

Apache Cassandra



Column Stores

- Usage: read/write extensions
- Popular DBs: HBase, Cassandra



Document Store

- Usage: working with occasionally changing/consistent data
- Popular DBs: Couchbase, MongoDB



Graph Databases

- Usage: spatial data storage
- Popular DBs: Neo4J, Bigdata

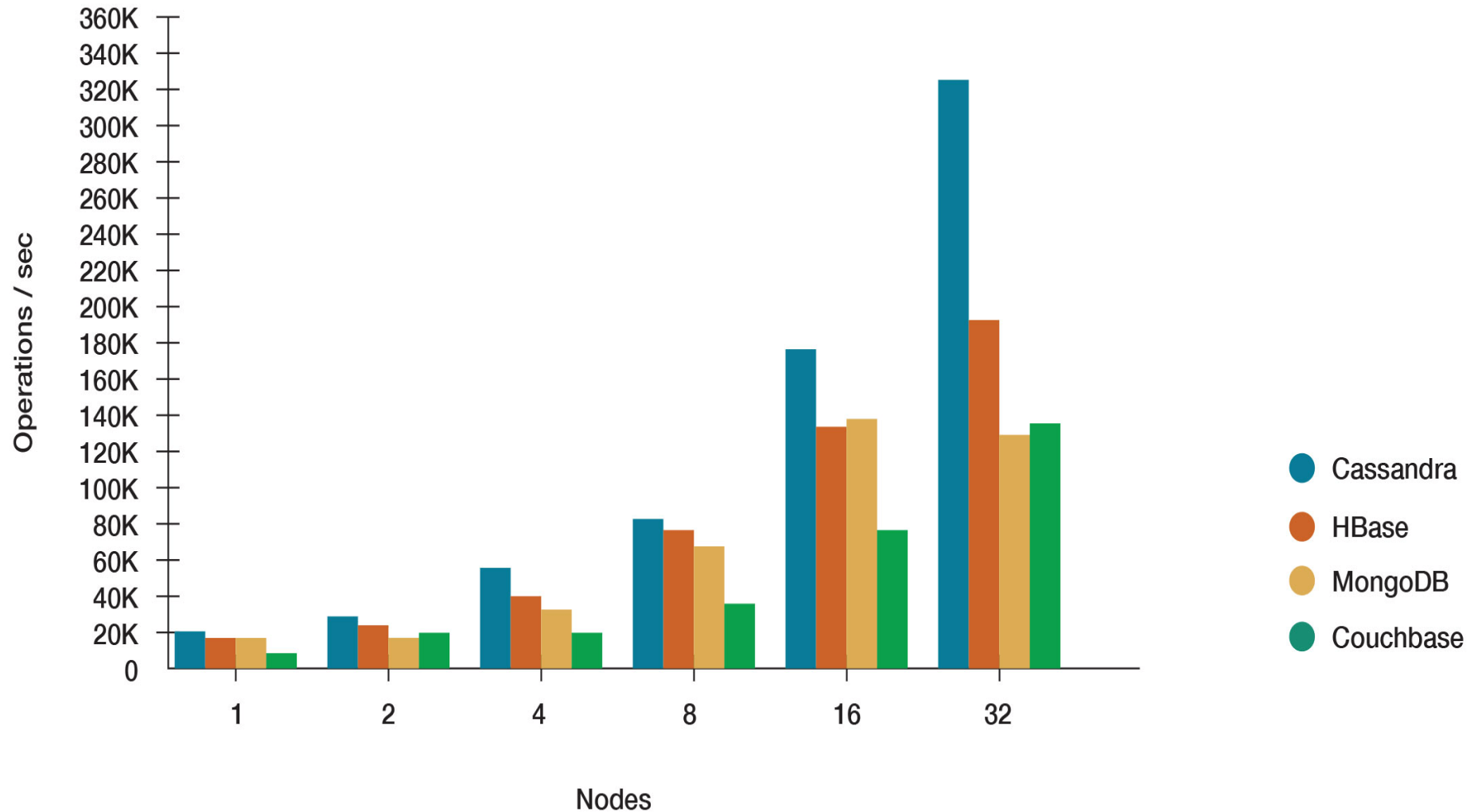


Key/Value

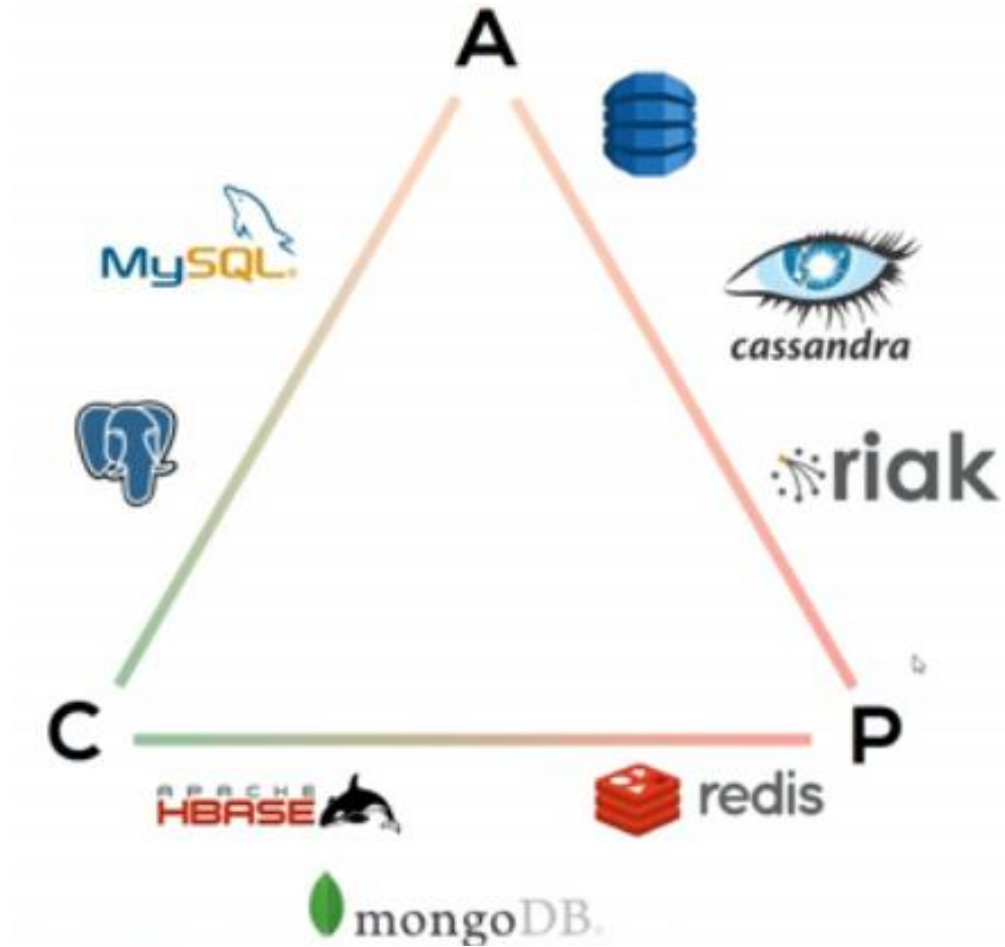
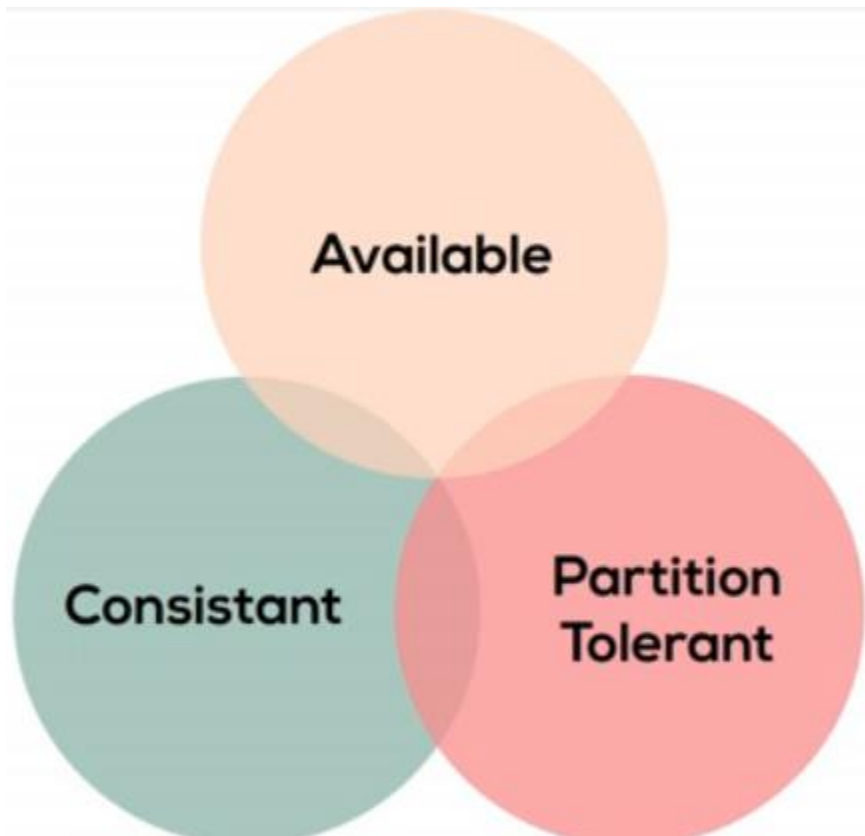
- Usage: briskly changing data and high availability
- Popular DBs: Riak, Redis

NoSQL Databases

Comparison



CAP's Theorem



Cassandra - Introduction

- Born at Facebook (Avinash Lakshman and Prashant Malik).
- After Facebook open sourced the code in 2008, it became an Apache Incubator project in 2009 and subsequently became a top-level Apache project in 2010.
- Built on Amazon's dynamo(Distributed Storage and replication techniques) and Google's BigTable(Data and storage engine model).
- Column-oriented database designed to support peer-to-peer symmetric nodes instead of master-slave architecture

Google

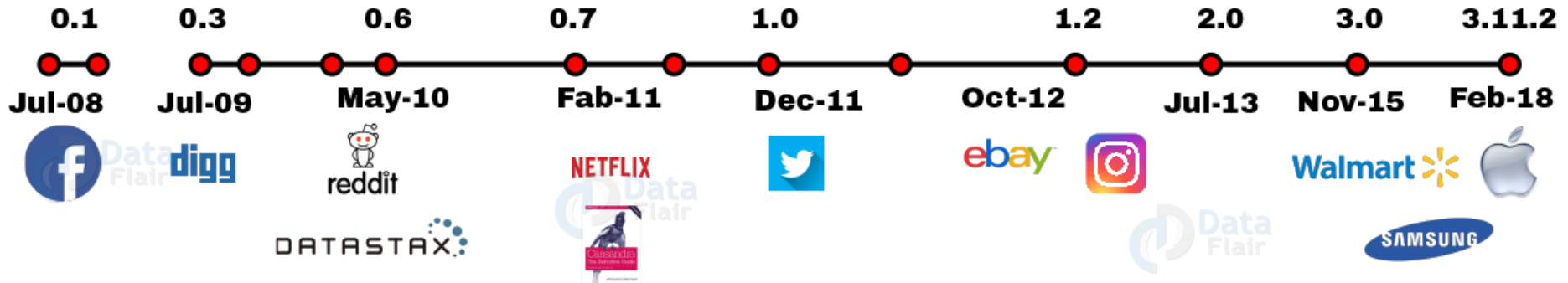
Bigtable, 2006

amazon.com

Dynamo, 2007

facebook.

OpenSource, 2008



Design Objectives

- 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



Open Source

**Elastic
Scalability**

**High Availability
and
Fault Tolerance**

**Peer to
Peer Architecture**



Cassandra

FEATURES

**High
Performance**

**Column
Oriented**

**Tuneable
Consistency**

Schema-Free

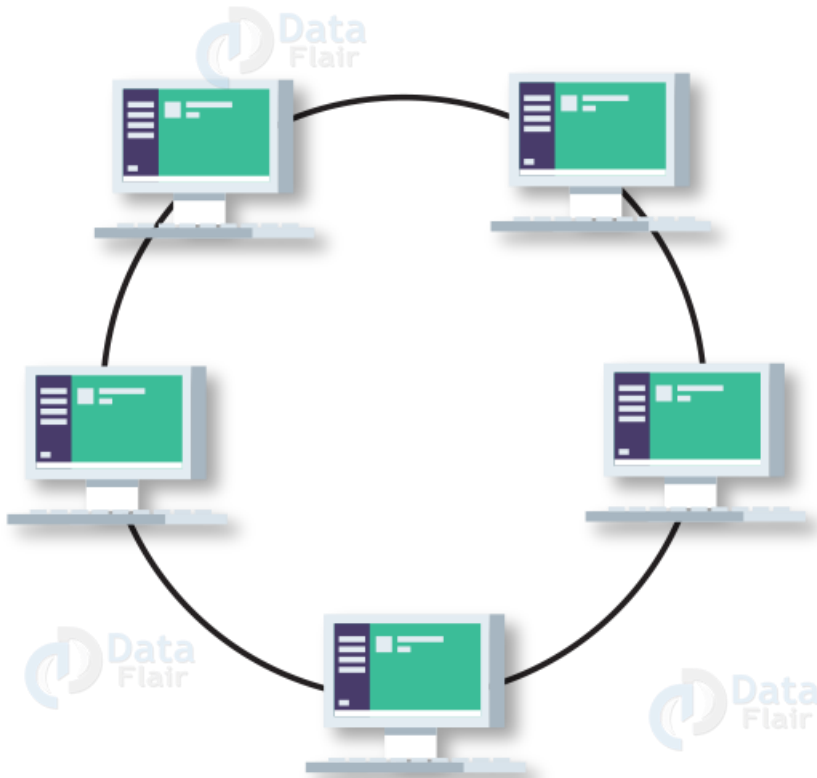
Open Source



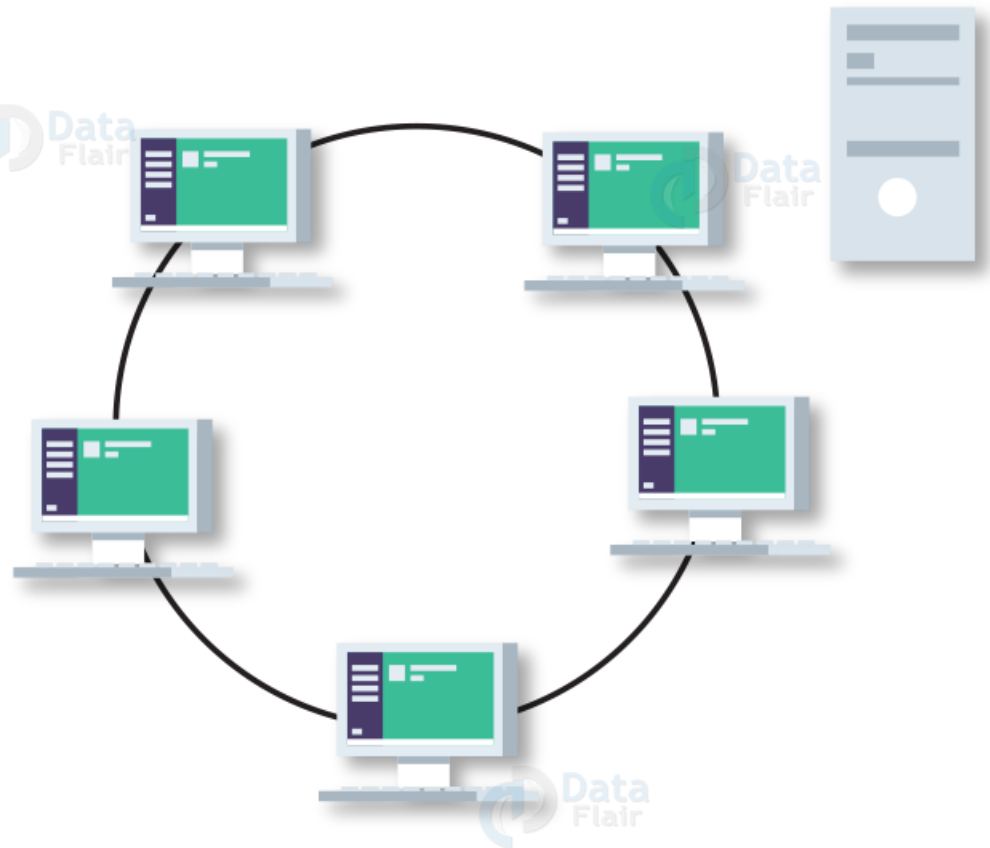
- Cassandra, though it is very powerful and reliable, is FREE!
- It is an open source project by Apache.
- Because of the open source feature, it gave birth to a huge Cassandra Community, where people discuss their queries and views.
- Possibility of integrating Cassandra with other Apache Open-source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

Peer-to-Peer Network

Architecture



peer to peer

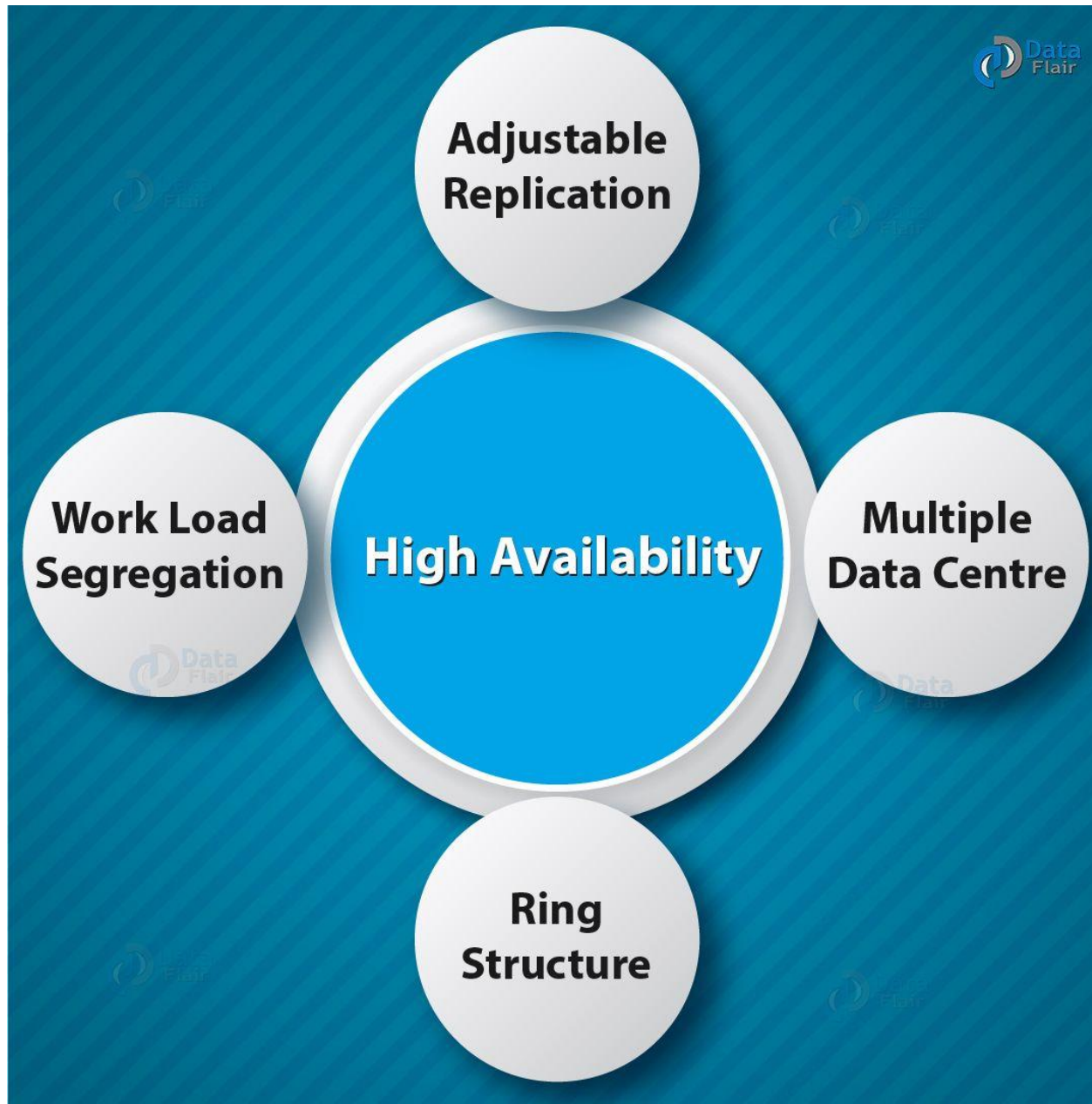


client - server

Elastic Scalability

- Can 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.
- Because of this, Cassandra has a very high throughput for the highest number of nodes.
- Moreover, there is zero downtime or any pause during scaling. Hence read and write throughput increases simultaneously without delay.

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

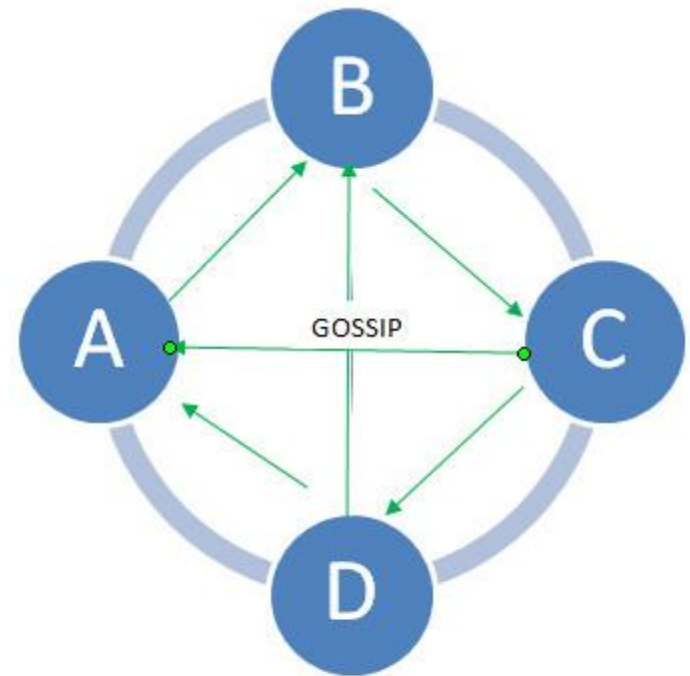
- Cassandra database has one of the best performance as compared to other NoSQL database.
- Developers wanted to utilize the capabilities of many multi-core machines. This is the base of development of Cassandra.
- Cassandra has proven itself to be excellently reliable when it comes to a large set of 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.

Gossip Protocol

- Spreads like virus – Most efficient way of spreading information in large cluster
- One node infects few nodes, which in turn infects few more nodes and the process goes on and on.
- It is not synchronous.

Gossip and Failure Detection

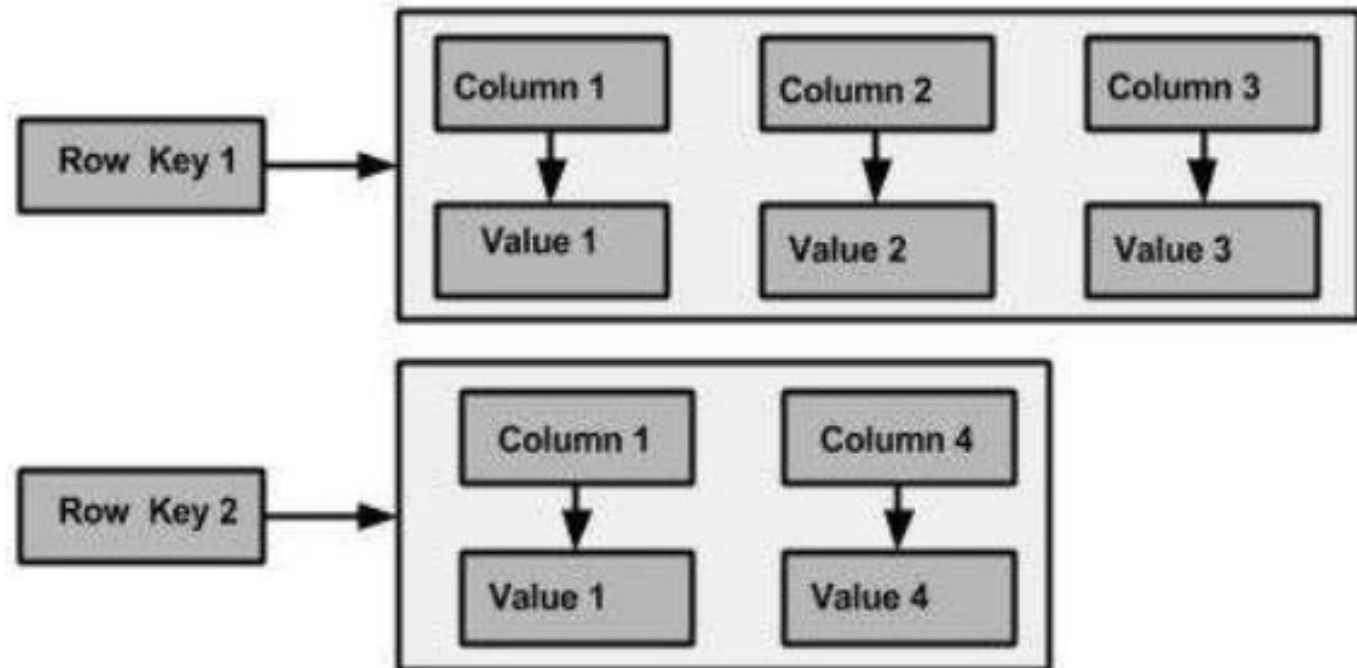
- Gossip is used for intra-ring communication
- The gossip process runs every second for every node and exchange state messages with up to three other nodes in the cluster.
- Each node independently will always select one to three peers to gossip with. It will always select a live peer (if any) in the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



PEER TO PEER DISTRIBUTION
MODEL OF CASSANDRA

Column-Family Database

- Cassandra's data model is column-oriented.
- In other databases, column name contains metadata, whereas, columns in Cassandra also contain actual data.
- In Cassandra, columns are stored based on column names.
- We use row key and column family name to address a column family.



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}		

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



- Two types of consistency in Cassandra:
 - Eventual consistency and Strong Consistency.
- 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.

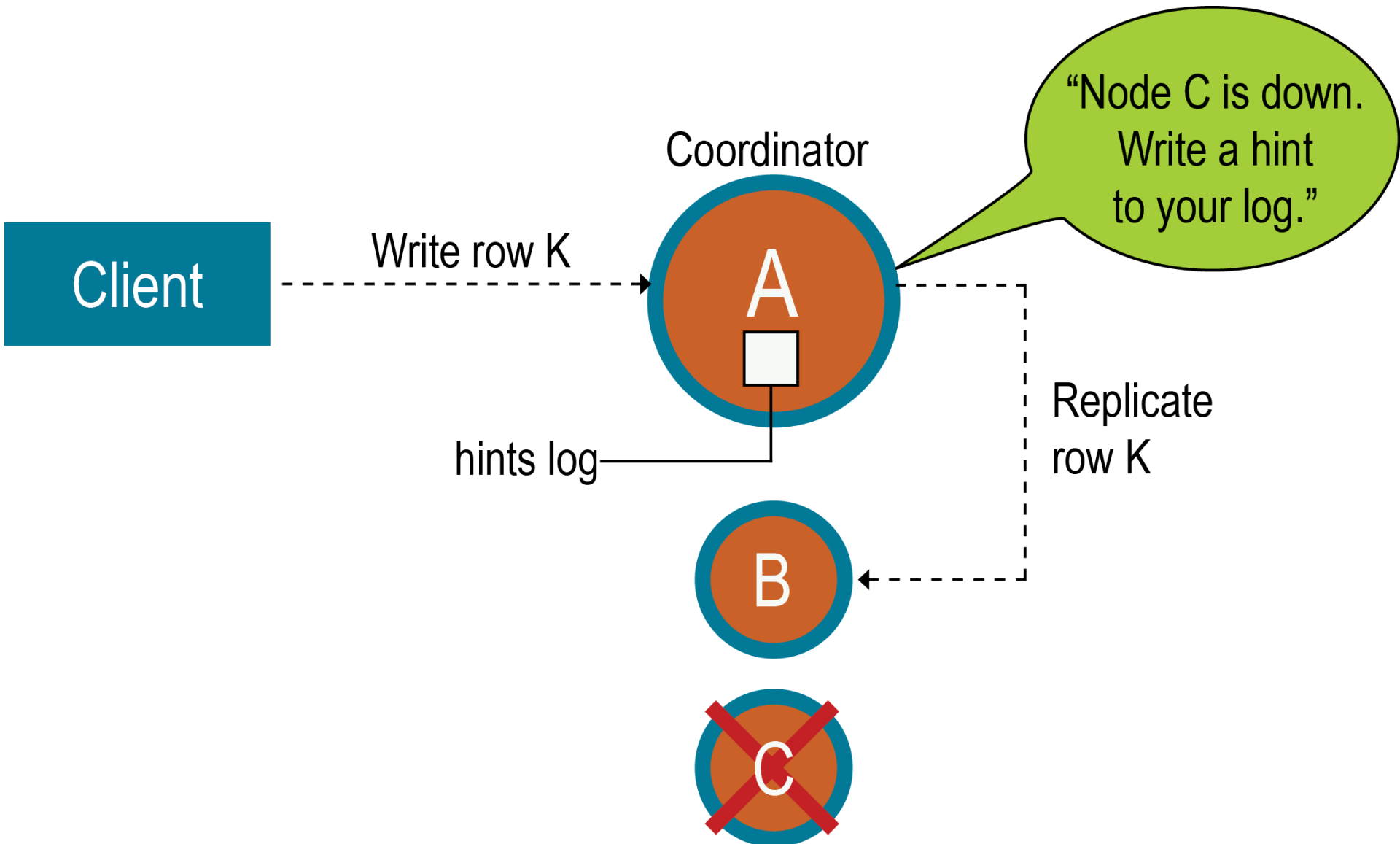
Anti-Entropy and Read Repair

- 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.

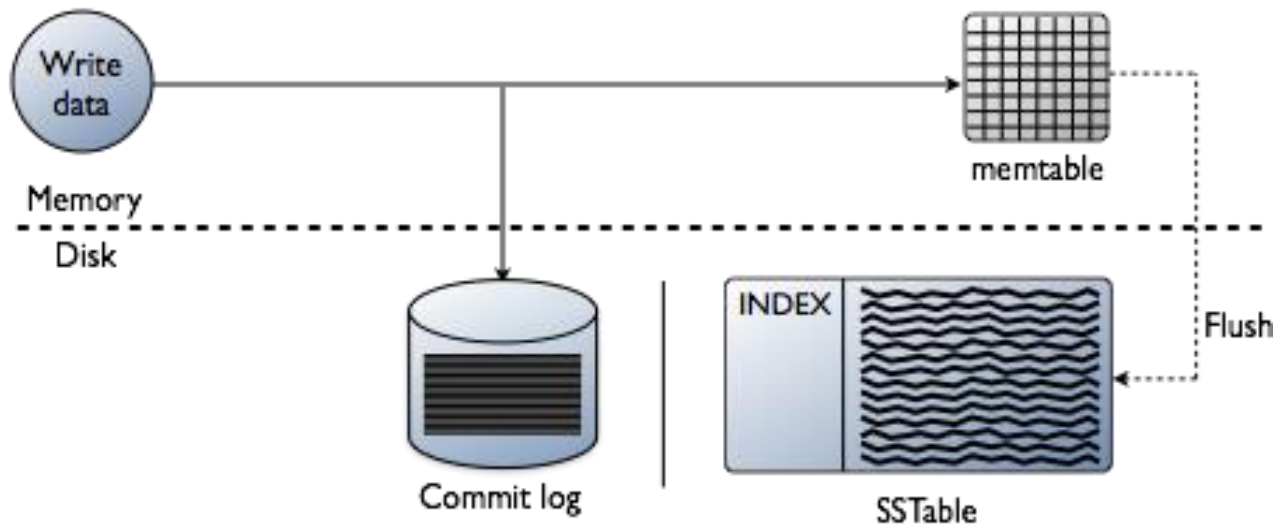
Hinted Handoffs

- Cassandra works on philosophy that it should be always available for writes.
- This is achieved by hinted handoffs.
- Assume
 - A cluster of three nodes – Node A, Node B and Node C.
 - Node C is down for some reason
 - Replication factor is 2
- Hint contains following information:
 - Location of the node on which the replica is to be placed
 - Version metadata
 - The actual data

Hinted Handoffs



Writes in Cassandra



- A write operation once completed is first written to commit log.
- Next it is pushed to a memory resident data structure called Memtable, which has a predefined threshold value.
- Only when write is completed in memtable and commitlog, a node responds successful message to coordinator.
- When no. of objects reaches a threshold, then the contents of Memtable are flushed to the disk in a file called SSTable(Sorted String Table)
- Flushing is a non-blocking operation.

Demonstration with Example

Que: Can Cassandra consistency level be higher than the replication factor?

Ans: We **can't** have a **consistency level higher than the replication factor** simply because you **can't** expect **more** nodes to answer to a request **than** the amount of nodes holding the data.

Cassandra Read Path

1. Bloom Filter : It tells us either our data is 100% not available in the SSTable, or it is present with high probability. (It is possible that we might end up scanning entire table and we do not get required data, but mostly the probability of availability is quite high.)
2. Key Cache: Stores byte location of frequently accessed data. E.g. India 2000.

Cassandra Read Path

3. Partition Summary:

- Helps when partition index grows so large that it becomes time consuming to even scan through partition index.
- It groups partition index entries into some groups. E.g. Germany, India, USA in one group.

4. Partition Index:

- Maintains index of all parts of our data. E.g. Germany 0, India 2000, USA 4500, Japan 5500, etc...

Cassandra Read Path

5. SSTable :

- We should never have to scan through SSTable a lot.
- There must be some mechanism in partition index to reach to required data easily.

- Row Cache :

- Caches frequently accessed rows.
- Though it occupies memory, it speeds up read operation.
- Disabled by default.

There are some other caches too like counter cache.

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

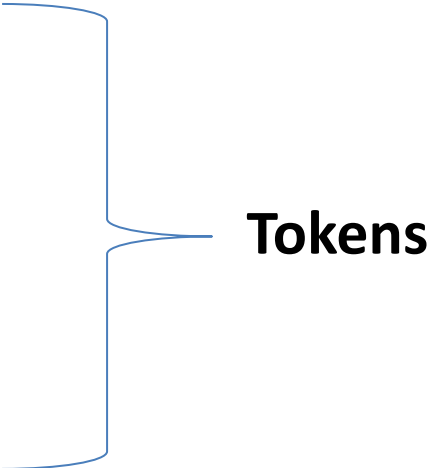
Employee By Car Make				
BMW*3	{Id : 1}	{Firstname : Elvis}	{Surname : Presley}	{Salary : \$30000}
	{Id : 4}	{Firstname : Elton}	{Surname : John}	{Salary : \$20000}
	{Id : 7}	{Firstname : Selena}	{Surname : Gomez}	{Salary : \$30000}
Audi*2	{Id : 2}	{Firstname : David}		{Salary : \$40000}
	{Id : 5}	{Firstname : 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

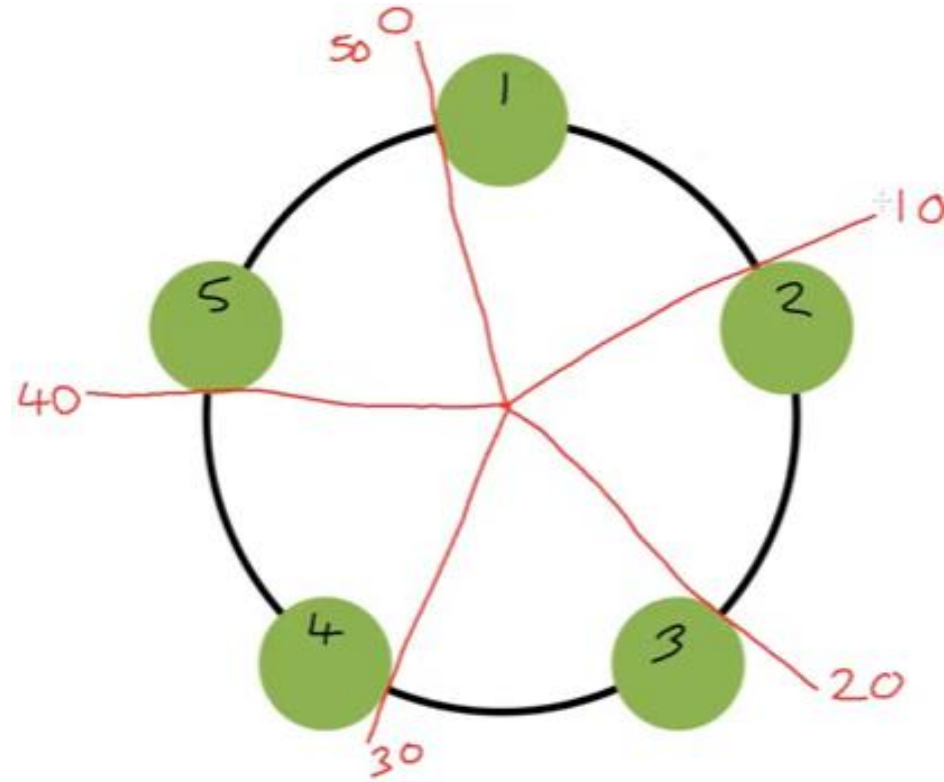
- Data belonging to same partition will be written on same node.
- In cassandra, data is accessed by partition keys and not by primary key though primary key may be part of the table.
- E.g. For Employee Table – Car Make, For Car table – Car ID
- Hash function is used for creating partitioning

Example

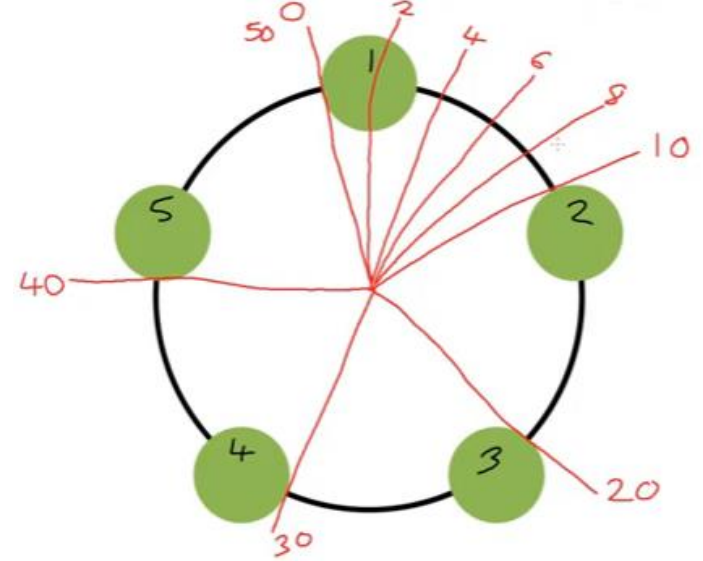
- BMW -> Hash Function -> 15
 - Audi -> Hash Function -> 22
 - Merc -> Hash Function -> 43
 - BMW -> Hash Function -> 15
- 
- Tokens
- Tokens are of 64 bit integers in cassandra.
 - Range: -2^{64} to $2^{63} - 1$

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 .



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.
 - E.g. Node having 256 GB of storage can be assigned 8 virtual nodes and node having 64 GB of storage can be assigned 2 virtual nodes.
- Default value in cassandra is 256 virtual nodes

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.
- The hierarchy of elements in Cassandra is:
 - Cluster
 - Data center(s)
 - Rack(s)
 - Server(s)
 - Node (more accurately, a vnode)

Demonstration

- Start Cassandra service
- Go to bin folder of cassandra
- Execute './cassandra -f'
- Open new tab
- Execute 'nodetool status'
- 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)

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 or etc/cassandra directory of 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

Strategies

- SimpleStrategy :
 - Just move around the ring to meet the replication factor
 - Stop once replication factor is met.
- NetworkTopologyStrategy:
 - Complicated
 - Allows to specify different replication factor to different data centers
 - Inside a datacenter, it stores replicas on different racks.

Keyspaces

- Can have one or more keyspaces across cassandra cluster
- Keyspaces are containers for data.
- E.g. We may have one cassandra cluster across the company and can have one keyspace for each sub-company.
- A keyspace can contain many tables, a table contains many rows which contains many key-value pair columns.

Create KEYSPACE

- After starting cassandra service, type 'cqlsh'.
- 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.

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;’

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

- Partitioning key: column on which data is partitioned
- Clustering key: column on which data is ordered
- 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.

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.)

- [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.