GROW DATA SKILLS

cassandra

What is NoSQL Database?

NoSQL databases, also known as "**non-SQL**" or "**not only SQL**", are databases that provide a mechanism to store and retrieve data modeled in ways other than the tabular format used in relational databases. They are typically used in large-scale or real-time web applications where the ability to scale quickly and handle large, diverse types of data is critical.

Here are some key characteristics and features of NoSQL databases:

1. *Schema-less*: NoSQL databases do not require a fixed schema, which gives you the flexibility to store different types of data entities together.

2. *Scalability*: NoSQL databases are designed to expand easily to handle more traffic. They are horizontally scalable, meaning you can add more servers to handle larger amounts of data and higher loads.

3. *Diverse Data Models*: NoSQL databases support a variety of data models including key-value pairs, wide-column, graph, or document. This flexibility allows them to handle diverse types of data and complex data structures.

4. *Distributed Architecture*: Many NoSQL databases are designed with a distributed architecture, which can improve fault tolerance and data availability.

5. *Performance*: Without the need for data relationships and joins as in relational databases, NoSQL data offer high-performance reads and writes.

Types of NoSQL Databases

NoSQL databases are categorized into four basic types based on the way they organize data. Let's go through each type and provide examples:

Document Databases: These store data in documents similar to JSON (JavaScript Object Notation) objects. Each document contains pairs of fields and values, and the values can typically be a variety of types including strings, numbers, booleans, arrays, or objects. Each document is unique and can contain different data from other documents in the collection. This structure makes document databases flexible and adaptable to various data models.

Example: MongoDB, CouchDB.

Key-Value Databases: These are the simplest type of NoSQL databases. Every single item in the database is stored as an attribute name (or 'key') and its value. The main advantage of a key-value store is the ability to read and write operations using a simple key. This type of NoSQL database is typically used for caching and session management.

Example: Redis, Amazon DynamoDB.

Types of NoSQL Databases

Wide-Column Stores: These databases store data in tables, rows, and dynamic columns. Wide-column stores offer high performance and a highly scalable architecture. They're ideal for analyzing large datasets and are capable of storing vast amounts of data (Big Data).

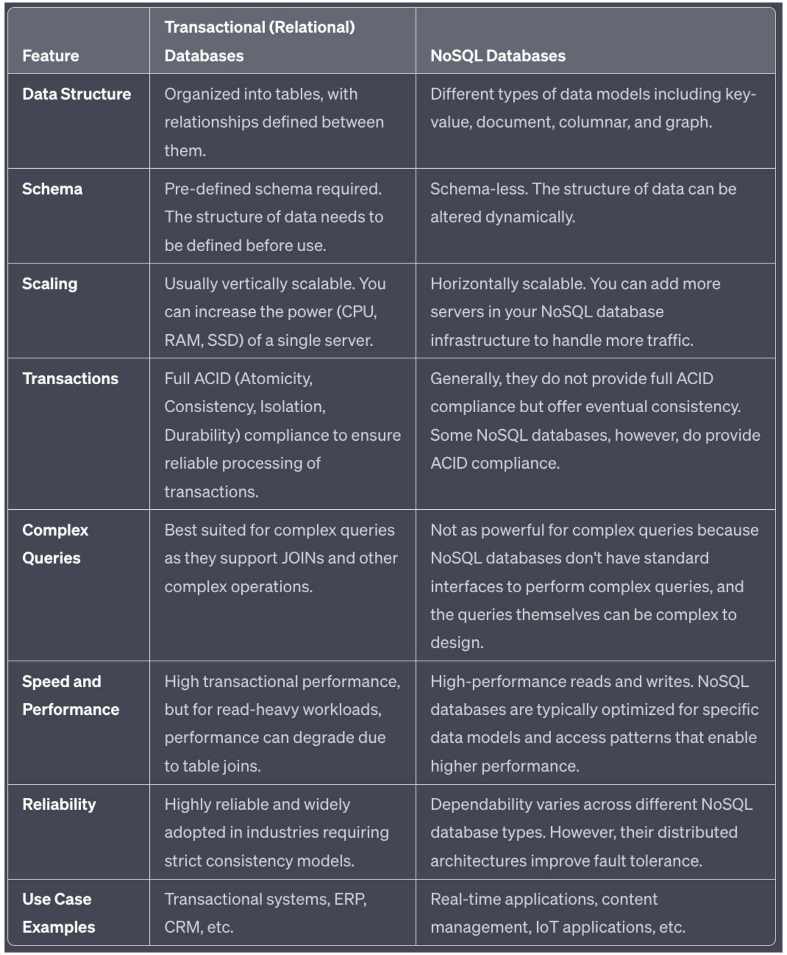
Example: Apache Cassandra, Google BigTable.

Graph Databases: These are used to store data whose relations are best represented as a graph. Each node of the graph represents an entity, and the relationship between nodes is stored directly, which allows the data to be retrieved in one operation. They're ideal for storing data with complex relationships, like social networks or a network of IoT devices.

Example: Neo4j, Amazon Neptune

Difference between Transactional & NoSQL Database

GROW DATA SKILLS



Feature	Transactional (Relational) Databases	NoSQL Databases
Data Structure	Organized into tables, with relationships defined between them.	Different types of data models including key-value, document, columnar, and graph.
Schema	Pre-defined schema required. The structure of data needs to be defined before use.	Schema-less. The structure of data can be altered dynamically.
Scaling	Usually vertically scalable. You can increase the power (CPU, RAM, SSD) of a single server.	Horizontally scalable. You can add more servers in your NoSQL database infrastructure to handle more traffic.
Transactions	Full ACID (Atomicity, Consistency, Isolation, Durability) compliance to ensure reliable processing of transactions.	Generally, they do not provide full ACID compliance but offer eventual consistency. Some NoSQL databases, however, do provide ACID compliance.
Complex Queries	Best suited for complex queries as they support JOINs and other complex operations.	Not as powerful for complex queries because NoSQL databases don't have standard interfaces to perform complex queries, and the queries themselves can be complex to design.
Speed and Performance	High transactional performance, but for read-heavy workloads, performance can degrade due to table joins.	High-performance reads and writes. NoSQL databases are typically optimized for specific data models and access patterns that enable higher performance.
Reliability	Highly reliable and widely adopted in industries requiring strict consistency models.	Dependability varies across different NoSQL database types. However, their distributed architectures improve fault tolerance.
Use Case Examples	Transactional systems, ERP, CRM, etc.	Real-time applications, content management, IoT applications, etc.

NoSQL Databases are Good fit for Analytical Queries?

While NoSQL databases can handle certain analytical tasks, their primary purpose is not for heavy analytical queries. Traditional relational databases and data warehousing solutions, such as Hive, Redshift, Snowflake or BigQuery, are often better suited for complex analytical queries due to their ability to handle operations like joins and aggregations more efficiently.

NoSQL Databases in BigData Ecosystem

The strength of NoSQL databases lies in their flexibility, scalability, and speed for certain types of workloads, making them ideal for specific use-cases in the Big Data ecosystem:

1. **Handling Large Volumes of Data at Speed:** NoSQL databases are designed to scale horizontally across many servers, which enables them to handle large volumes of data at high speed. This is particularly useful for applications that need real-time read/write operations on Big Data.

2. **Variety of Data Formats**: NoSQL databases can handle a wide variety of data types (structured, semi-structured, unstructured), making them ideal for Big Data scenarios where data formats are diverse and evolving.

3. **Fault Tolerance and Geographic Distribution**: NoSQL databases have a distributed architecture that provides high availability and fault tolerance, critical for applications operating on Big Data.

4. **Real-time Applications**: Many Big Data applications require real-time or near-real-time functionality. NoSQL databases, with their high-speed read/write capabilities and ability to handle high volumes of data, are often used for real-time analytics, IoT data, and other similar scenarios.

That said, the choice between SQL, NoSQL, or other database technologies should be based on the specific needs of the use-case at hand.

CAP Theorem

The CAP theorem is a concept that a distributed computing system is unable to simultaneously provide all three of the following guarantees:

1. **Consistency (C):** Every read from the system receives the most recent write or an error. This implies that all nodes see the same data at the same time. It's the idea that you're always reading fresh data.

2. **Availability (A):** Every request receives a (non-error) response, without the guarantee that it contains the most recent write. It's the idea that you can always read or write data, even if it's not the most current data.

3. **Partition Tolerance (P):** The system continues to operate despite an arbitrary number of network or message failures (dropped, delayed, scrambled messages). It's the idea that the system continues to function even when network failures occur between nodes.

CAP Theorem

Now, the key aspect of the CAP theorem, proposed by computer scientist Eric Brewer, is that a distributed system can satisfy any two of these three guarantees at the same time, but not all three. Hence the term "CAP" - Consistency, Availability, and Partition tolerance.

Here's how the three dichotomies look like:

- **CA (Consistent and Available)** systems prioritize data consistency and system availability but cannot tolerate network partitions. In such a system, if there is a partition between nodes, the system won't work as it doesn't support partition tolerance.

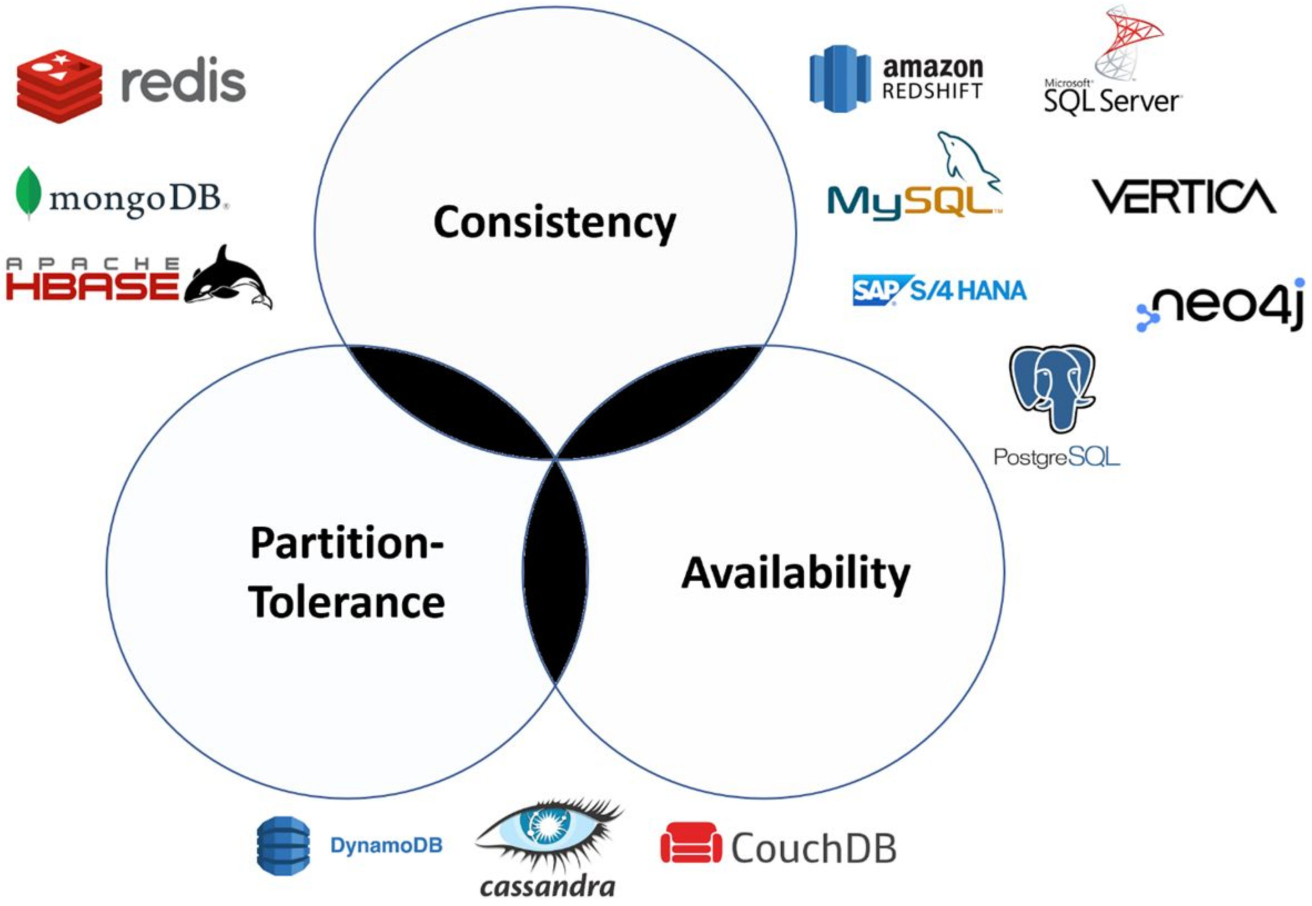
- **CP (Consistent and Partition-tolerant)** systems prioritize data consistency and partition tolerance. If a network partition occurs, the system sacrifices availability to ensure data consistency across all nodes.

- **AP (Available and Partition-tolerant)** systems prioritize system availability and partition tolerance. If a network partition occurs, all nodes may not immediately reflect the same data, but the system remains available.

Remember, real-world systems must tolerate network partitions (P), so the practical choice is between consis
and availability (A) when partitions occur.

CAP Theorem

GROW DATA SKILLS



Cassandra

- **Open-Source Database System:** Cassandra is an open-source distributed database management system designed to handle large amounts of data across many commodity servers.

- **Wide-Column Store:** Cassandra falls under the wide-column store category of NoSQL databases. It arranges data by columns, allowing for great flexibility and scalability.

- **Highly Scalable**: Cassandra is designed to handle big data through its distributed architecture. It can scale horizontally to accommodate more data by simply adding more nodes to the system.

- **Fault-Tolerant:** It provides high availability and fault tolerance with no single point of failure. If a node fails, data can still be accessed from other nodes.

- **Decentralized**: Every node in the system has the same role with no master-slave dichotomy. This contributes to its robustness and resilience.

- **CAP Theorem**: In the context of CAP Theorem, Cassandra prioritizes Availability and Partition tolerance (AP) over Consistency. It provides eventual consistency.

- **Use Cases**: It's often used in applications where large data volumes are expected and a highly scalable, reliable system is required, such as in IoT, messaging systems, and fraud detection systems.

Cassandra is a good fit for??

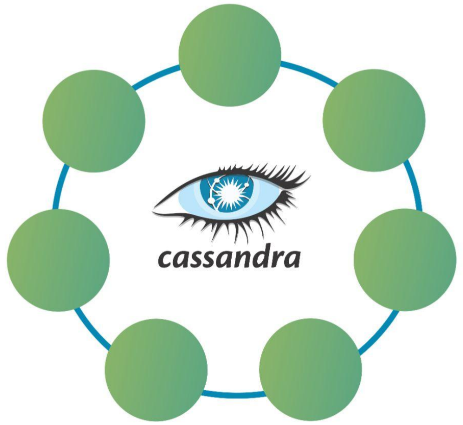
Apache Cassandra is ideal for scenarios that need high availability, scalability, and quick data processing. It's best for handling **time-series** data, powering **messaging** systems, managing product **catalogs** and **playlists**, supporting **recommendation** systems, and running **fraud detection** systems.

Cassandra doesn’t support JOINS

Cassandra doesn't support JOIN operations or other complex queries that relational databases can handle. JOIN operations typically involve merging data from multiple tables, which can be computationally expensive and slow, especially with large volumes of data. This goes against Cassandra's goal of providing high-speed, real-time performance.

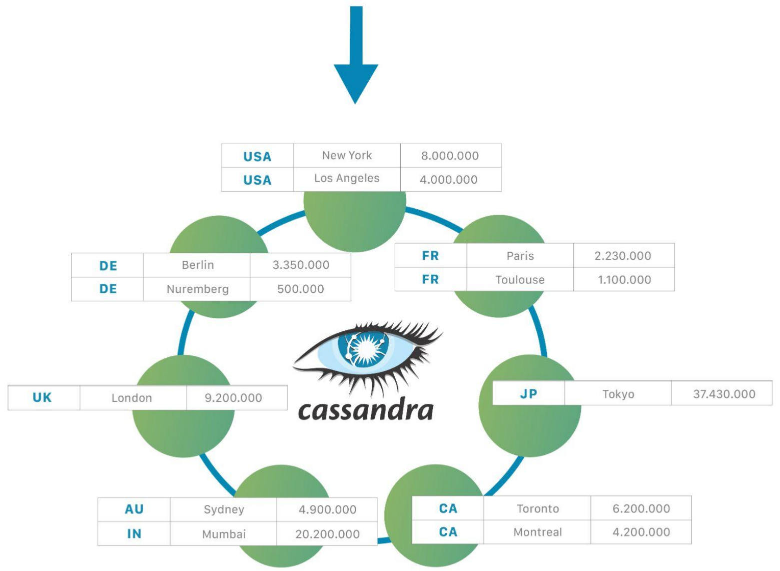
Cassandra Architecture

GROW DATA SKILLS



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



USA	New York	8.000.000
USA	Los Angeles	4.000.000

DE	Berlin	3.350.000
DE	Nuremberg	500.000

FR	Paris	2.230.000
FR	Toulouse	1.100.000

UK	London	9.200.000

JP	Tokyo	37.430.000

AU	Sydney	4.900.000
IN	Mumbai	20.200.000

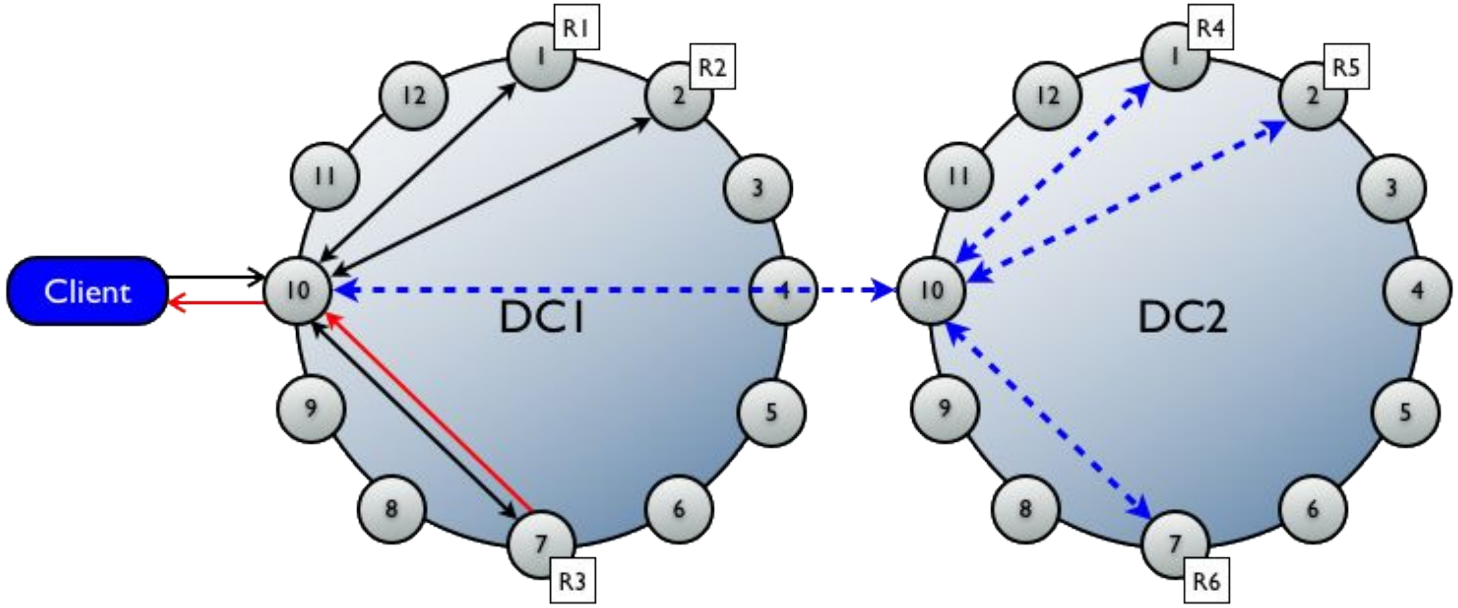
CA	Toronto	6.200.000
CA	Montreal	4.200.000

Features of Cassandra

- No single point of failure.
- Massive scalability .
- Possibility to add nodes any time.
- Highly distributed.
- High performance.
- No masters and slaves (Peer to peer).
- Ring type architecture
- Automatic data distribution across all nodes
- Replication of data across nodes
- Data kept in memory and written to disk in a lazy fashion
- Hash values of the keys are used to distribute data among nodes

Main Components of Cassandra

- **Node** − It is the place where data is stored.
- **Data center** − It is a collection of related nodes.
- **Cluster** − A cluster is a component that contains one or more data centers.



Data Partitioning & Token

Cassandra organizes data into partitions. This is a common concept of distributed data systems. All the data is distributed into chucks called partitions. Partitioning is very important for performance and scalability.

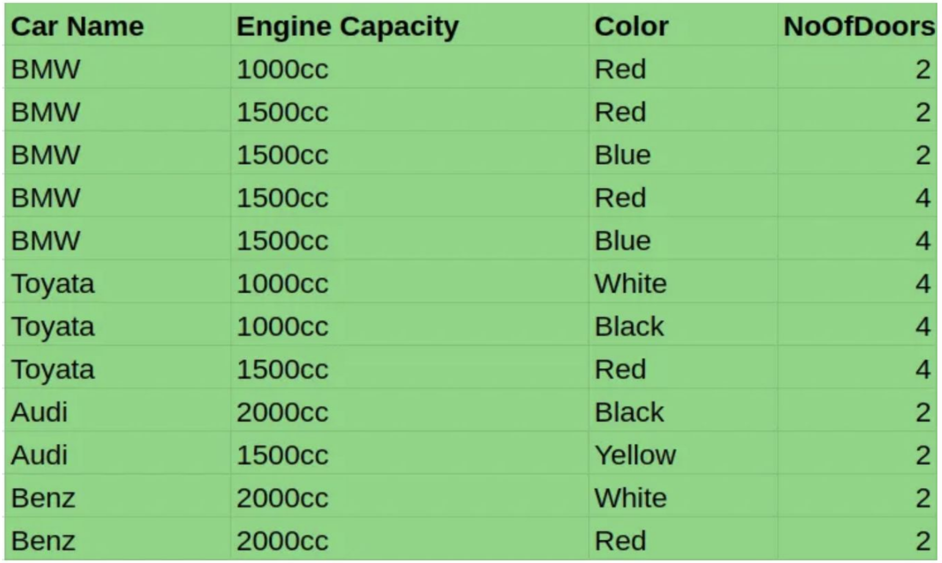
- **Partition Key:** The partition key is responsible for distributing data among nodes. In Cassandra, we can only access data from the partitioning key.

- **Token**: A token in Cassandra is essentially a hash value. When data is written to Cassandra, it is hashed based on its partition key to produce a token value. This token value determines where the data should be stored. In this way, each piece of data is assigned a specific token. By default, a token is a **64-bit integer**. Therefore, the possible range for tokens is from **-2^63 to 2^63-1**

- **Token Range**: Each node in the cluster is responsible for a segment of the token ring, this segment is referred to as the **"token range"**. The token range defines the set of data that the node is responsible for. When a node owns a token range, it is responsible for storing all data that has a token value within that range.

- **Partitioners:** The partitioner determines how data is distributed across the nodes in a Cassandra cluster. Basically, a partitioner is a hash function to determine the token value by hashing the partition key of a row's data. Then, this partition key token is used to determine and distribute the row data within the ring. The **Murmur3Partitioner** is the default partitioner since Cassandra version 1.2. It uses the **MurmurHash** function that creates a 64-bit hash of the partition key.

Data Partitioning & Token

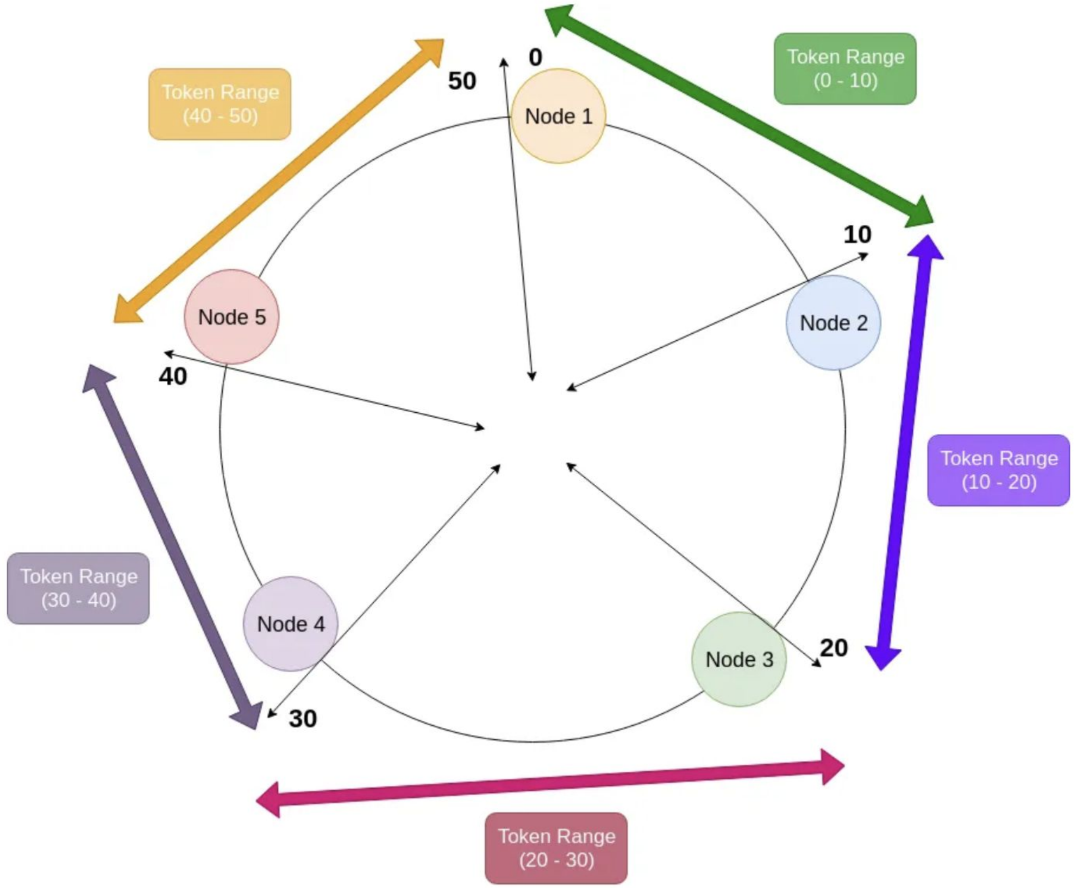


Car Name	Engine Capacity	Color	NoOfDoors
BMW	1000cc	Red	2
BMW	1500cc	Red	2
BMW	1500cc	Blue	2
BMW	1500cc	Red	4
BMW	1500cc	Blue	4
Toyata	1000cc	White	4
Toyata	1000cc	Black	4
Toyata	1500cc	Red	4
Audi	2000cc	Black	2
Audi	1500cc	Yellow	2
Benz	2000cc	White	2
Benz	2000cc	Red	2

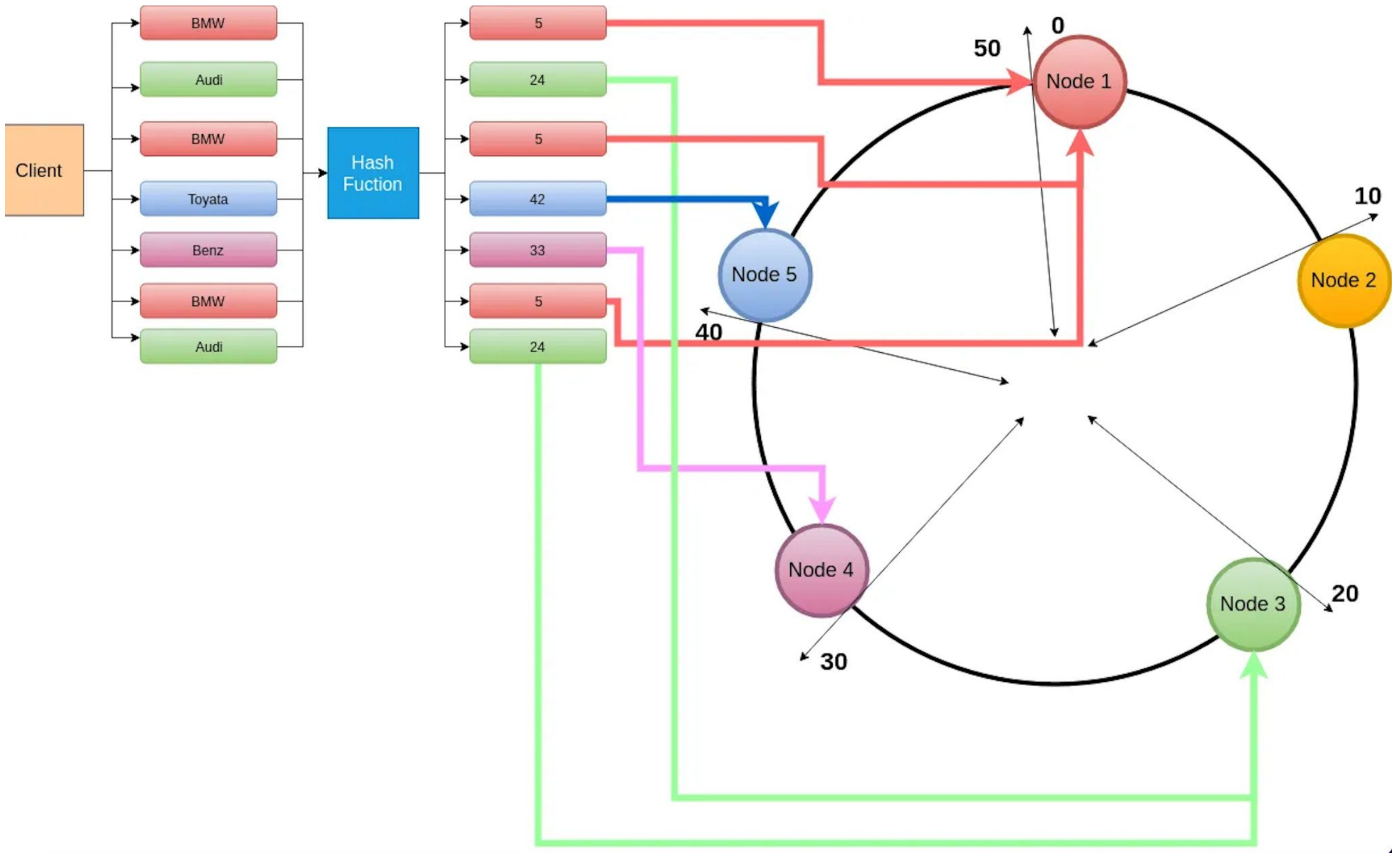
In above table, **Car Name** is a partitioning key. When data is coming to the cassandra, it gets token of that incoming data by providing partitioning key to the hash function.

BMW -> Hash Function -> 9
Toyata -> Hash Function -> 17
Audi -> Hash Function -> 31
Benz -> Hash Function -> 25

Data Partitioning & Token



Data Partitioning & Token



VNodes in Cassandra

In a traditional configuration, a single node is responsible for a range of tokens, splitting the token space between nodes. For example, with a three-node cluster, the first node could be responsible for token values 1 to 100, the second for 101 to 200, and the third for 201 to 300. This is called "vnode=1" configuration, which means each node has only one token range.

However, this can lead to problems when adding or removing nodes, because moving a large range of tokens can be a heavy operation. It also doesn't provide very good data distribution if some ranges of primary keys are more heavily used than others.

That's where Virtual Nodes, or "vnodes", come in. Instead of each node having a single token range, with vnodes each node can have multiple token ranges. For example, the first node might be responsible for token values 1 to 50 and 151 to 200, the second for 51 to 100 and 201 to 250, and the third for 101 to 150 and 251 to 300.

This gives a few advantages:

- **Better distribution of data and load**: Because each node handles multiple ranges, the data and load can be spread more evenly.

- **Easier to add and remove nodes**: When a new node is added, it takes over some of the token ranges from existing nodes, and it's easier to move several small ranges than one large range.

VNodes in Cassandra

Now let's consider an example of how vnodes help in distributing data. Assume we have 6 vnodes and 3 physical nodes in our cluster. Each physical node will hold 2 vnodes.

Let's say that the entire token range of our system is from 0 to 300 (this is a simplification, as in reality the token range is based on the hash function used, which typically produces a much larger range, but the concept is the same). We can divide this entire range over our 6 vnodes:

- vnode 1: 0 to 50
- vnode 2: 51 to 100
- vnode 3: 101 to 150
- vnode 4: 151 to 200
- vnode 5: 201 to 250
- vnode 6: 251 to 300

Next, we distribute these vnodes among the physical nodes:

- Physical Node 1: vnode 1 (0-50) and vnode 4 (151-200)
- Physical Node 2: vnode 2 (51-100) and vnode 5 (201-250)
- Physical Node 3: vnode 3 (101-150) and vnode 6 (251-300)

So each physical node is responsible for multiple ranges in the overall token range, making data distribution more balanced and allowing the system to handle adding and removing nodes more easily.

To calculate the token range of each vnode, you simply divide the total token range by the number of vnodes. Note that, as mentioned above, in a real system the total token range would be determined by the hash function used (in Cassandra, it's Murmur3, which gives a range from -2^63 to +2^63-1), but the principle is the same.

Write operation in Cassandra

- **Client Request**: The client sends a write request to the coordinator node in the Cassandra cluster.

- **Commit Log Write**: The first thing the coordinator node does is write the operation to the commit log on disk. The commit log is a crash-recovery mechanism in Cassandra. It ensures that all writes are durable and can be recovered in case a node crashes or experiences a power loss.

- **Memtable Write**: After the write has been recorded in the commit log, the data is then written to an in-memory data structure called the memtable. The memtable stores data in sorted order until its size limit is reached.

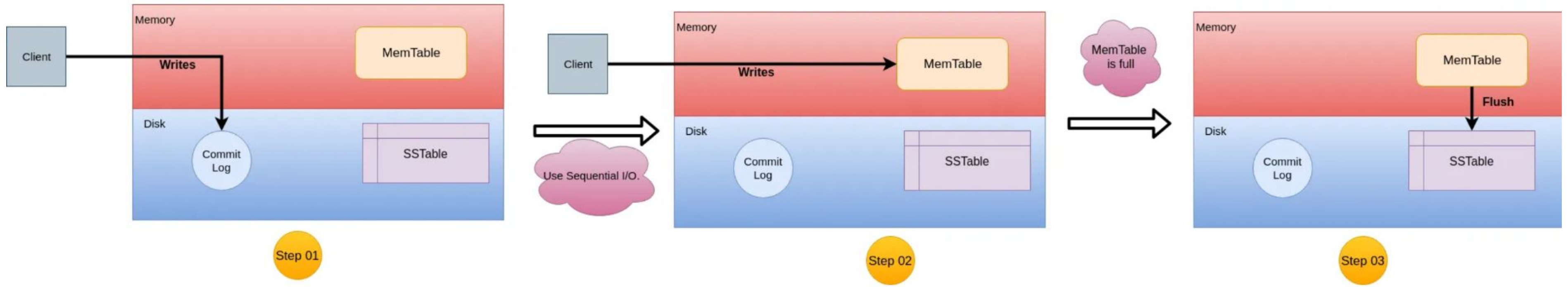
- **SSTable Flush:** When the memtable is full, it is flushed to disk as an SSTable (Sorted String Table). SSTables are immutable data files which store rows of data sorted by the key.

- **Replica Write**: Depending on the configured replication factor and strategy, the write is also sent to replica nodes where steps 2, 3, and 4 are repeated. This process ensures the redundancy and high availability of data in the cluster.

- **Acknowledgement**: Once the data is written into the memtable and commit log, the node sends an acknowledgement to the coordinator node.

- **Response to Client:** After the coordinator has received acknowledgements from a sufficient number of replicas (according to the consistency level specified in the write request), it sends a success response to the client.

Write operation in Cassandra



Why Cassandra write operation is fast?

SSTables in Cassandra are indeed immutable. When there's a need to update a column, Cassandra can't modify the SSTables directly. Instead, it adds a new record. However, this process could be slow if it were to write directly to the SSTable.

For better performance, Cassandra does not write updates directly to the SSTable. It first writes to the commit log and the memtable, and then sends an acknowledgement to the client. The write operation is held in memory, and once it reaches a configurable limit, Cassandra flushes those changes to the SSTable using sequential I/O. This approach minimizes disk I/O operations at the time of the write, contributing to Cassandra's high-speed write operations.

What happen if Cassandra machine crashes before flushing data into the SSTable?

All data in Cassandra is initially written to the memtable. Suppose there's a machine crash. When the machine is rebooted and operational again, all the data previously stored in the memtable is lost.

To prevent such data loss, Cassandra uses commit logs. Every time there's a change in the memtable, Cassandra records these changes in the commit log. This allows it to keep track of memtable changes and restore any lost data when necessary.

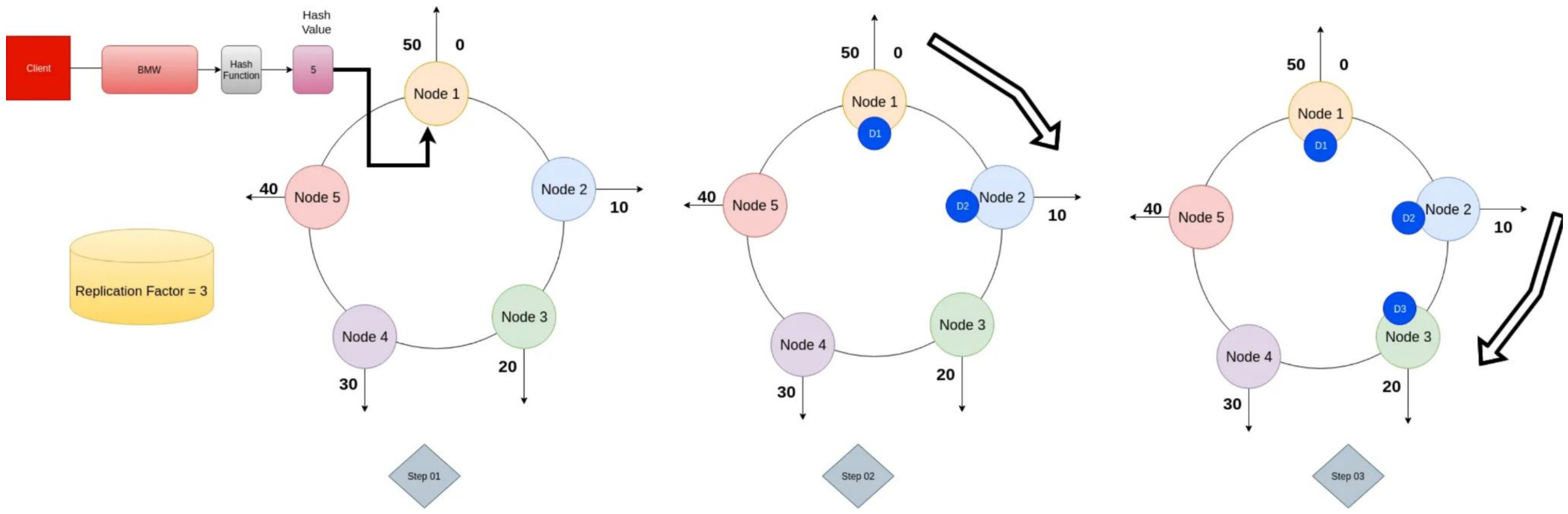
Why Cassandra writes data to commit log instead of SSTable?

The commit log in Cassandra stores changes in a single file, which greatly reduces disk I/O operations. Furthermore, even when receiving updates for multiple column families simultaneously, the disk doesn't need to perform numerous seek operations, optimizing efficiency.

Replication in Cassandra

In Apache Cassandra, the replication strategy determines the nodes where replicas are placed. The replication strategy you choose affects the distribution of data and the level of data redundancy. There are two types of replication strategies in Cassandra:

- **SimpleStrategy:** This strategy is used when you have just one data center. It places the first replica on the node selected by the partitioner. The remaining replicas are placed on the next nodes in the ring clockwise, without considering the topology (rack or data center location).



Replication in Cassandra

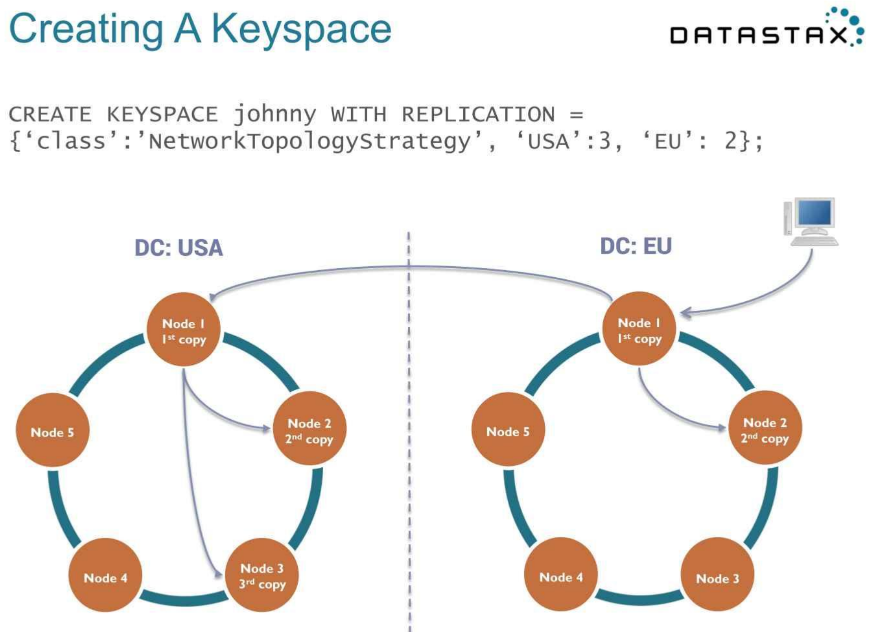
- **NetworkTopologyStrategy:** This strategy is used when you have more than one data center. It places replicas in the same data center by walking the ring clockwise until it has placed the requested number of replicas. The strategy decides how many replicas to place in each data center, and on which racks. This strategy is rack-aware and attempts to place replicas on different racks for fault tolerance.

    - **Data Center Selection:** For each write operation, Cassandra chooses a coordinator node (based on request routing or client configuration). The coordinator is responsible for handling the client's read/write requests.

    - **Replica Placement in First Data Center:** The coordinator first identifies which node in its data center owns the token range for the data to be written. This node is the primary replica. Additional replicas are placed on nodes in different racks (if available) within the same data center, up to the replication factor specified for that data center.

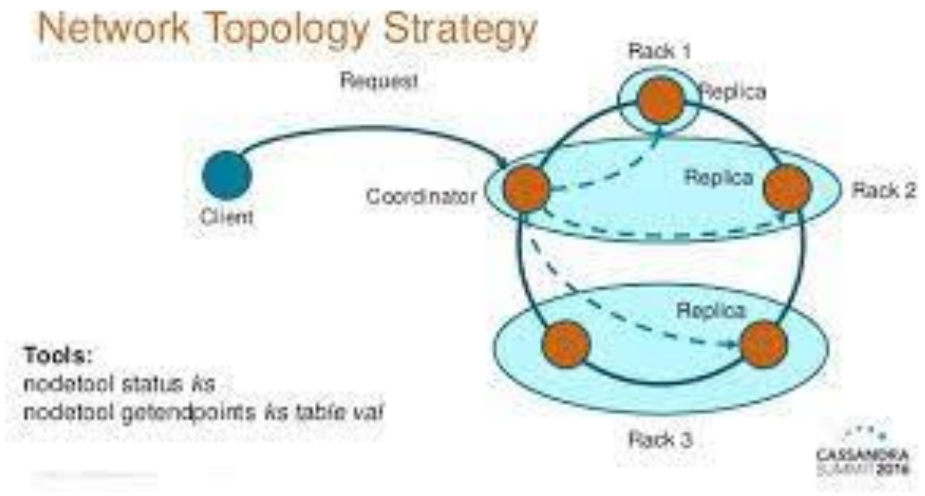
    - **Replica Placement in Additional Data Centers:** For the next data center, the process repeats independently. The node owning the token range for the data in the second data center becomes the primary replica for that data center. Subsequent replicas are placed on different racks within that data center, respecting the replication factor defined for that specific data center.

Replication in Cassandra



Creating A Keyspace

```
CREATE KEYSPACE johnny WITH REPLICATION =
{'class':'NetworkTopologyStrategy', 'USA':3, 'EU': 2};
```



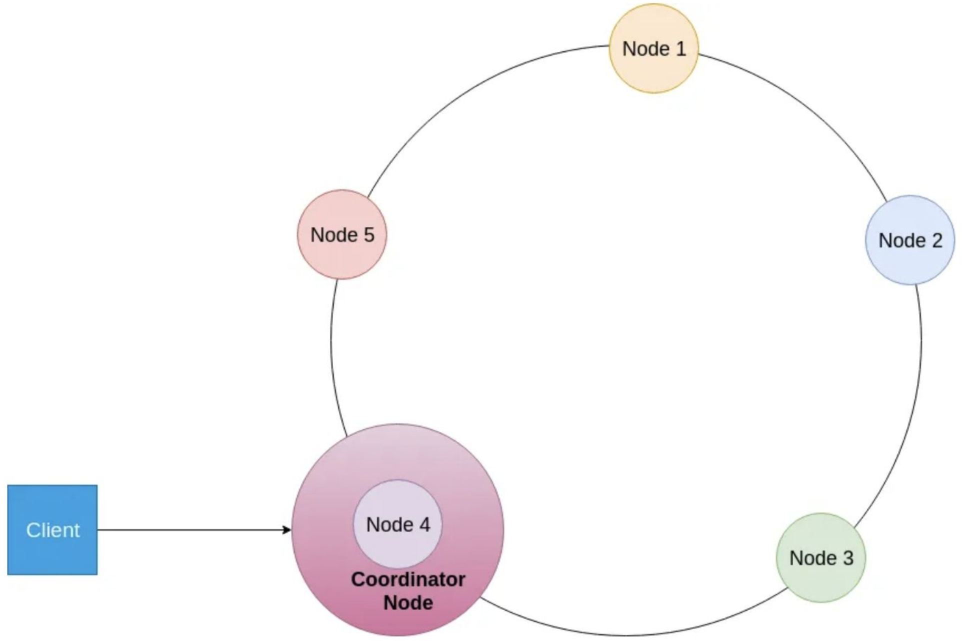
Network Topology Strategy

Tools:
nodetool status ks
nodetool getendpoints ks table val

What is Coordinator in Cassandra ?

When Cassandra client hits a **write** or **read** request on the node in the cluster. That node is called as **coordinator**. Coordinator node may be changed every time.

Coordinator is selected by the cassandra driver based on the policy, you have set. Most common policies are **DCAwareRoundRobinPolicy** and **TokenAwarePolicy**.



What is Coordinator in Cassandra ?

- **DCAwareRoundRobinPolicy**: This policy aims to prioritize nodes in the same datacenter to reduce cross-datacenter latency. It first routes requests to the local datacenter nodes and only if these nodes are unavailable or overloaded, it directs requests to nodes in other datacenters.

- **TokenAwarePolicy**: This policy directs read requests to the node that holds the replica of the data being requested, reducing the need for internode communication. It's aware of the partition key used in the query and can thus route the request directly to the nodes that own the data, resulting in increased efficiency.

Read operation in Cassandra

When read operation comes to Cassandra, that operation hit on one node. that node is coordinator node.

The row key must be supplied for every read operation. The row key is another name for the PRIMARY KEY.(partition key + clustering keys)

These are the steps when reading data from Cassandra,

1. Check the memtable
2. Check row cache, if enabled

   Row cache pulls entire partitions into memory. if any part of partition has been modified , the entire cache for that row is invalidated. So invalidating big pieces of memory.

3. Checks Bloom filter
   A Bloom filter in Apache Cassandra is a space-efficient, probabilistic data structure used to quickly check if an element (specifically, a row key) is likely present in an SSTable. It doesn't hold actual data but instead uses hash functions to map data to a bit array

4. Checks partition key cache, if enabled
   The key cache holds the location of keys in memory on a per-column family basis. Key cache helps where a particular partition begins in the SSTable.

Read operation in Cassandra

5. Goes directly to the compression offset map if a partition key is found in the partition key cache, or checks the partition summary if not If the partition summary is checked, then the partition index is accessed

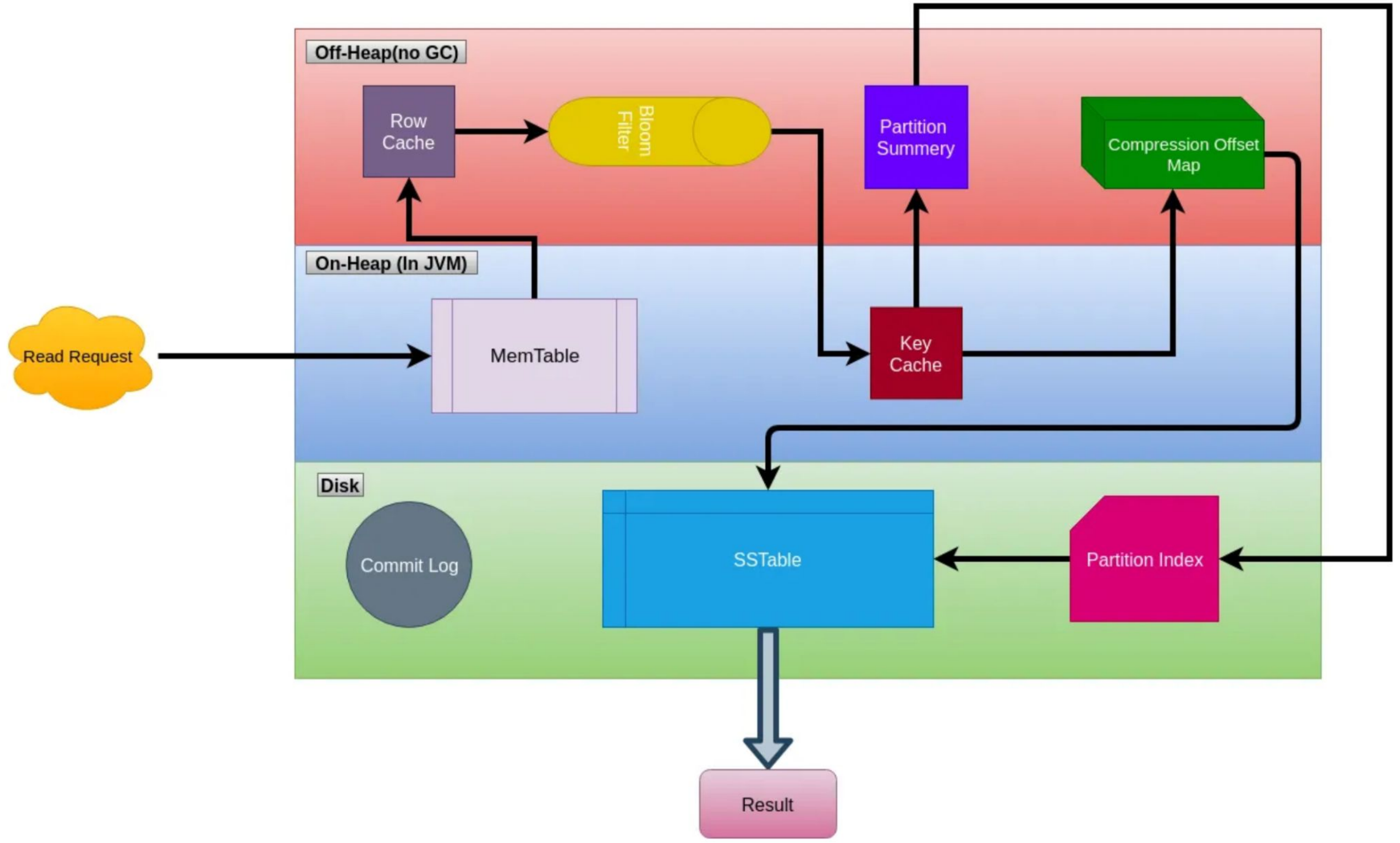
> Partition summery is sampling of partition index to speedup the access to index on disk.Default sampling ratio is 128, meaning that for every 128 records for a index in index file, we have 1 records in partition summary. Each of these records of partition summary will hold key value and offset position in index.

6. Locates the data on disk using the compression offset map

> Compression offset maps holds the offset information for compressed blocks.

7. Fetches the data from the SSTable on disk

Read operation in Cassandra



Compaction in Cassandra

Compaction in Apache Cassandra is the process of merging multiple SSTables (sorted string tables) into one for the purpose of improving performance and reclaiming storage space.

1. **Compaction Strategies:** Cassandra offers several compaction strategies that control how and when compaction occurs, each suited to different use cases. These include SizeTieredCompactionStrategy, LeveledCompactionStrategy, and TimeWindowCompactionStrategy.

    a. **SizeTieredCompactionStrategy (STCS):** This strategy triggers compaction when a certain number of similarly-sized SSTables exist. It's most suitable for write-intensive workloads.

    b. **LeveledCompactionStrategy (LCS):** LCS compacts SSTables into different "levels" based on their size and age, reducing read amplification at the cost of write amplification. It's most suited for read-intensive workloads.

    c. **TimeWindowCompactionStrategy (TWCS):** TWCS groups SSTables into time windows and compacts them separately. It's designed for time-series data, where older data is rarely updated.

Gossip Protocol in Cassandra

Gossip is a peer-to-peer communication protocol used by Apache Cassandra for inter-node communication. It plays a crucial role in maintaining the health status of all nodes in a cluster. Here's a quick breakdown:

1. **Information Sharing**: Every node in a Cassandra cluster communicates with each other using Gossip to share information about themselves and about other nodes that they have gossiped about.

2. **Heartbeat**: Each node in a Cassandra cluster sends a Gossip message, or "heartbeat," to a random node every second. The heartbeat contains information about the state of the sender as well as other nodes in the cluster.

3. **Scalable and Reliable**: As a peer-to-peer protocol, Gossip is highly scalable and can function even if some nodes in the cluster are unreachable.

4. **Node State**: Gossip messages include information about a node's state (UP, DOWN), schema version, boot time, and other operational details.

5. **Failure Detection**: Gossip also helps in failure detection. If a node fails to receive a gossip message from another node within a certain period, it will consider that node as unresponsive and mark it down.

6. **Data Distribution**: Gossip is integral to data distribution in Cassandra as it helps in deciding where to read/write data, providing details about the location of replicas.

Write Consistency in Cassandra

Consistency in Apache Cassandra refers to the guarantee of how up-to-date and synchronized a row of Cassandra data is on all of its replicas.

1. Any - a write must succeed on any available node.(Highest availability)
2. One - a write must succeed on any node responsible for that row.(either primary or replica)
3. Two - a write must succeed on two nodes.
4. Quorum - a write must succeed on a quorum of replica nodes.
    a. **quorum nodes = (replication factor/2) + 1**
5. Local Quorum - a write must succeed on a quorum of replica nodes in the same data center as the coordinator nodes.
6. All - a write must succeed on all replica nodes.(Lowest availability)

```
INSERT INTO employees (id, first_name, last_name, email)
VALUES (1, 'John', 'Doe', 'john.doe@example.com')
USING CONSISTENCY QUORUM;

UPDATE employees
USING CONSISTENCY QUORUM
SET email = 'johndoe@example.com'
WHERE id = 1;
```

Read Consistency in Cassandra

- ANY: This level is not applicable for read operations. Used only for writes.
- ONE: Only one replica needs to respond. Fastest read but might return stale data.
- TWO: Two replicas must respond.
- QUORUM: A majority of replicas (in the entire cluster) must respond. Helps balance between performance and data accuracy.
- LOCAL_QUORUM: A majority of replicas (in the local datacenter) must respond. Provides lower latency than QUORUM in multi-datacenter deployments.
- ALL: All replicas must respond. Provides the highest level of consistency but at the cost of availability if any replica is down.

```
SELECT * FROM employees WHERE id = 1 USING CONSISTENCY QUORUM;
```

Partition Key, Cluster Key and Row Key Declaration

```
CREATE TABLE employees (
        department_id int,
        office_id int,
        employee_id int,
        first_name text,
        last_name text,
        email text,
        PRIMARY KEY ((department_id, office_id), employee_id, last_name)
);
```