
ID	Make	Model	Cost	Engine
1	BMW	5 Series	\$50000	1.8
2	Audi	A6	\$55000	1.6
3	Mercedes	C-Class	\$60000	1.6
4	Mercedes	A-Class	\$30000	1.4
5	BMW	3 Series	\$35000	1.6

Cassandra Tables

Partition Keys

Employee By Car Make				
BMW*3	{Id : 1}	{Firstname : Elvis}	{Surname : Presley}	{Salary : \$30000}
	{Id : 4}	{Firstname : Elton}	{Surname : John}	{Salary : \$20000}
	{Id : 7}	{Firstname : Selena}	{Surname : Gomez}	{Salary : \$30000}
Audi*2	{Id : 2}	{Firstname : David}		{Salary : \$40000}
	{Id : 5}	{Firstname : Mariah}	{Surname : Carey}	{Salary : \$30000}
Mercedes*2	{Id : 3}	{Firstname : Kylie}	{Surname : Jenner}	
	{Id : 6}	{Firstname : Justin}	{Surname : Bieber}	{Salary : \$50000}

Company Car By ID				
1	{Make : BMW}	{Model : 5 Series}	{Cost : \$50000}	{Engine : 1.8}
2	{Make : Audi}	{Model : A6}	{Cost : \$55000}	{Engine : 1.6}
3	{Make : Mercedes}	{Model : C-Class}	{Cost : \$60000}	{Engine : 1.6}
4	{Make : Mercedes}	{Model : A-Class}	{Cost : \$30000}	{Engine : 1.4}
5	{Make : BMW}	{Model : 3 Series}	{Cost : \$35000}	{Engine : 1.6}

Partition keys

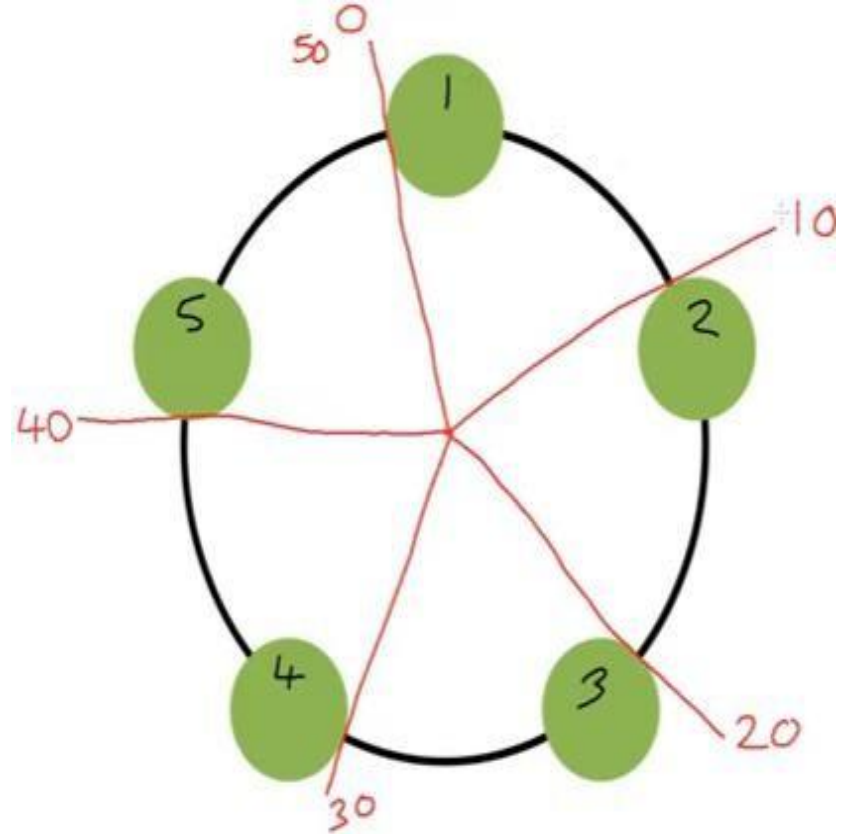
- Data belonging to same partition will be written on same node.
- In cassandra, data is accessed by partition keys and not by primary key though primary key may be part of the table.
- E.g. For Employee Table – Car Make, For Car table – Car ID
- **Hash function is used for creating partitioning**

Example

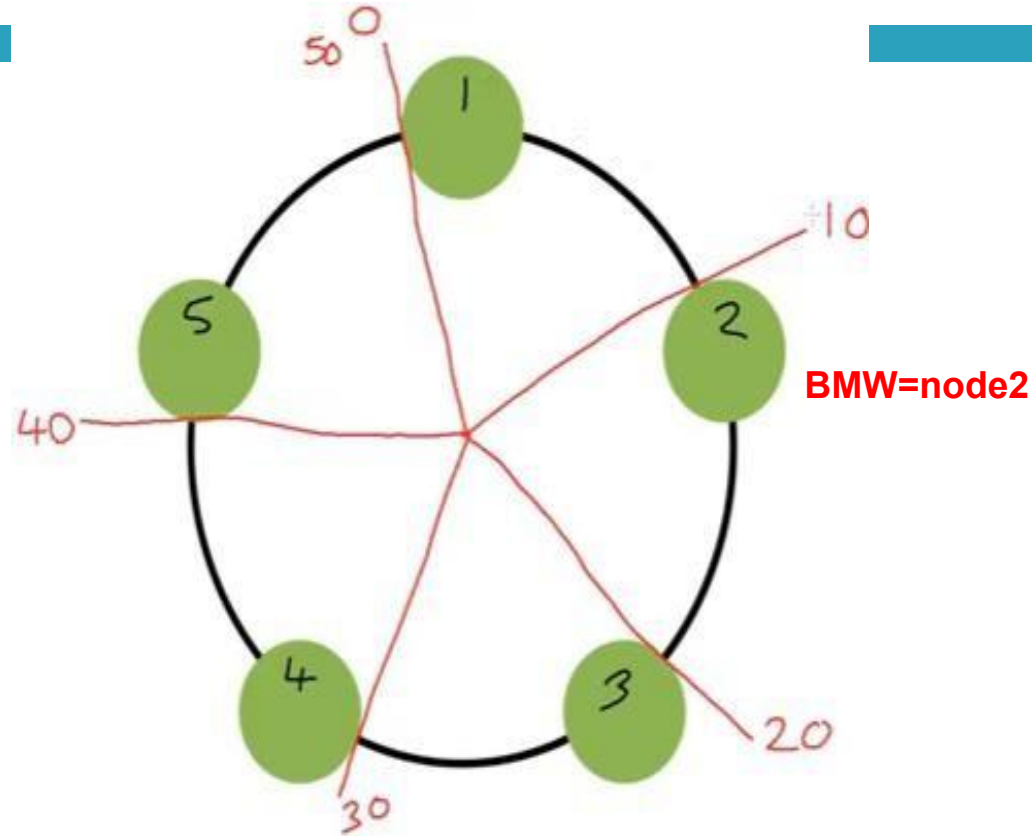
- BMW -> Hash Function -> 15
 - Audi -> Hash Function -> 22 **Tokens**
 - Merc -> Hash Function -> 43
 - BMW -> Hash Function -> 15
-
- Tokens are of 64 bit integers in cassandra.
 - Range: -2^{63} to $2^{63} - 1$

Ring

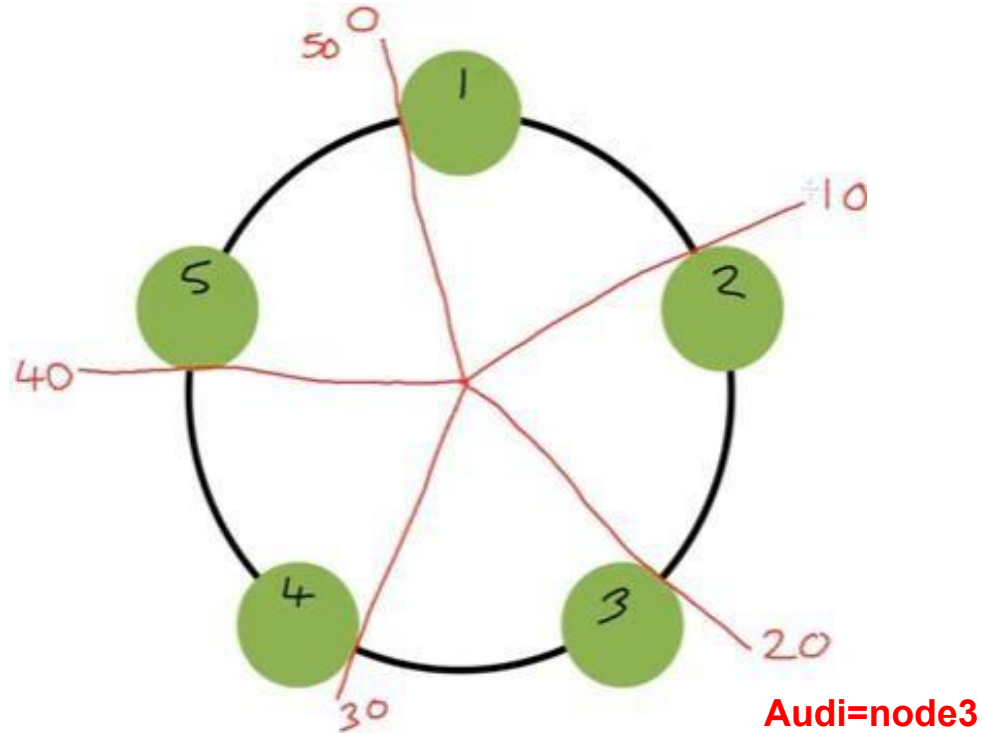
- Ideally, each node is given a token
- In real life, each node can store a range of tokens.
- E.g. node 2 can store tokens >10 and <20 .



BMW -> Hash Function -> 15

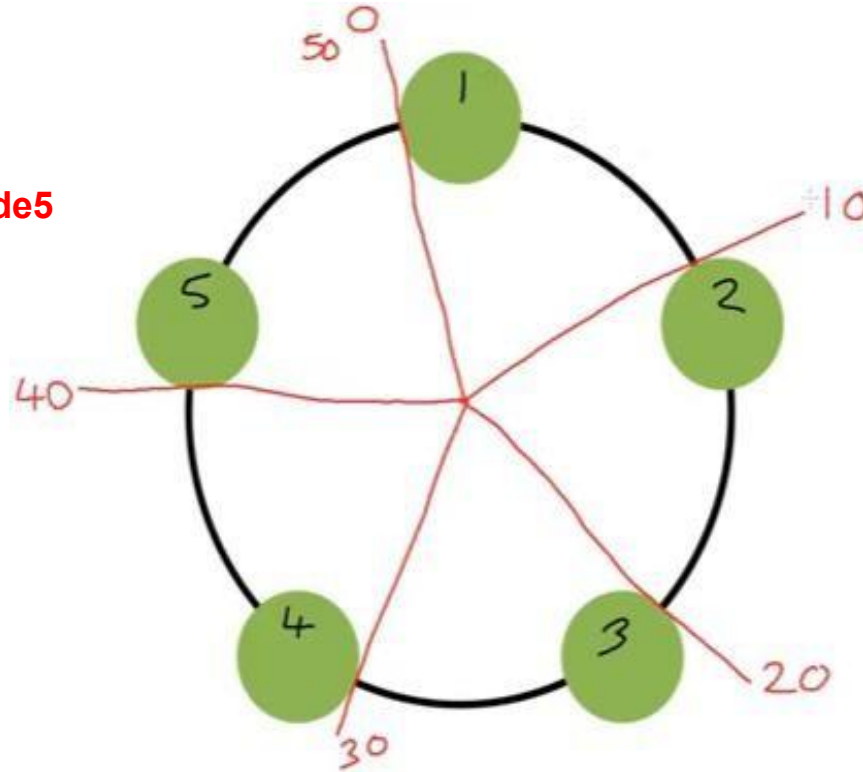


Audi -> Hash Function -> 22



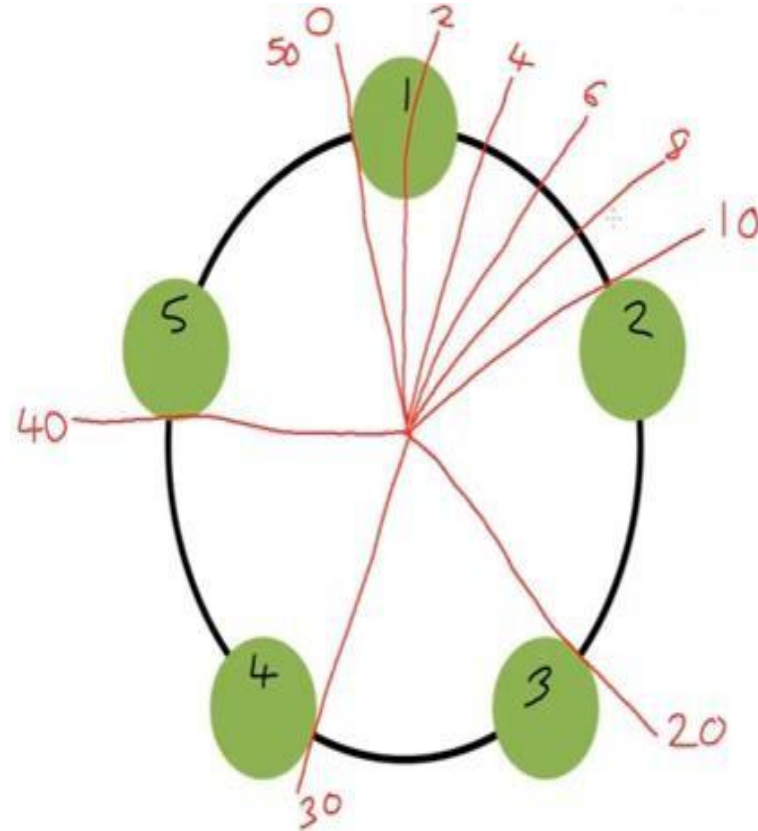
Merc -> Hash Function -> 43

Merc=node5



Tokens

- In real life, a range assigned to a node can be much larger and Cassandra does not want to do this.
- Instead it assigns multiple token ranges to a node. This concept is called virtual nodes.
- Any token in those ranges still go to that single node, but this gives greater flexibility.

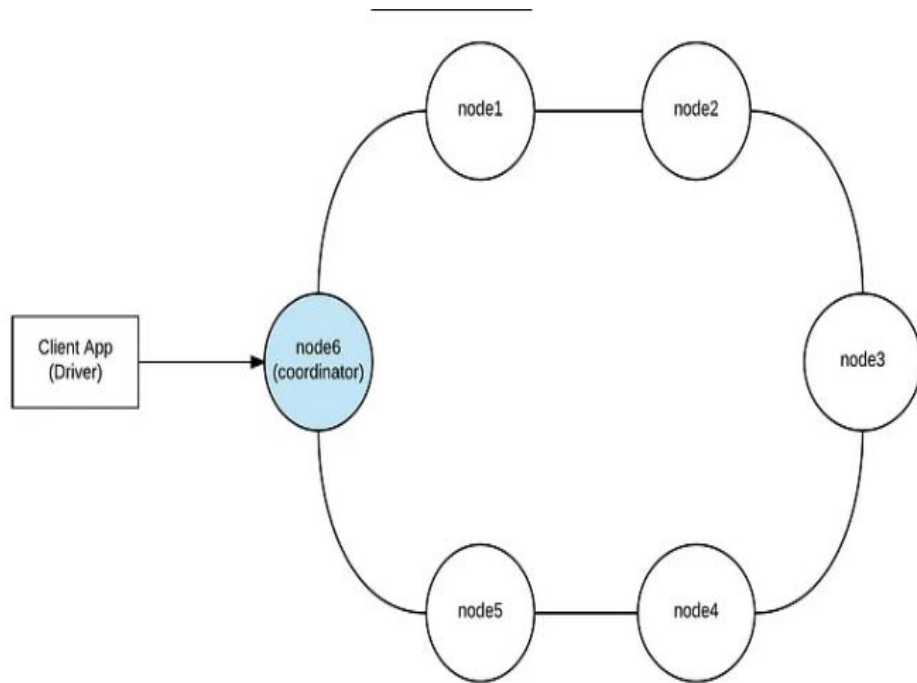


Advantages of Virtual Nodes

- Makes the process of assigning tokens to nodes less manual
- Nodes can be added seamlessly in cassandra
 - Provides more storage
 - Gives high throughput
- Nodes with different capacity can be assigned different number of virtual nodes.
- Default value in cassandra is 256 virtual nodes

Coordinator node

- When a request is sent to any Cassandra node, this node acts as a proxy for the application (actually, the Cassandra driver) and the nodes involved in the request flow. This proxy node is called as the coordinator.
- The coordinator is responsible for managing the entire request path and to respond back to the client.



Data Replication in Cassandra

As hardware problem can occur or link can be down at any time during data process, a solution is required to provide a backup when the problem has occurred. So data is replicated for assuring no single point of failure.

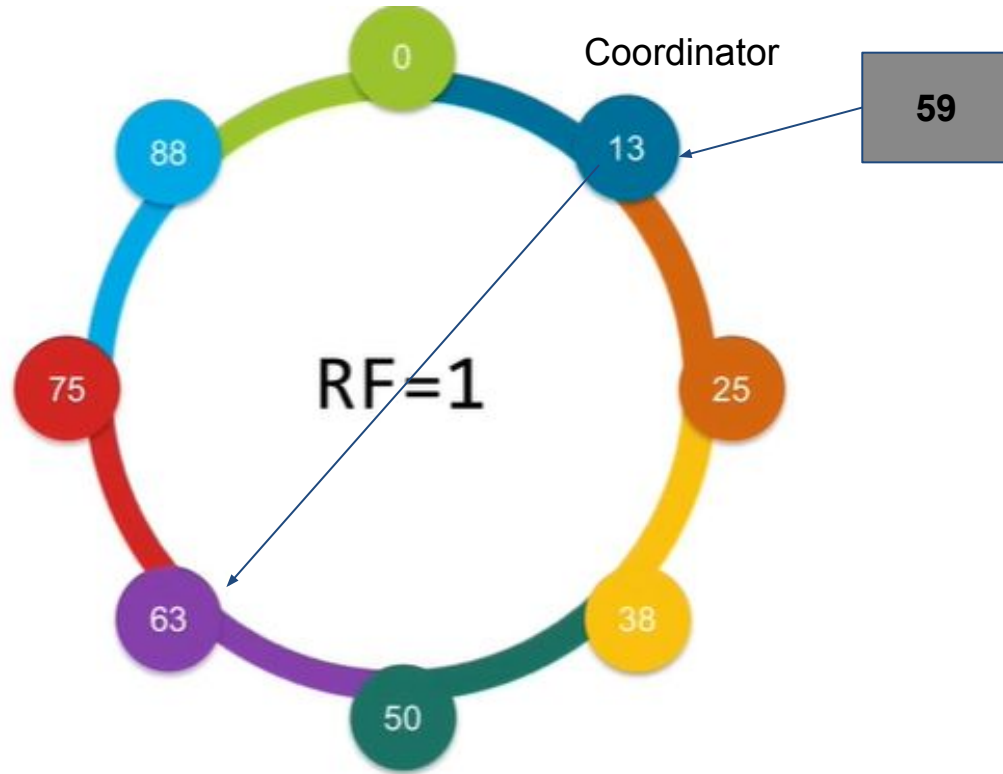
Cassandra places replicas of data on different nodes based on these two factors.

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

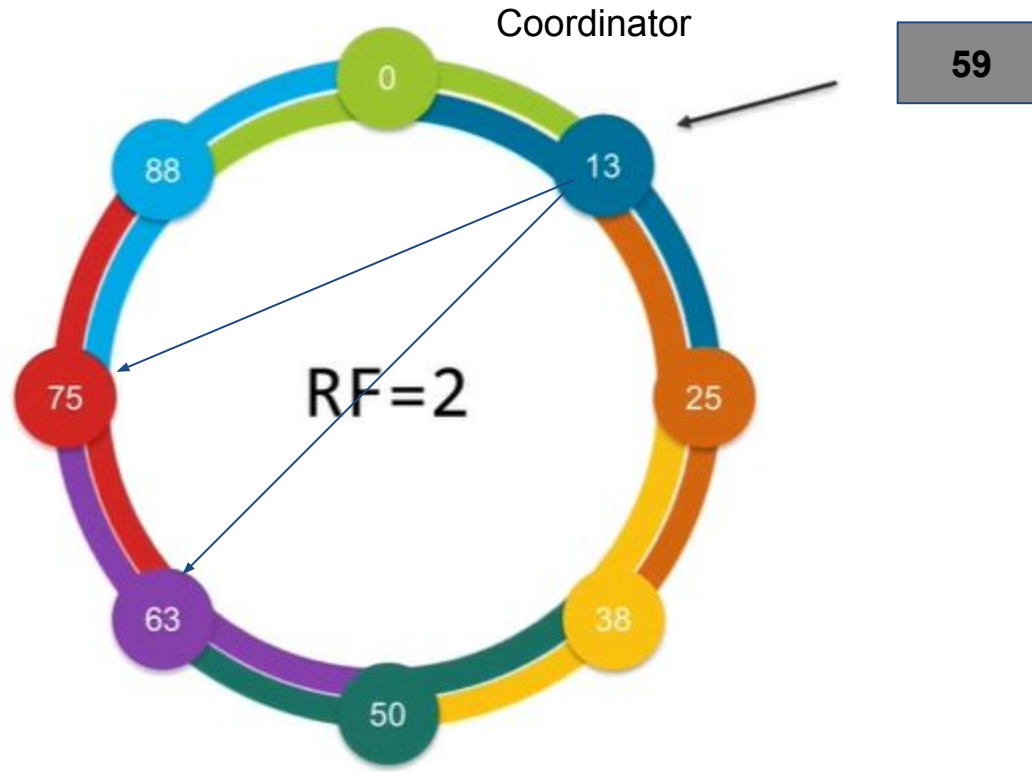
One Replication factor means that there is only a single copy of data while three replication factor means that there are three copies of the data on three different nodes.

For ensuring there is no single point of failure, **replication factor must be three**.

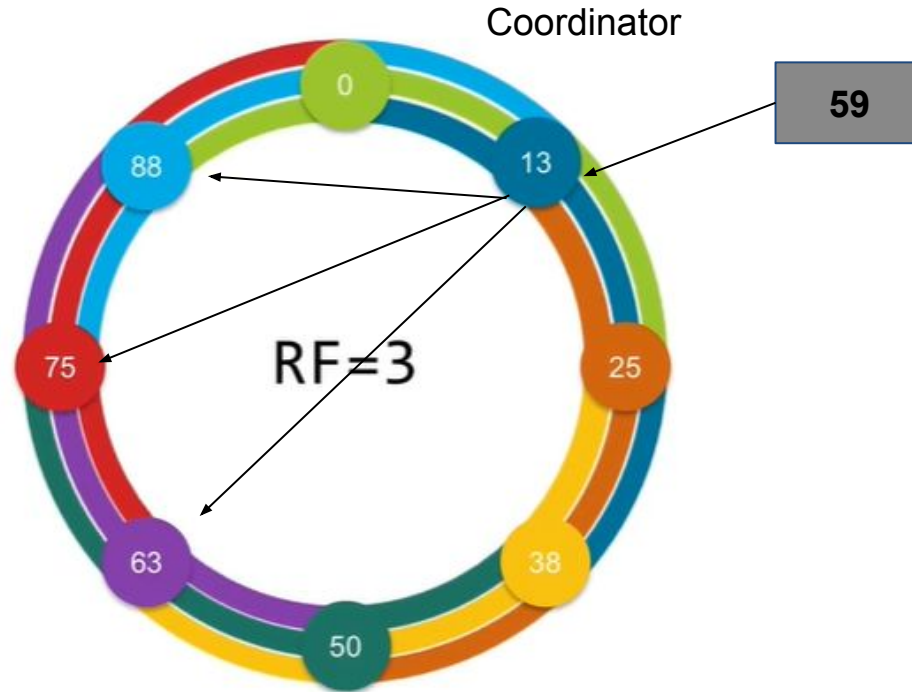
RF=1



RF=2



Default RF=3



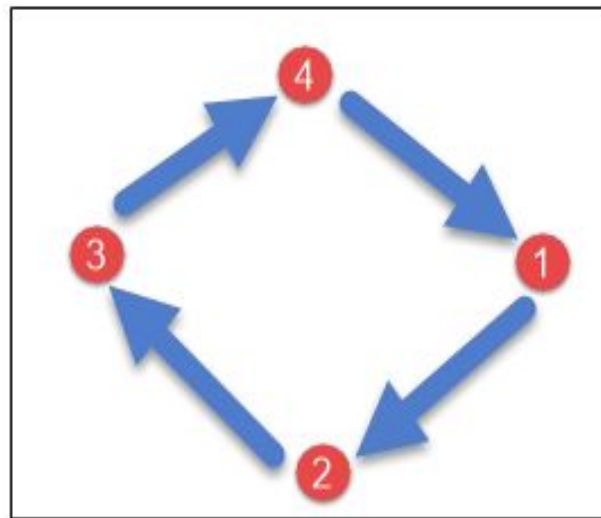
Replication Strategy



1. SimpleStrategy
2. NetworkTopologyStrategy

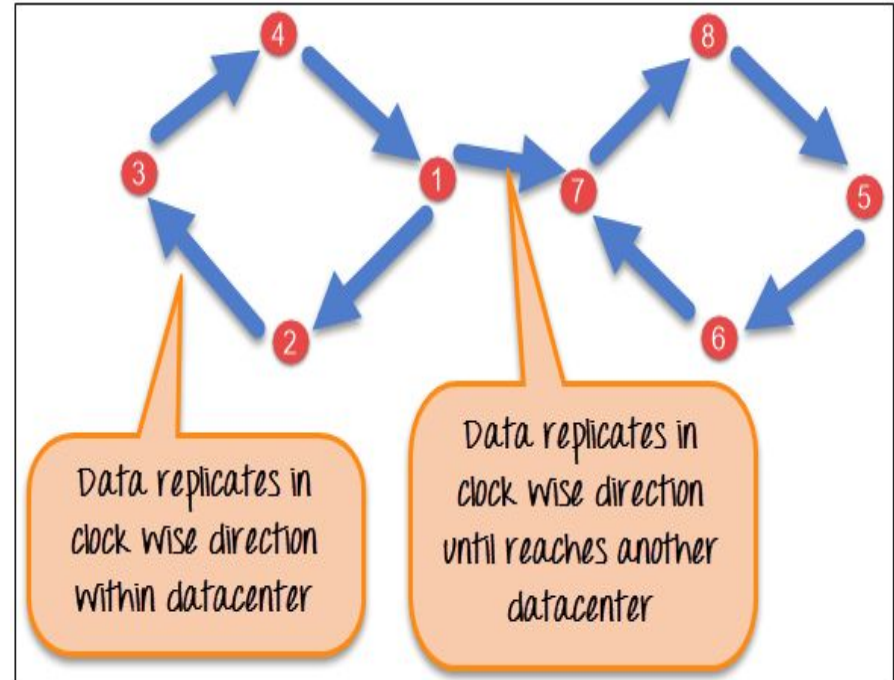
SimpleStrategy in Cassandra

- **SimpleStrategy** is used when you have just one data center.
- SimpleStrategy places the first replica on the node selected by the partitioner.
- After that, remaining replicas are placed in **clockwise direction** in the Node ring.

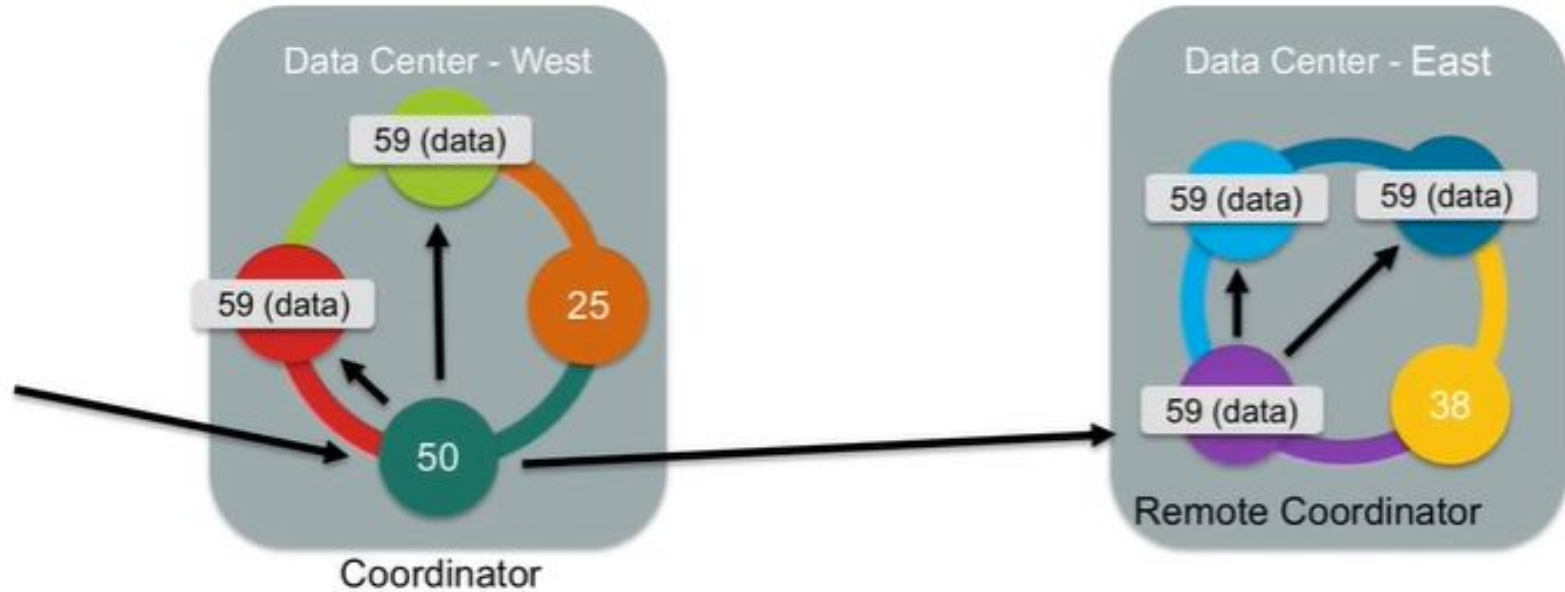


NETWORK TOPOLOGY STRATEGY

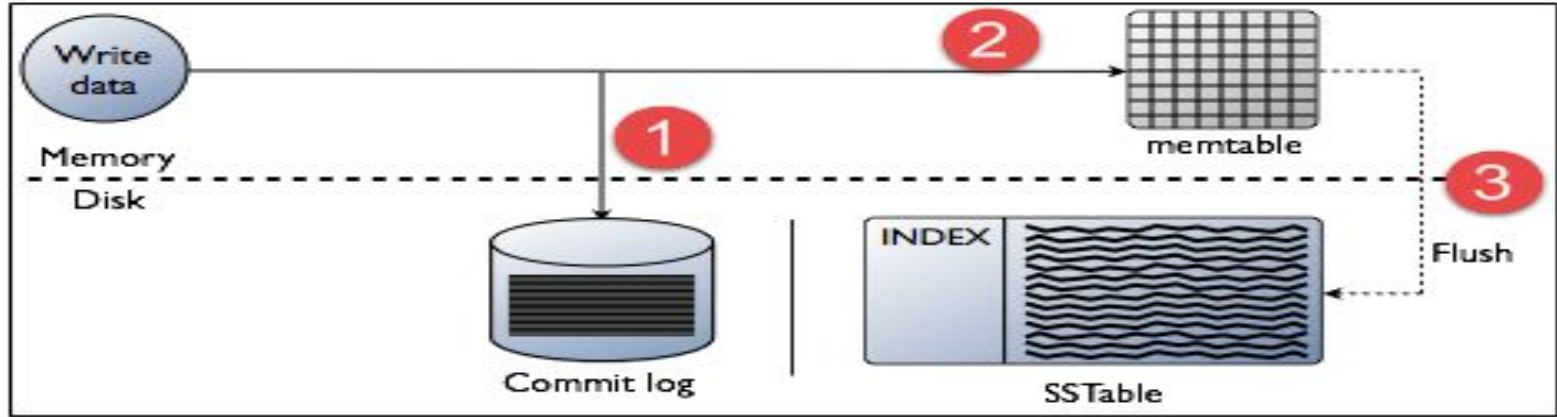
- This strategy is used when we have more than two data centers and it tries to place replicas on different racks in the same data center.




Multi Data Center Replication



Writes in Cassandra



1. When write request comes to the node, first of all, it logs in the commit log.
2. Then Cassandra writes the data in the mem-table. Data written in the mem-table on each write request also writes in commit log separately. Mem-table is a temporarily stored data in the memory while Commit log logs the transaction records for backup purposes.
3. When mem-table is full, data is flushed to the SSTable data file.

- 
1. **Memtable** – Memtable is an in-memory cache (RAM), it is like Hash Table data structure where contents are stored as key-value pairs.
 2. **SSTable(Sorted Strings Table)**- When a Memtable exceeds the configured size, a flush is triggered which moves data from Memtable to SSTable. SSTable a flat file of Key-Value pairs which is sorted by keys and it is used by Cassandra to persist the data on the disk, SSTable files are immutable, each SSTable contains a sequence of blocks where by default each block is of 64 KB size and it can be configured.
 3. **Commit Log**-The main purpose of Commit Log is to recreate the Memtable in case if a node gets crashed, Commit Log is a flat file which is created on Disk. Once the Memtable is full, the data will be flushed (written) to SSTable that's when data will be Purged from Commit Log as well.

Write Consistency level

- Consistency indicates how recent and in-sync all replicas of a row of data are. Cassandra prefers availability over consistency.
- For write operations, the consistency level specifies **how many replica nodes must acknowledge** back before the coordinator successfully reports back to the client.

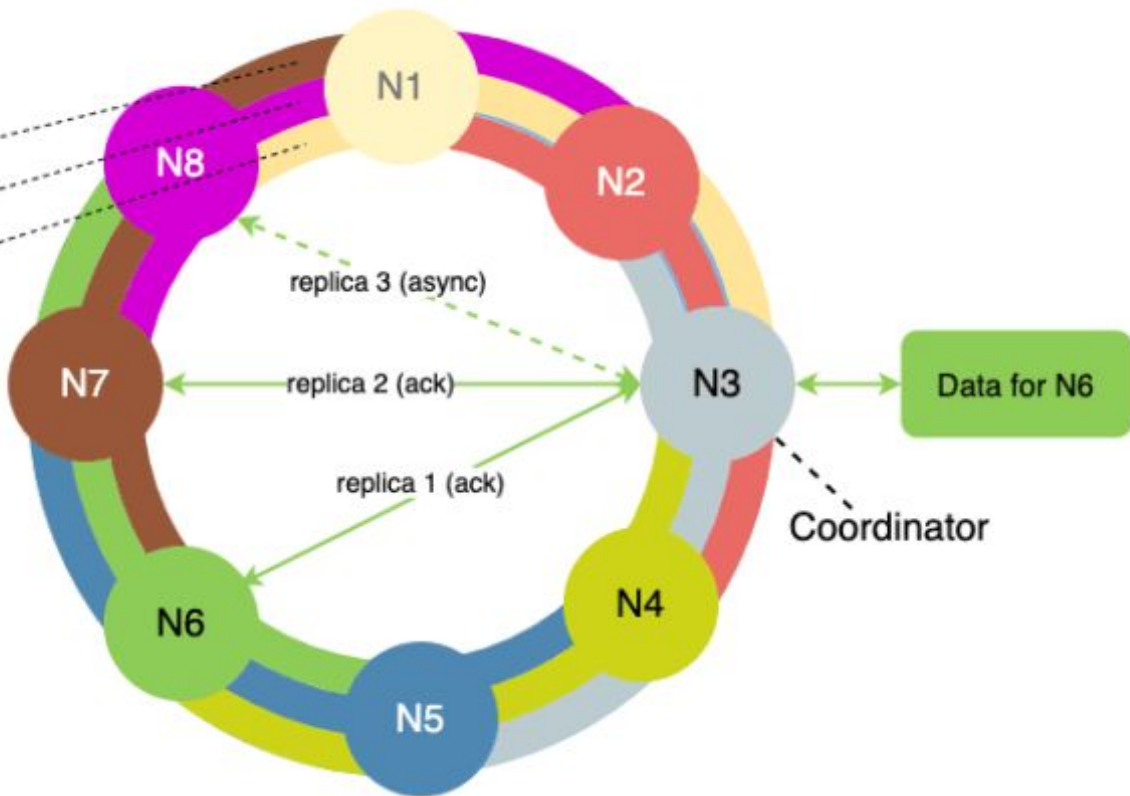
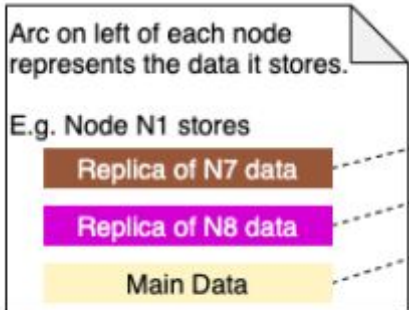
Setting	Description
ANY	Storing a hint at minimum is satisfactory
ONE, TWO, THREE	Checks closest node(s) to coordinator
QUORUM	Majority vote, $(\text{sum_of_replication_factors} / 2) + 1$
LOCAL_ONE	Closest node to coordinator in same data center
LOCAL_QUORUM	Closest quorum of nodes in same data center
EACH_QUORUM	Quorum of nodes in each data center, applies to writes only
ALL	Every node must participate

Writes in Cassandra Example

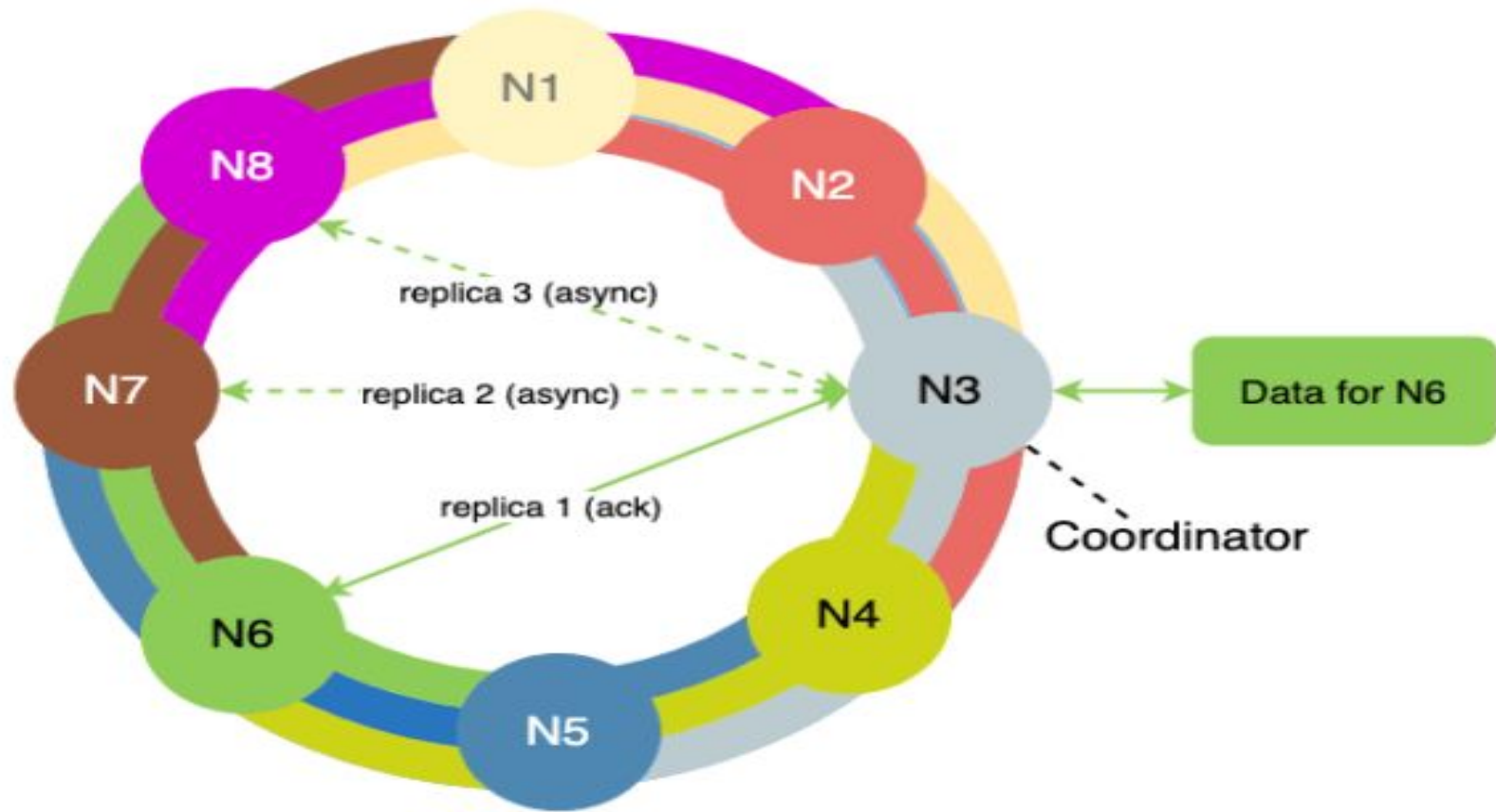
- `WRITE_CONSISTENCY_LEVEL=TWO`
- `TABLE_REPLICATION_FACTOR=3`
- `REPLICATION_STRATEGY=SimpleStrategy`

Cassandra Write With RF = 3

CL = LOCAL_QUORUM



CL = ONE



Durable_writes

- Writes in CASSANDRA are durable.
- All writes to a replica node are recorded both in memory and in a commit log on disk before they are acknowledged as a success. If a crash or server failure occurs before the memtables are flushed to disk, the commit log is replayed on restart to recover any lost writes. In addition to the local durability (data immediately written to disk), the replication of data on other nodes strengthens durability.
- By default, the durable_writes properties of a table is set to **true**, however it can be set to false.

Write path

MemTable →

RAM

4	IgotUr Data	TX	Austin
5	Always Onomnom	TX	Dallas
2	ComeTo DSE	TX	Dallas
6	Lone Star	TX	El Paso
1	Dev Awesome	TX	Houston
3	Lone Node	TX	Snyder

Commit Log →

HDD

1	Dev Awesome	TX	Houston
2	ComeTo DSE	TX	Dallas
3	Lone Node	TX	Snyder
4	IgotUr Data	TX	Austin
5	Always Onomnom	TX	Dallas
6	Lone Star	TX	El Paso

Data Flush to SSTable when memtable is full

RAM

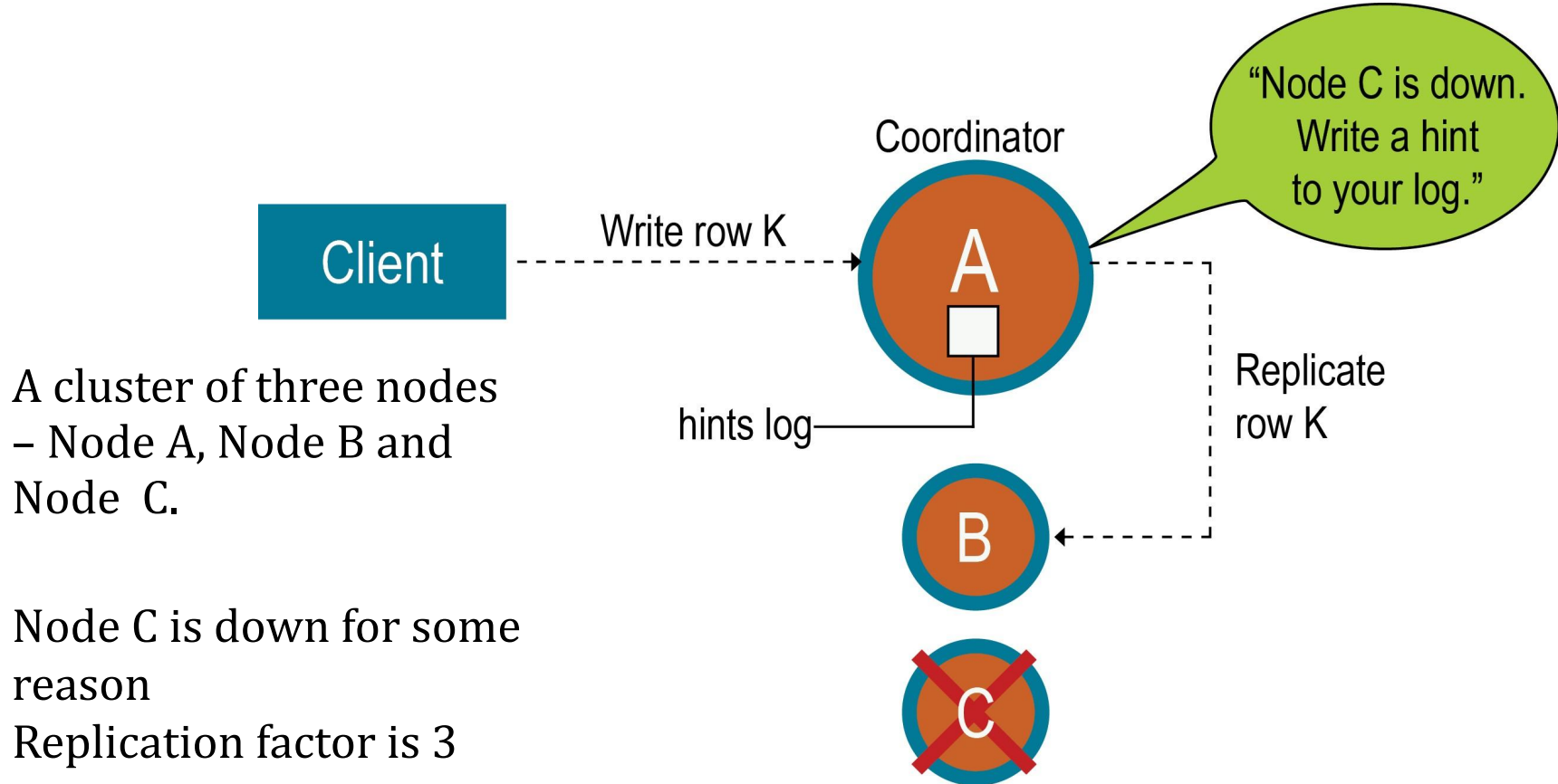
HDD

SSTable
(immutable)



4	IgotUr Data	TX	Austin
5	Always Onomnom	TX	Dallas
2	ComeTo DSE	TX	Dallas
4	Lone Star	TX	El Paso
5	Dev Awesome	TX	Houston
6	Lone Node	TX	Snyder

Hinted Handoffs



Hinted Handoffs

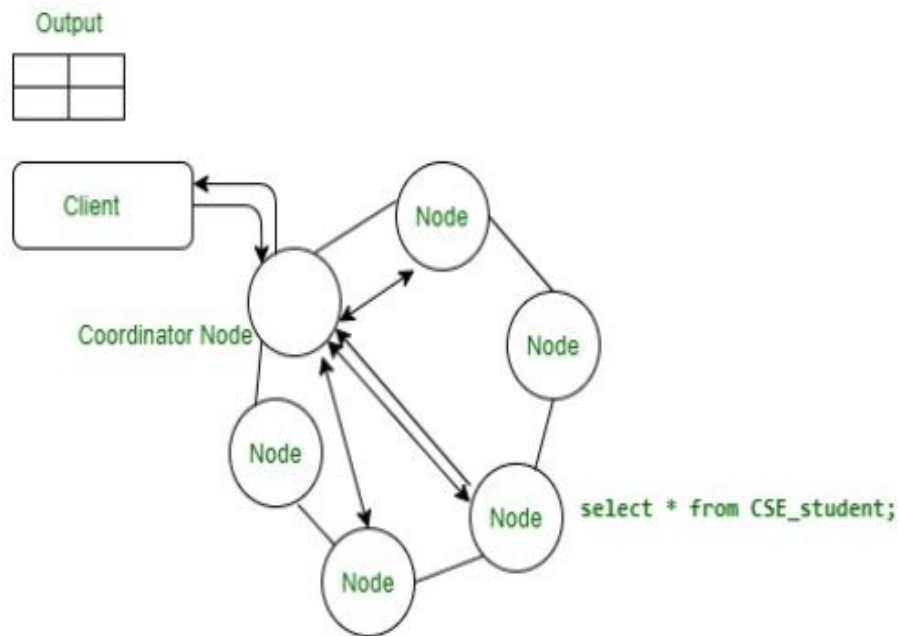
- Cassandra works on philosophy that it should be always available for writes.
- This is achieved by hinted handoffs.
- **once the node comes up the coordinator node will write the information from the stored hints.**
- Hint contains following information:
 - Location of the node on which the replica is to be placed
 - Version metadata
 - The actual data
- Hints are by **default** stored in the node for **3 hours**.

Read Path Cassandra

In Cassandra while reading data, any server may be queried which acts as the coordinator.

when we want to access read data then we contact nodes with requested key.

In a data center, on each node, data is pulled from SStable and is merged.



Consistency Level on Read

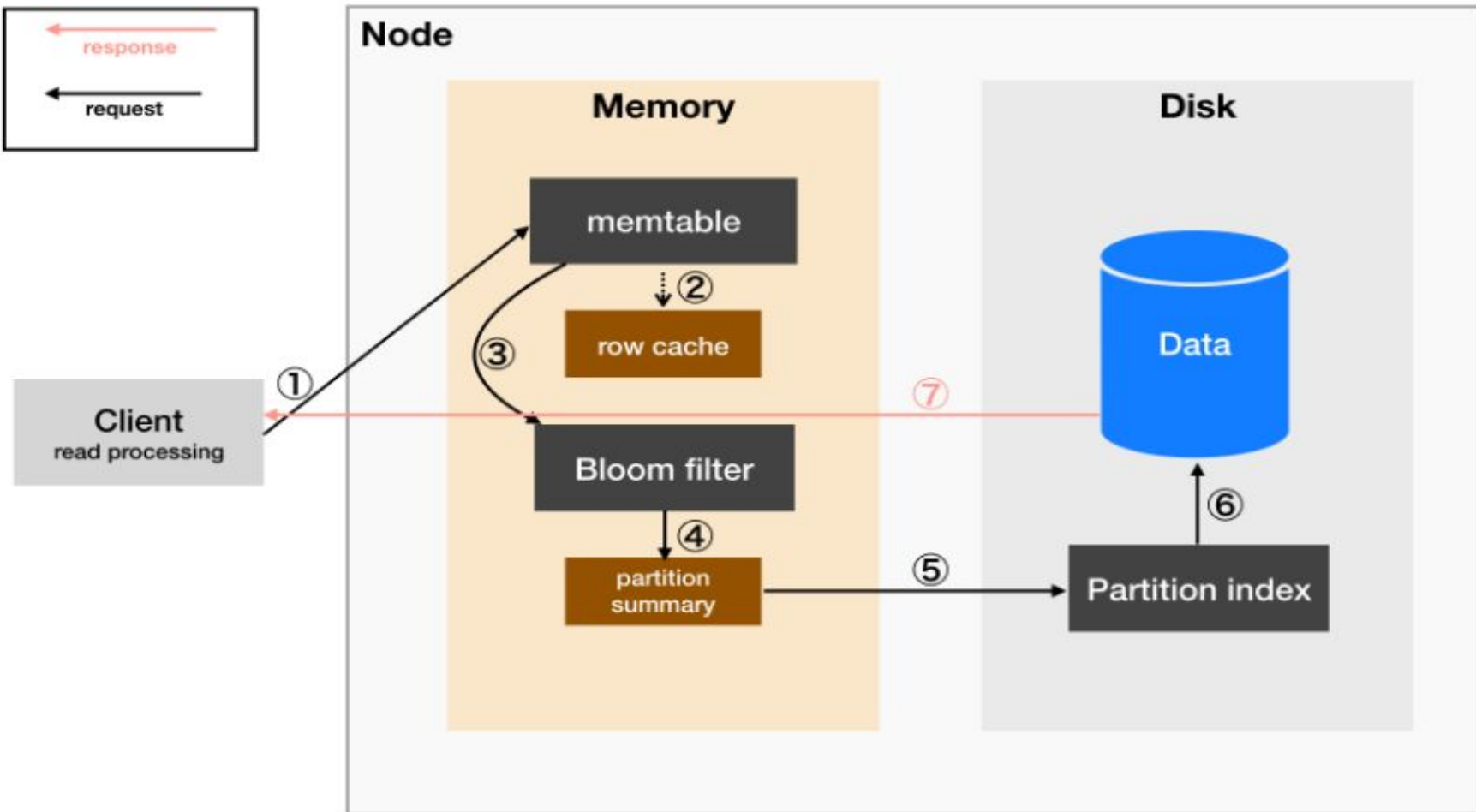
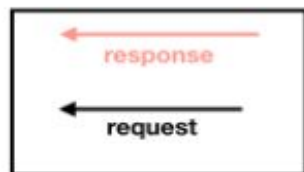
For read operations, the consistency level specifies how many replica nodes must respond with the latest consistent data before the coordinator successfully sends the data back to the client.

- **ONE** – Requests a response from a the closest node holding the data
- **QUORUM** – Returns a result from a quorum of servers with most recent timestamp. E.g. $\{(RF+1)/2\}$
- **LOCAL_QUORUM** – Same as above, but from the same data center as the coordinator node
- **EACH_QUORUM** – Same as above, but from all data centers
- **ALL** – Returns a result after all replica nodes have responded. Highest level of consistency and lowest level of availability

Read Path in Cassandra



1. Check data in memtable.
 - a. If row cache is enable, check row cache.
2. Check bloom filter.
 - a. If partition key cache is enable, check partition key cache.
 - b. If partition key cache has data, move to offset map.
3. Check partition summary. Access to partition index.
4. Get data on disk.
5. Fetch SSTable data on disk.



Reading Data



MemTable

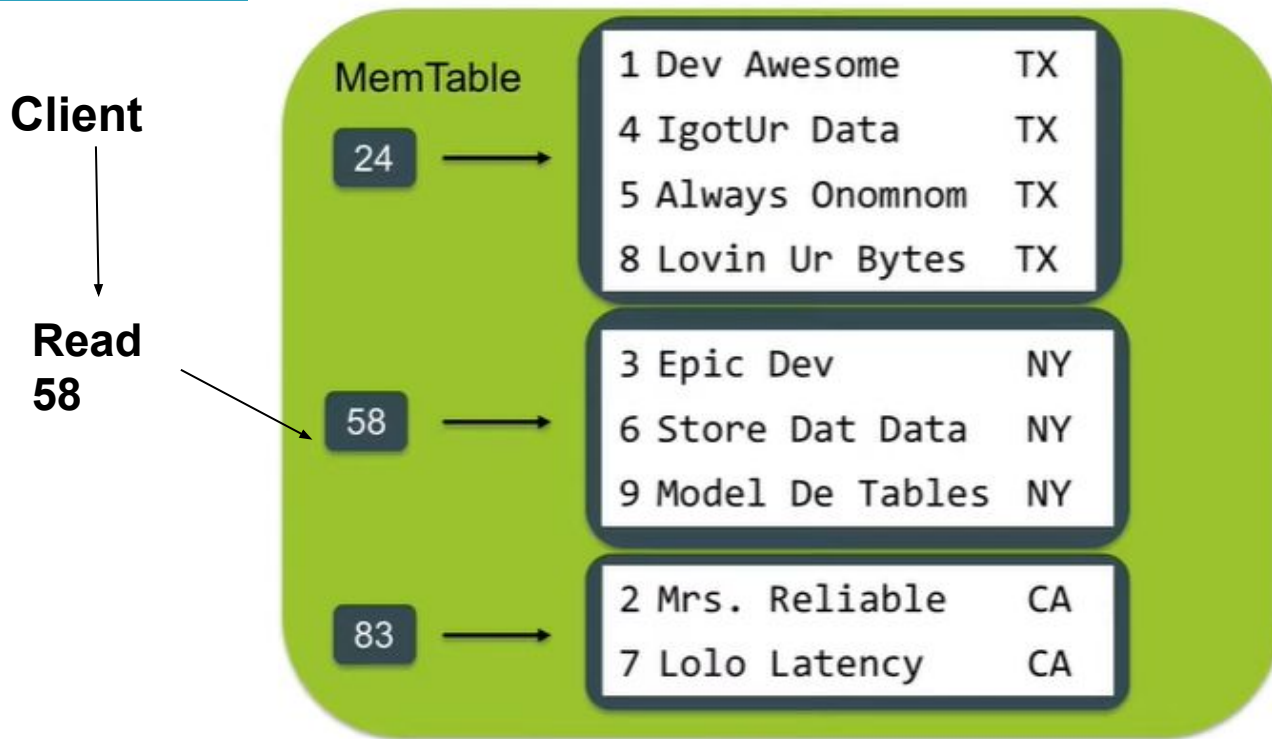
The diagram illustrates the data reading process. On the left, a green rounded rectangle labeled 'MemTable' is shown. On the right, three gray rounded rectangles are stacked vertically, labeled 'SSTable #1', 'SSTable #2', and 'SSTable #3'. A blue arrow points from the MemTable towards the SSTables, indicating the flow of data reading.

SSTable #1

SSTable #2

SSTable #3

Reading Data from a MemTable



Row Cache

- The row cache, in off-heap memory, holds rows most recently read from the local SSTables.
- Each local read operation stores its result set in the row cache and sends it to the coordinator node.
- The next read first checks the row cache.
- If the required data is there, Cassandra returns it immediately. This initial read can **save** further seeks in the Bloom filter, partition key cache, partition summary, partition index, and SSTables.
- Cassandra uses **LRU** (least-recently-used) eviction to ensure that the row cache is refreshed with the most frequently accessed rows. The size of the row cache can be **configured** in the **cassandra.yaml** file.

Bloom Filter

- In the read path, Cassandra merges data on disk (in SSTables) with data in RAM (in memtables). To avoid checking every SSTable data file for the partition being requested, Cassandra employs a data structure known as a bloom filter.
- **Bloom filters** are a probabilistic data structure that allows Cassandra to determine one of two possible states: -
 1. The data definitely does not exist in the given file, or -
 2. The data probably exists in the given file.

Working of Bloom Filter

- An empty bloom filter is a bit array of m bits, all of which are initially set to zero. A bit array is an extremely space-efficient data structure that has each position in the array set to either 0 or 1.
- A bloom filter also includes a set of k hash functions with which we hash incoming values. These hash functions must all have a range of 0 to $m-1$. If these hash functions match an incoming value with an index in the bit array, the bloom filter will make sure the bit at that position in the array is 1.

Insert "Hello"
insert "Hello"

hash("hello")=1

hash("hello")=3

bit		
1	0	1
1	2	3

index

Reading an SSTable

(Using Partition index-Partition cache)

Client : Request 58

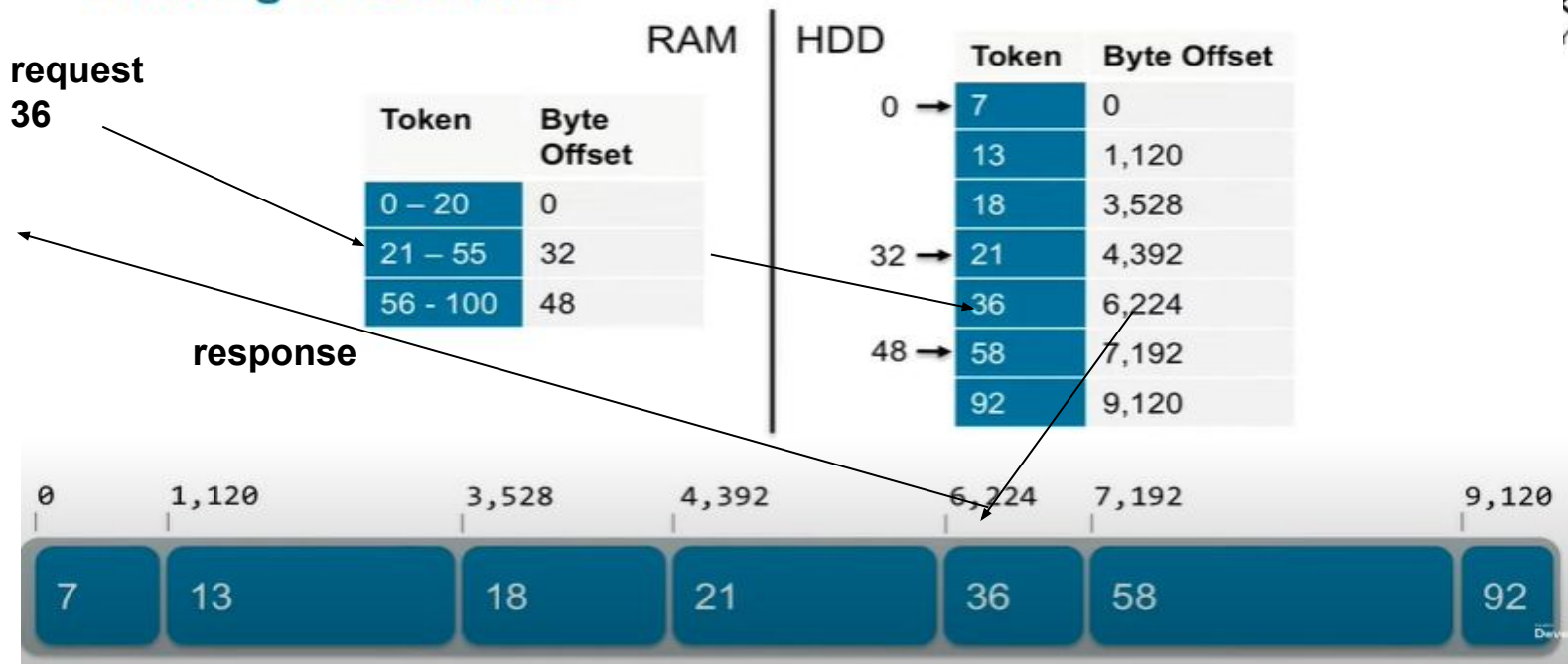
Token	Byte Offset
7	0
13	1,120
18	3,528
21	4,392
36	6,224
58	7,192
92	9,120

A C A D E M Y



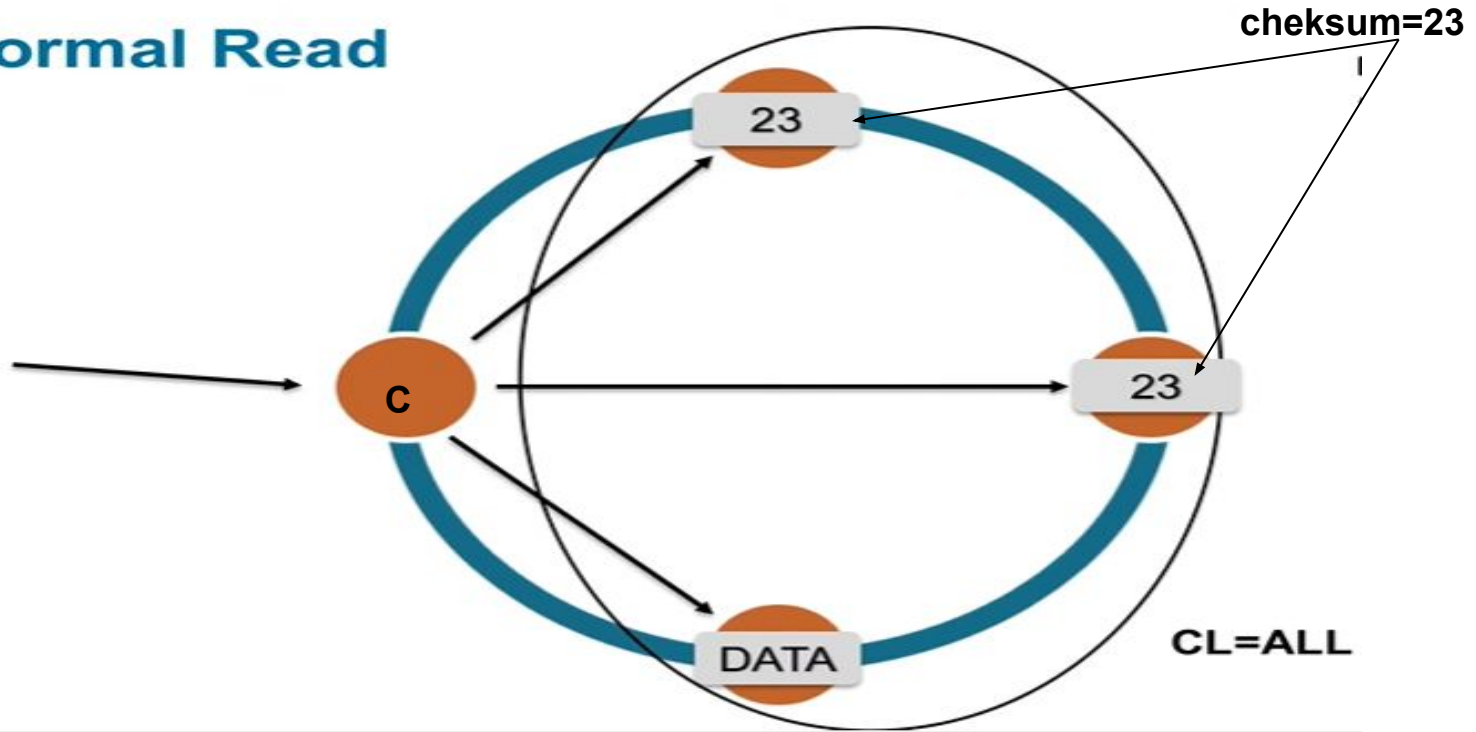
Large Partition index (Use Partition Summary)

Reading an SSTable



Normal Read(Request to read data)

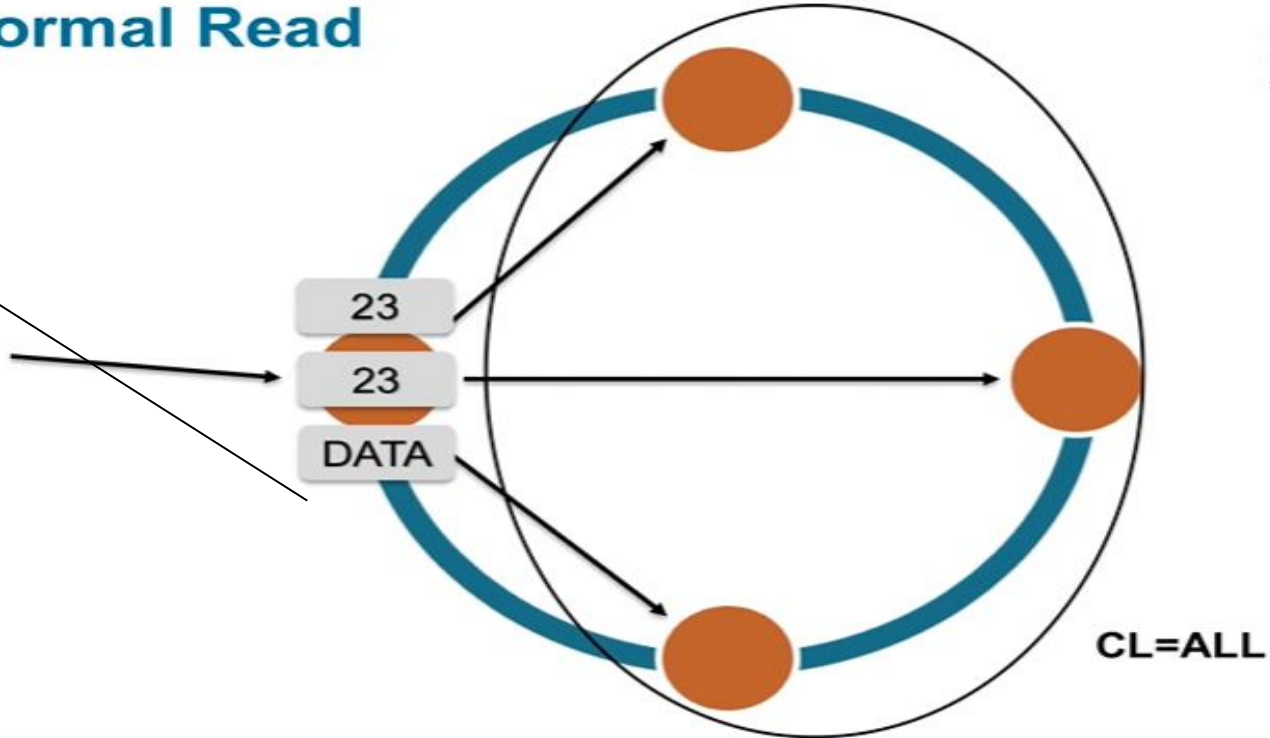
Normal Read



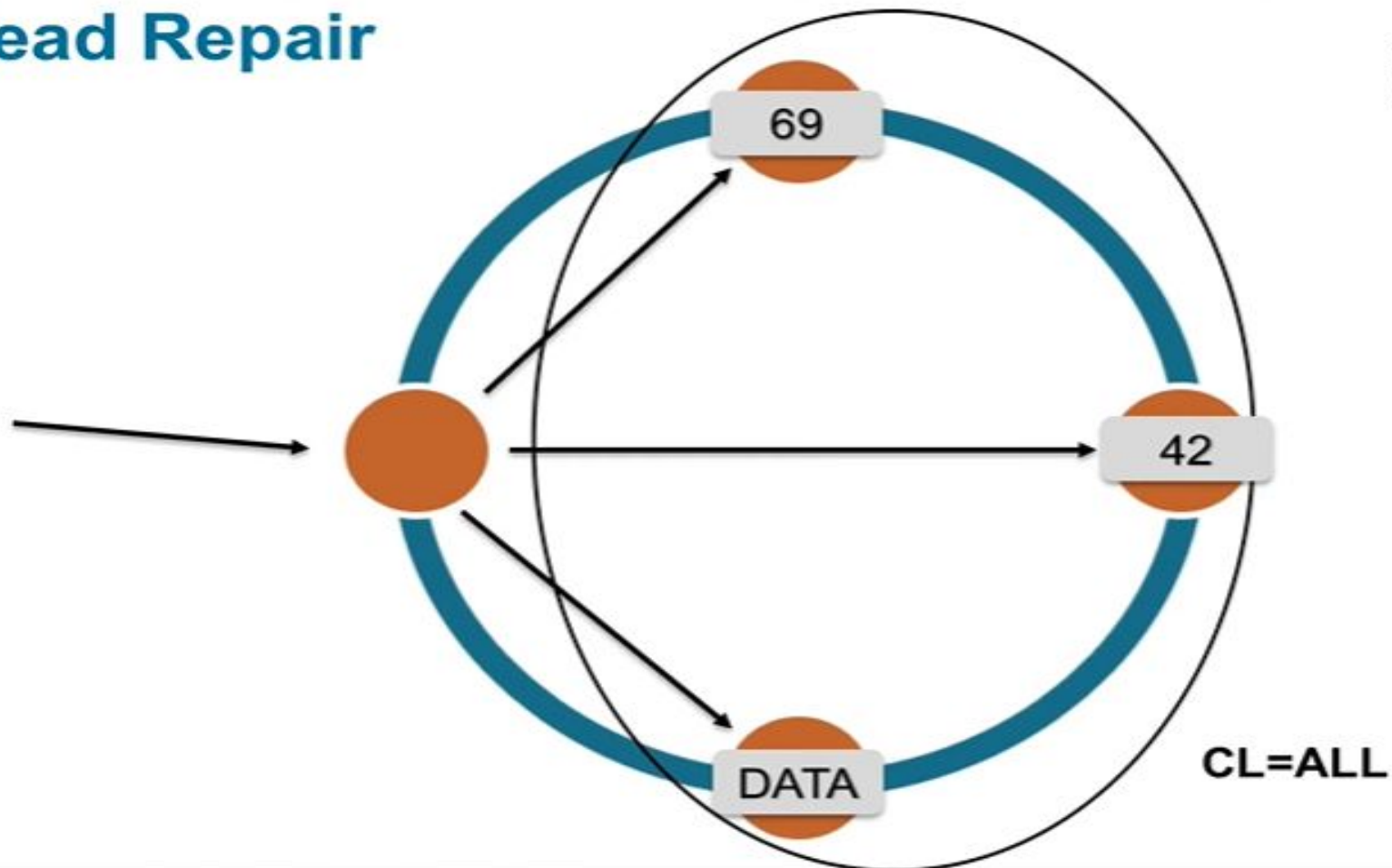
Normal Read(Compare checksum)

Normal Read

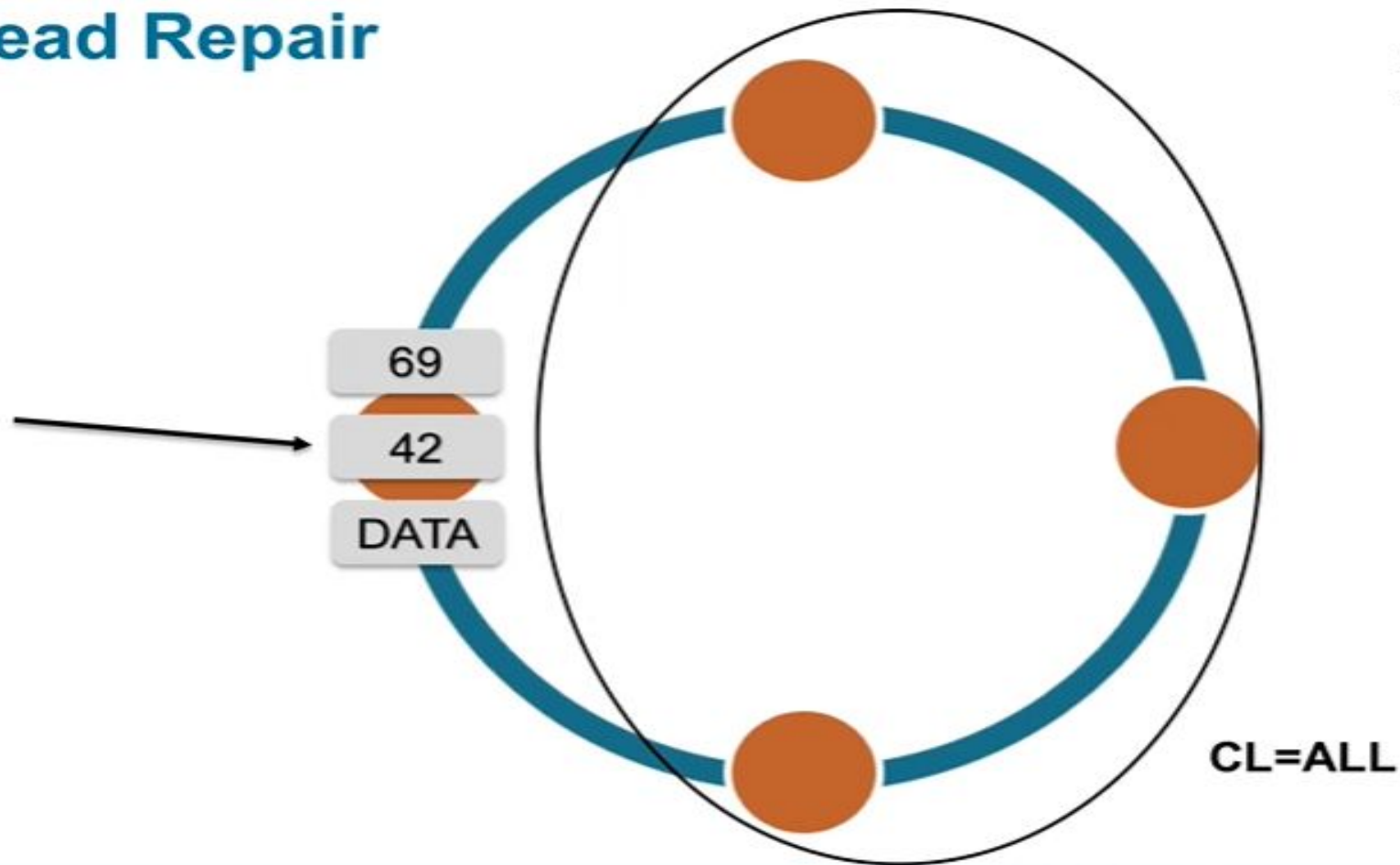
Client



Read Repair

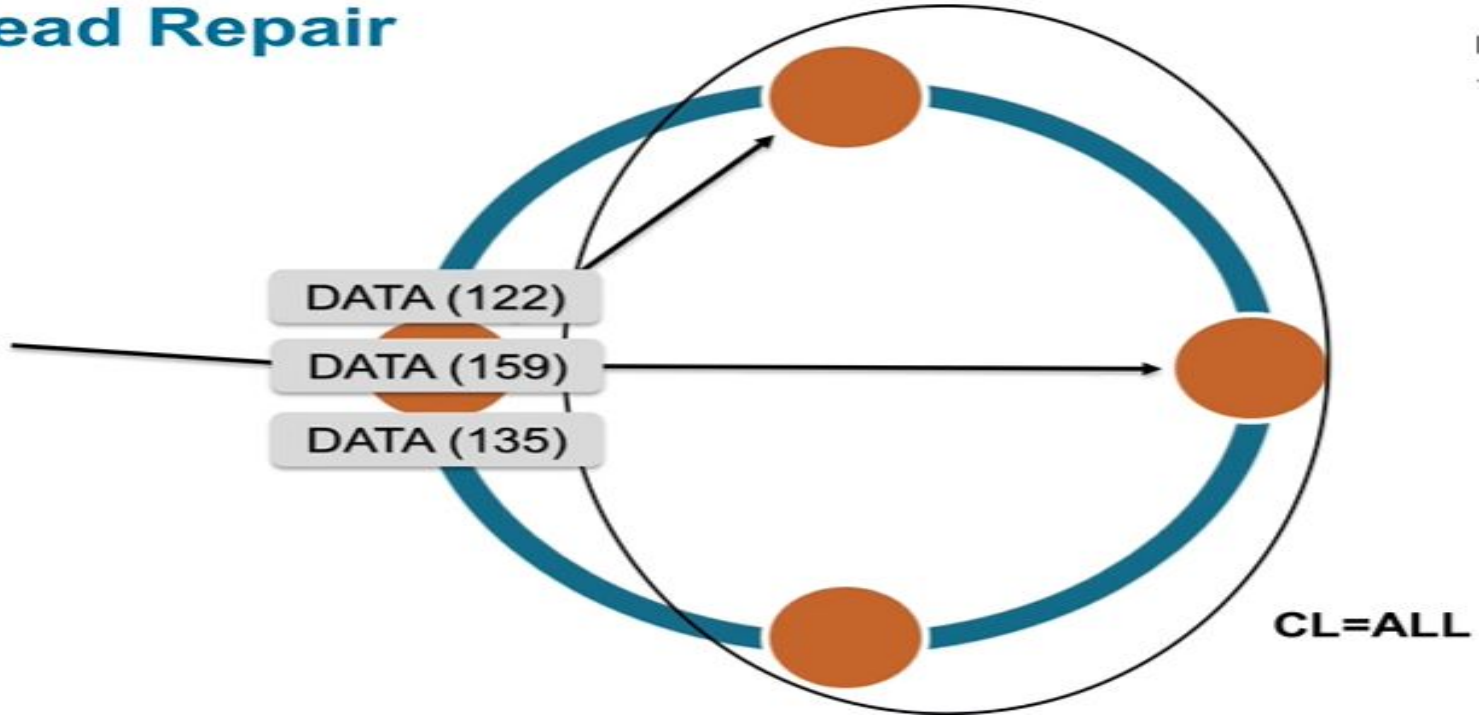


Read Repair



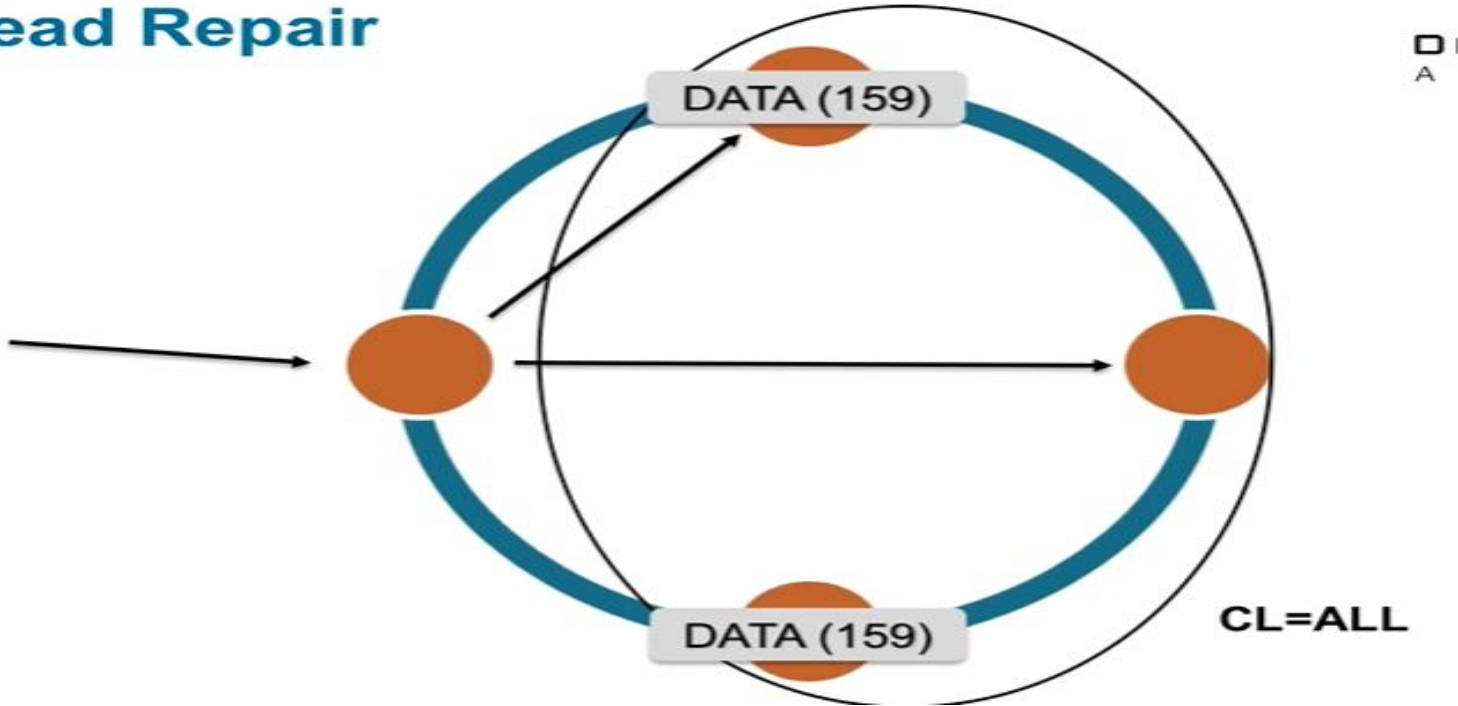
Compare Timestamp of all node

Read Repair



Replace all the replicas with latest timestamp

Read Repair



Read Repair

- Read Repair is the process of **fixing inconsistencies** among the replica nodes at the time of read request.
- 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.

Anti-Entropy



Anti-entropy Repair is a process of comparing the data of all replicas and updating them with the newest version of data using Merkle Tree. Anti-entropy repair is triggered manually. It has two phases to the process:

1. Building a Merkle tree for each replica
2. Comparing the Merkle trees to discover differences

Anti-Entropy and read repair

- For repairing unread data, Cassandra uses an anti-entropy version of the gossip protocol.
- Anti-entropy implies **comparing all the replicas of each piece of data and updating each replica to the newest version.**
- The read repair operation is performed either before or after returning the value to the client as per the specified consistency level.

Cassandra installation(Prerequisites)

1. Install the latest version of **Java 8 or Java 11**

To verify that you have the correct version of java installed, type **java -version**.

2. For using cqlsh, the latest version of Python 3.6+ or **Python 2.7** (support deprecated). To verify that you have the correct version of Python installed, type **python --version**.

Cassandra installation

1. **Install cassandra** :-sudo yum install cassandra
2. **set environment variable**:-C:\apache-cassandra-4.0.11\bin
3. **Execute 'nodetool status'**

```
root@hadoop-clone ~]# nodetool status
```

```
Datacenter: datacenter1
```

```
=====
```

```
Status=Up/Down
```

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

Address	Load	Tokens	Owns (effective)	Host ID	Rack
UN 127.0.0.1	70.92 KiB	256	100.0%	53c1e408-a30f-42ee-9453-531ad873d9dd	rack1

4. Displays datacenter and rack information with some other interesting details like tokens, Owns (Represents what percentage of token range is owned by a machine, 100% for a single machine)

Cassandra Query Language (CQL)

- Cassandra provides a prompt Cassandra query language shell (cqlsh) that allows users to communicate with it. Using this shell, you can execute Cassandra Query Language (CQL).
- Using cqlsh, you can
 - define a schema,
 - insert data
 - execute a query.

Starting cassandra cqlshell



```
[hadoop@linux bin]$ cqlsh
```

```
Connected to Test Cluster at 127.0.0.1:9042.
```

```
[cqlsh 5.0.1 | Cassandra 2.1.2 | CQL spec 3.2.0 | Native protocol v3]
```

```
Use HELP for help.
```

```
cqlsh>
```

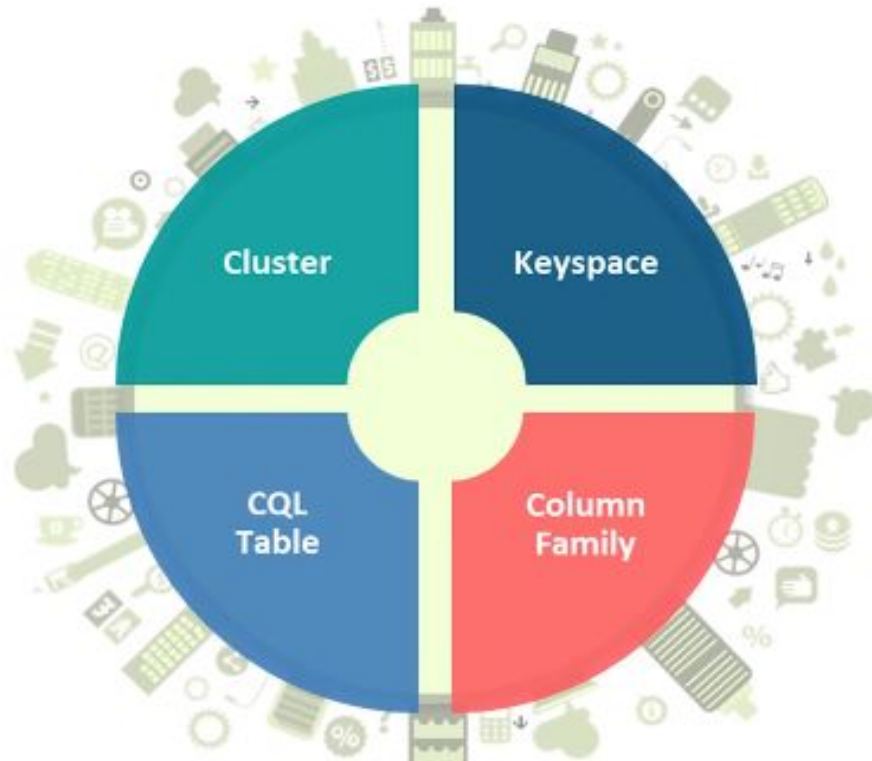
Adding new node

- To add new node to a different datacenter or rack, we need to edit 'cassandra- rackdc.properties' file.
- This file is located in conf directory of cassandra or /etc/cassandra installation directory.
- Edit 'dc' and 'rack' properties in this file.

Replication in Cassandra

- Replication factor determines to how many nodes a keyspace (like databases in relation databases) will be replicated.
- RF should be >1 and $<\text{no. of nodes in cluster}$
- Two Strategies:
 - SimpleStrategy
 - NetworkTopologyStrategy

CASSANDRA DATABASE ELEMENTS



Clusters

Cluster is a container for Keyspaces

Keyspace

Keyspace corresponds to database

Column Family

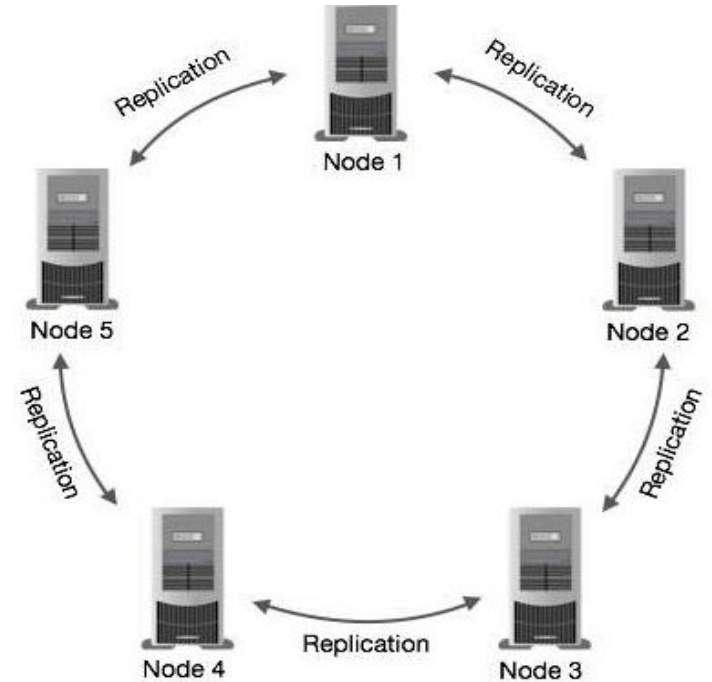
Set of rows with a similar structure

CQL Table

Tables in Cassandra Query Language

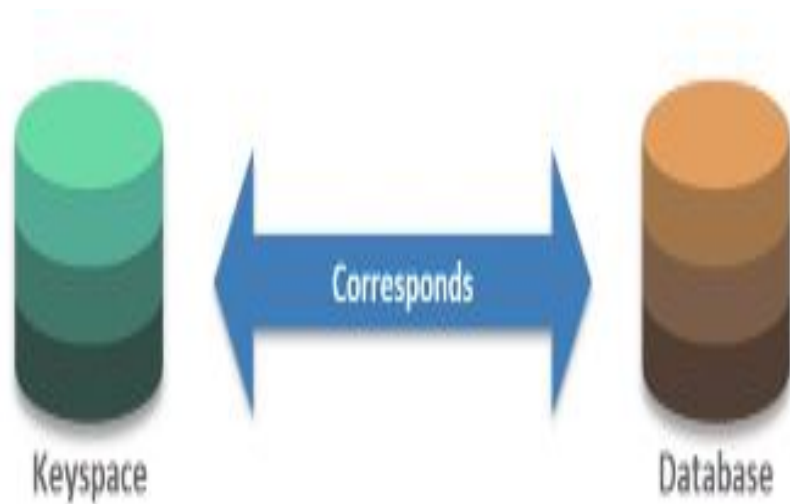
CLUSTER

- The outermost structure in Cassandra is the Cluster which is the container of Keyspaces.
- Sometimes called as ring, because Cassandra assigns data to nodes in the cluster by arranging them in ring fashion.



Keyspace

- The outermost structure container of data in Cassandra.
- Like a relational database, a key space has a name and set of attributes, based on this Cassandra decides how to distribute the data across various nodes, how to perform partition and clustering.
- **Keyspace is used to group Column Families together.**
- A cluster contain one key space per node.



Create KEYSPACE

- Command to **create** a KEYSPACE(use tab):
`'CREATE KEYSPACE test_keyspace with replication = {'class':'SimpleStrategy', 'replication_factor':1} and durable_writes = 'true';'`
- DURABLE_WRITES = true|false
 - Optionally (not recommended), bypass the commit log when writing to the keyspace by disabling durable writes (DURABLE_WRITES = false). Default value is true.
 - **CAUTION:** Never disable durable writes when using SimpleStrategy replication.

Create KEYSPACE(networktopology)

- CREATE KEYSPACE excalibur WITH replication = {'class': 'NetworkTopologyStrategy', 'replication_factor' : 3}; DESCRIBE KEYSPACE excalibur;

will result in

- CREATE KEYSPACE excalibur WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1': '3', 'DC2': '3'} AND durable_writes = true;

Create KEYSPACE(networktopology)

- CREATE KEYSPACE excalibur WITH replication = {'class': 'NetworkTopologyStrategy', 'replication_factor' : 3, 'DC2': 2};

DESCRIBE KEYSPACE excalibur;

will result in:

- CREATE KEYSPACE excalibur WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1': '3', 'DC2': '2'} AND durable_writes = true;

Alter Keyspace

```
ALTER KEYSPACE excelsior WITH replication =  
{'class': 'SimpleStrategy', 'replication_factor' : 4};
```

Describe and Drop KEYSPACE

- Command to **drop** a KEYSPACE:
‘ DROP KEYSPACE test_keyspace;’
- Command to **describe** KEYSPACES:
‘DESC KEYSPACES;’
- Command to **use** a KEYSPACE:
‘USE test_keyspace;’

Data Types

ascii	string	ASCII character string
bigint	integer	64-bit signed long
blob	blob	Arbitrary bytes (no validation)
boolean	boolean	Either true or false

Data Types

counter	integer	Counter column (64-bit signed value). See counters for details.
date	integer, string	A date (with no corresponding time value). See dates below for details.
decimal	integer, float	Variable-precision decimal
double	integer float	64-bit IEEE-754 floating point

Data Types

float	integer, float	32-bit IEEE-754 floating point
inet	string	An IP address, either IPv4 (4 bytes long) or IPv6 (16 bytes long). Note that there is no <code>inet</code> constant, IP address should be input as strings.
int	integer	32-bit signed int
smallint	integer	16-bit signed int

Data Types

<code>text</code>	<code>string</code>	UTF8 encoded string
<code>timestamp</code>	<code>integer, string</code>	A timestamp (date and time) with millisecond precision. See <code>timestamps</code> below for details.
<code>timeuuid</code>	<code>uuid</code>	Version 1 UUID, generally used as a “conflict-free” timestamp. Also see <code>timeuuid-functions</code> .
<code>uuid</code>	<code>uuid</code>	A UUID (of any version)

Create and Drop tables

- Command to **create** a table:
`'CREATE table employee_by_id(id int primary key, name text, position text);'`
- ✓ This will create a table where data is partitioned on id which is a primary key.
- Command to **drop** a table:
`'DROP table employee_by_id;'`
- Command to **describe** tables:
`'DESC TABLES;'`
`'DESC TABLE employee_by_id' (For individual table)`

Primary Key in tables

- `CREATE TABLE t (a int, b int, c int, d int, PRIMARY KEY (a))`

or

- `CREATE TABLE t (a int PRIMARY KEY , b int, c int, d int)`

Primary Key in tables

- **Partitioning key:** column on which data is partitioned, A partition is the set of rows that share the same value for their partition key.
- CREATE TABLE t (a int, b int, c int, d int, PRIMARY KEY ((a, b), c, d));
- INSERT INTO t (a, b, c, d) VALUES (0,0,0,0);
- INSERT INTO t (a, b, c, d) VALUES (0,0,1,1);
- INSERT INTO t (a, b, c, d) VALUES (0,1,2,2);
- INSERT INTO t (a, b, c, d) VALUES (0,1,3,3);
- INSERT INTO t (a, b, c, d) VALUES (1,1,4,4);
- SELECT * FROM t;

a	b	c	d	
0	0	0	0	1
0	0	1	1	
0	1	2	2	2
0	1	3	3	
1	1	4	4	3

(5 rows)

Clustering Columns

- The clustering columns of a table define the **clustering order** for the partition of that table. For a given partition, all rows are ordered by that clustering order. Clustering columns also add uniqueness to a row in a table.
- `CREATE TABLE t2 (a int, b int, c int, d int, PRIMARY KEY (a, b, c))`
- `INSERT INTO t2 (a, b, c, d) VALUES (0,0,0,0);`
- `INSERT INTO t2 (a, b, c, d) VALUES (0,0,1,1);`
- `INSERT INTO t2 (a, b, c, d) VALUES (0,1,2,2);`
- `INSERT INTO t2 (a, b, c, d) VALUES (0,1,3,3);`
- `INSERT INTO t2 (a, b, c, d) VALUES (1,1,4,4);`

a	b	c	d
1	1	4	4
0	0	0	0
0	0	1	1
0	1	2	2
0	1	3	3

(5 rows)

- **Primary Key**(column1,column2,column3) : Here, column1 is the partitioning key and column2 and column3 are the clustering columns.
- **Primary Key**((column4,column5),column2, column3) : Here, column4 and column5 are the partitioning keys and column2 and column3 are the clustering columns.

Table options(WITH comment)

```
CREATE TABLE monkey_species  
( species text PRIMARY KEY,  
  common_name text,  
  population varint  
  , average_size int )  
WITH comment='Important biological records';
```

WITH compaction



```
CREATE TABLE timeline
( userid uuid,
  posted_month int,
  posted_time uuid,
  body text,
  posted_by text,
  PRIMARY KEY (userid, posted_month, posted_time) )
WITH compaction = { 'class' : 'LeveledCompactionStrategy' };
```

```
cqlsh:test_keyspace> CREATE TABLE employee_by_uuid (id uuid PRIMARY KEY, first_name text, last_name text);
cqlsh:test_keyspace> SELECT * FROM employee_by_uuid;
```

id	first_name	last_name
----	------------	-----------

(0 rows)

```
cqlsh:test_keyspace> INSERT INTO employee_by_uuid (id, first_name, last_name) VALUES (uuid(), 'ton', 'dunne');
cqlsh:test_keyspace> SELECT * FROM employee_by_uuid;
```

id	first_name	last_name
bfc7603f-1e7c-45be-a26b-a023b9b198ae	ton	dunne

(1 rows)

```
cqlsh:test_keyspace> INSERT INTO employee_by_uuid (id, first_name, last_name) VALUES (uuid(), 'tin', 'smith');
cqlsh:test_keyspace> INSERT INTO employee_by_uuid (id, first_name, last_name) VALUES (uuid(), 'bob', 'hanson');
cqlsh:test_keyspace> SELECT * FROM employee_by_uuid;
```

id	first_name	last_name
991d8e50-5632-4f67-8e92-ca5564cc43b5	bob	hanson
bfc7603f-1e7c-45be-a26b-a023b9b198ae	ton	dunne
d26ec9f1-1e7f-4730-9bc9-43602e6826cf	tin	smith

(3 rows)

```
cqlsh:test_keyspace>
```

```
cqlsh:test_keyspace> CREATE TABLE employee_by_timeuuid (id timeuuid PRIMARY KEY, first_name text, last_name text);
cqlsh:test_keyspace> SELECT * FROM employee_by_timeuuid;
```

```
id | first_name | last_name
----+-----+-----
```

(0 rows)

```
cqlsh:test_keyspace> INSERT INTO employee_by_timeuuid (id, first_name, last_name) VALUES (now(), 'tin', 'jones');
... ;
```

SyntaxException: line 1:91 missing EOF at ':' (...now(), 'tin', 'jones')[:];)

```
cqlsh:test_keyspace> INSERT INTO employee_by_timeuuid (id, first_name, last_name) VALUES (now(), 'tin', 'jones');
cqlsh:test_keyspace> INSERT INTO employee_by_timeuuid (id, first_name, last_name) VALUES (now(), 'ally', 'smith');
cqlsh:test_keyspace> INSERT INTO employee_by_timeuuid (id, first_name, last_name) VALUES (now(), 'kate', 'smith');
cqlsh:test_keyspace> SELECT * FROM employee_by_timeuuid ;
```

```
id | first_name | last_name
----+-----+-----
a663c860-1676-11e9-ba31-97be5a9c8f57 | tin | jones
b7ef0e00-1676-11e9-ba31-97be5a9c8f57 | ally | smith
bbc2f960-1676-11e9-ba31-97be5a9c8f57 | kate | smith
```

(3 rows)

```
cqlsh:test_keyspace> █
```

```
cqlsh:test_keyspace> CREATE TABLE purchases_by_customer_id (id uuid PRIMARY KEY, purchases counter);
cqlsh:test_keyspace> UPDATE purchases_by_customer_id SET purchases = purchases + 1 WHERE id=uuid();
cqlsh:test_keyspace> SELECT * FROM purchases_by_customer_id;
```

id	purchases
0301558b-fc8e-495a-b808-9e24f8982d8f	1

(1 rows)

```
cqlsh:test_keyspace> UPDATE purchases_by_customer_id SET purchases = purchases + 1 WHERE id=uuid();
cqlsh:test_keyspace> UPDATE purchases_by_customer_id SET purchases = purchases + 1 WHERE id=uuid();
cqlsh:test_keyspace> SELECT * FROM purchases_by_customer_id;
```

id	purchases
0301558b-fc8e-495a-b808-9e24f8982d8f	1
2eefb15a-c85c-425f-bfd5-105d5c024ab8	1
53b6da52-dff5-4bc0-bcd9-dc9c6530d08e	1

(3 rows)

```
cqlsh:test_keyspace> UPDATE purchases_by_customer_id SET purchases = purchases + 1 WHERE id=uuid();
```

WITH CLUSTERING



```
CREATE TABLE loads
( machine inet,
  cpu int,
  mtime timeuuid,
  load float,
  PRIMARY KEY ((machine, cpu), mtime) )
WITH CLUSTERING ORDER BY (mtime DESC);
```

with caching



```
CREATE TABLE simple  
( id int, key text, value text,  
  PRIMARY KEY (key, value) )  
WITH caching = {'keys': 'ALL', 'rows_per_partition': 10};
```

Creating table sorted on one column

- To **create** a table partitioned by car_make:

‘CREATE table

employee_by_car_make(car_make text, id int,
car_model text, primary key (car_make,id));’

- ✓ This will create a table where data is partitioned on car_make and for a specific car_make, it is sorted on id.

Creating table sorted on two columns

- To **create** a table partitioned by car_make and sorted by two columns:

```
'CREATE table  
employee_by_car_make_sorted(car_make text, age int,  
id int, name text, primary key (car_make,age,id));'
```

- ✓ This will create a table where data is partitioned on car_make and then sorted by age and then for a specific age, it is sorted on id.

Imbalanced Partitions

- Sometimes, there is more data for a single partition and very less data is there for other partitions.
 - E.g. All employees are driving BMW and that specific node is overloaded and all other nodes are not getting many requests.
- To split the data more evenly, we may decide to split data on multiple partition keys.
 - E.g. car_make and car_model

Two Partition Keys

- Command to create a table multiple partition keys:

```
'CREATE table employee_by_car_make_and_model (car_make  
text, car_model text, id int, name text, primary key  
((car_make,car_model),id));'
```

- ✓ This will create a table where data is partitioned on car_make and then car_model and for a specific node, it is sorted on id.

Read Consistency Levels

- **ONE** – Requests a response from a the closest node holding the data
- **QUORUM** – Returns a result from a quorum of servers with most recent timestamp. E.g. $\{(RF+1)/2\}$
- **LOCAL_QUORUM** – Same as above, but from the same data center as the coordinator node
- **EACH_QUORUM** – Same as above, but from all data centers
- **ALL** – Returns a result after all replica nodes have responded. Highest level of consistency and lowest level of availability

Write Consistency Levels

- **ALL** – Highest level of consistency. Write to all replica nodes
- **EACH_QUORUM** – Quorum of replica nodes in all data centers
- **QUORUM** – Quorum of replica nodes
- **LOCAL_QUORUM** – Quorum of replica nodes in the same data center as the coordinator node
- **ONE** – At least one replica node
- **TWO** - At least two replica node
- **THREE** - At least three replica node
- **LOCAL_ONE** - At least one replica node in the local data center

Checking and Setting Consistency levels

- Command to check current consistency level :
‘CONSISTENCY;’
- Command to set current consistency level to Quorum :
‘CONSISTENCY QUORUM’
- Once set, consistency is applicable to all further CQL statements.

Insert Operation

- Text values need to be placed inside single quotes.
- Command to insert:

```
'Insert into    employee_by_id(id,name,position)      values
```

```
    (1,'Ram','Manager')
```

```
'Insert into    employee_by_id(id,name,position)  values
```

```
    (2,'Raj','CEO')
```

```
'Insert into employee_by_car_make (car_make, id, car_model) values
```

```
    ('BMW',1,'Sports Car')
```

```
//Insert BMW,2,Sports Car;Audi,4,Truck; Audi,5,Hatchback
```

Select Operation

- Very similar to SQL
- Command to select particular data:

`'Select * from employee_by_id where id=1;' //Work`

`'Select * from employee_by_id where name='John';'`
`//Will not work because name is not part of primary key`

- **Can just query on primary key columns** as data is distributed here.
- Such queries though can be supported will be very unperformed queries.

Select Operation

- Command to select and order particular data:

```
'Select * from employee_by_car_make where id=1;'
```

//Will not work as id is not part of partitioning key

```
'Select * from employee_by_car_make where car_make='BMW'  
Order by id;'
```

//Work as id is part of clustering column

- **Where** clause can be used with **partitioning keys** and **Order by** can only be performed on **clustering columns**.
- Can order data on multiple clustering columns, but we must perform ordering as specified in schema. (i.e. on previous column first and then order data on next column.)

Upsert(Insert or update)

- [employee_by_car_make_and_model : ((car_make, car_model), id)]
- 'Insert into employee_by_car_make_and_model (car_make, car_model, id, name) values ('BMW', 'Hatchback', 1, 'Ram');' //Works
- 'Insert into employee_by_car_make_and_model (car_make, id, name) values ('BMW', 1, 'Ram');' //will not work
- 'Insert into employee_by_car_make_and_model (car_make, car_model, name) values ('BMW', 'Sports Car', 'Ram');' //will not work.
- Compulsory to include values for primary key columns, other column values can be omitted.

```
'Insert into employee_by_car_make_and_model ( car_make, car_model, id,  
    name) values ('BMW', 'Hatchback', 1,'Raj');'  
//Works, Replaces 'Ram' with 'Raj'
```

```
'Insert into employee_by_car_make_and_model ( car_make, car_model, id)  
    values ('BMW', 'Hatchback', 1,);'  
// Works, but will not insert anything in table as this data is already present  
there
```

```
'Insert into employee_by_car_make_and_model ( car_make, car_model, id)  
    values ('BMW', 'Hatchback',2);'  
// Works, will insert this record in table
```

- Though cassandra displays 'null' for the column where data is not inserted, this is just for display purpose and it does not store null actually.

Insert some more rows in employee_by_car_make_and_model:

car_make	car_model	id	name
BMW	HATCHBACK	1	Bob
BMW	HATCHBACK	2	null
AUDI	HATCHBACK	3	null
BMW	TRUCK	8	FRANK
AUDI	TRUCK	7	AMY
AUDI	SPORTS CAR	4	TIM
AUDI	SPORTS CAR	5	JIM
AUDI	SPORTS CAR	6	NICK

'Select * from employee_by_car_make_and_model where car_make='BMW';' //Will not work as two columns have been specified in partition keys, cassandra does not know which node to go to fetch the data

'Select * from employee_by_car_make_and_model where car_make='BMW' and car_model='Hatchback';' //Works

Timestamps

- Can check the timestamp of the last write operation.

`'Select car_make, car_model, writetime
(car_model) from employee_by_car_make;'`

Update Operation

‘Update employee_by_car_make Set
car_model=‘TRUCK’ where car_make=‘BMW’
and id = 1;

- Updates specific row/s, both primary key columns has to be specified.

TTL – Time to live

- If TTL is specified for a certain column value, it means that that data is valid for only particular time period.
- Useful in circumstances like a user, when visits a website, is issued a validity token for 24 hours only, which then should expire.

‘Update employee_by_car_make Using TTL 60 Set car_model = ‘TRUCK’ Where car_make=‘BMW’ and id=2;’
// Should expire after 60 seconds and can see null after 60s.

Collection DataType



Collections are meant for storing/denormalizing relatively small amount of data. They work well for things like “the phone numbers of a given user”, “labels applied to an email”, etc.

1. Map
2. set
3. List

Maps

```
CREATE TABLE users (  
  id text PRIMARY KEY,  
  name text,  
  favs map<text, text> // A map of text keys, and text values  
);
```

```
INSERT INTO users (id, name, favs)  
  VALUES ('jsmith', 'John Smith', { 'fruit': 'Apple', 'band': 'Beatles' });  
// Replace the existing map entirely.  
UPDATE users SET favs = { 'fruit': 'Banana' } WHERE id = 'jsmith';
```

A **map** is a (sorted) set of key-value pairs, where **keys are unique** and the map is **sorted by its keys**. You can define and insert a map with:

```
token@cqlsh:mykeyspace> select * from users;
```

id	favs	name
jsmith	{ 'band': 'Beatles', 'fruit': 'Apple' }	John Smith

```
(1 rows)
```

Operations with Map

- **Updating or inserting one or more elements:**

```
UPDATE users SET favs['author'] = 'Ed Poe' WHERE id = 'jsmith';
```

```
UPDATE users SET favs = favs + { 'movie' : 'Cassablanca', 'band' : 'ZZ Top' } WHERE  
id = 'jsmith';
```

- **Removing one or more element (if an element doesn't exist, removing it is a no-op but no error is thrown):**

```
DELETE favs['author'] FROM users WHERE id = 'jsmith';
```

```
UPDATE users SET favs = favs - { 'movie', 'band' } WHERE id = 'jsmith';
```

Sets

```
CREATE TABLE images (  
  name text PRIMARY KEY,  
  owner text,  
  tags set<text> // A set of text values  
);
```

```
INSERT INTO images (name, owner, tags)  
VALUES ('cat.jpg', 'jsmith', { 'pet', 'cute' });
```

// Replace the existing set entirely

```
UPDATE images SET tags = { 'kitten', 'cat', 'lol' } WHERE name = 'cat.jpg';
```

A **set** is a (sorted) collection of unique values.

name	owner	tags
cat.jpg	jsmith	{ 'cute', 'pet' }

name	owner	tags
cat.jpg	jsmith	{ 'cat', 'kitten', 'lol' }

Operations with Set

- **adding one or multiple elements (as this is a set, inserting an already existing element is a no-op):**

```
UPDATE images SET tags = tags + { 'gray', 'cuddly' } WHERE name =  
'cat.jpg';
```

- **Removing one or multiple elements (if an element doesn't exist, removing it is a no-op but no error is thrown):**

```
UPDATE images SET tags = tags - { 'cat' } WHERE name = 'cat.jpg';
```

Lists

```
CREATE TABLE plays (  
  id text PRIMARY KEY,  
  game text,  
  players int,  
  scores list<int> // A list of integers  
)
```

```
INSERT INTO plays (id, game, players, scores)  
VALUES ('123-afde', 'quake', 3, [17, 4, 2]);
```

```
// Replace the existing list entirely  
UPDATE plays SET scores = [ 3, 9, 4] WHERE id = '123-afde';
```

A list is a (sorted) collection of non-unique values where elements are ordered by their position in the list.

id	game	players	scores
123-afde	quake	3	[17, 4, 2]

id	game	players	scores
123-afde	quake	3	[3, 9, 4]

Operations with List

- **Appending and prepending values to a list:**

UPDATE plays SET players = 5, scores = scores + [14, 21] WHERE id = '123-afde';

UPDATE plays SET players = 6, scores = [3] + scores WHERE id = '123-afde';

- **Setting the value at a particular position in a list that has a pre-existing element for that position. An error will be thrown if the list does not have the position.:**

UPDATE plays SET scores[1] = 7 WHERE id = '123-afde';

Operations with List

- **Removing an element by its position in the list that has a pre-existing element for that position. An error will be thrown if the list does not have the position. Further, as the operation removes an element from the list, the list size will decrease by one element, shifting the position of all the following elements one forward:**

`DELETE scores[1] FROM plays WHERE id = '123-afde';`

- **Deleting *all* the occurrences of particular values in the list (if a particular element doesn't occur at all in the list, it is simply ignored and no error is thrown):**
- `UPDATE plays SET scores = scores - [12, 21] WHERE id = '123-afde';`

CREATE MATERIALIZED VIEW

```
CREATE MATERIALIZED VIEW monkeySpecies_by_population AS  
  SELECT * FROM monkeySpecies  
  WHERE population IS NOT NULL AND species IS NOT NULL  
  PRIMARY KEY (population, species)  
  WITH comment='Allow query by population instead of species';
```


References

17

1. https://cassandra.apache.org/_/cassandra-basics.html
2. <https://cassandra.apache.org/doc/latest/cassandra/cql/index.html>