BIG DATA ANALYTICS (CS-431)

Dr. Sriparna Saha
Associate Professor

Website: https://www.iitp.ac.in/~sriparna/
Google Scholar: https://scholar.google.co.in/citations?user=Fj7jA_AAAAAJ&hl=en
Research Lab: SS_Lab
Core Research AREA: NLP, GenAI, LLMs, VLMs, Multimodality, Meta-Learning, Health Care, FinTech, Conversational Agents

TAs: Sarmistha Das, Nitish Kumar, Divyanshu Singh, Aditya Bhagat, Harsh Raj

Challenges in Big Data Systems

- Big Data systems like Hadoop and Spark run on large clusters of machines.
- Coordinating tasks and managing shared state across hundreds of nodes is complex.
- Node failures are common and must be handled automatically to prevent system downtime.
- This creates a need for a robust, distributed coordination service.

Apache ZooKeeper's Role in Big Data

It basically keeps track of information that must be synchronized across your cluster

- Which node is the master?
- What tasks are assigned to which workers?
- Which workers are currently available?

It's a tool that applications can use to recover from partial failures in your cluster.

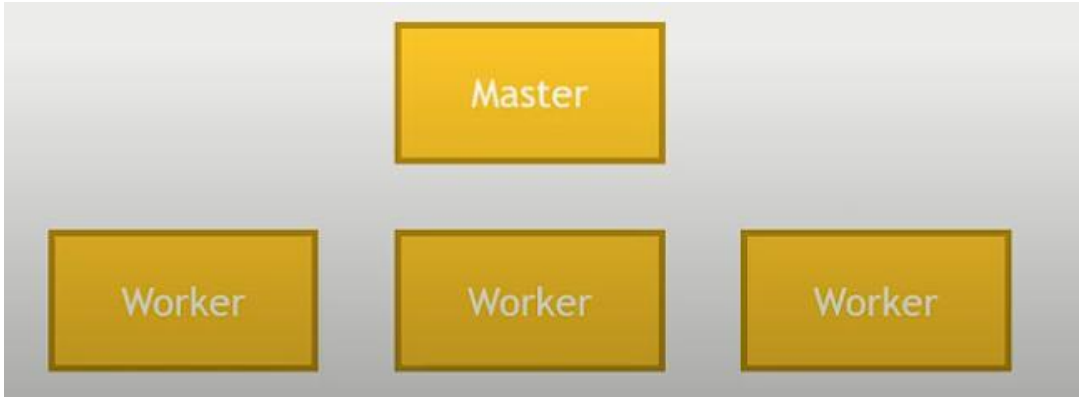
An integral part of HBase, High-Availability (HA) MapReduce, Drill, Storm, Solr, and much more

Apache ZooKeeper's Failure Mode

Master crashes, needs to fail over to a backup

Worker crashes - its work needs to be redistributed

Network trouble - part of your cluster can't see the rest of it



Primitive operations in a distributed system

Master election

- One node registers itself as a master, and holds a “lock” on that data
- Other nodes cannot become master until that lock is released
- Only one node allowed to hold the lock at a time

Crash detection

- “Ephemeral” data on a node’s availability automatically goes away if the node disconnects, or fails to refresh itself after some time-out period.

Group management

Metadata

- List of outstanding tasks, task assignments

Primitive operations in a distributed system

Master election

- One node registers itself as a master, and holds a “lock” on that data
- Other nodes cannot become master until that lock is released
- Only one node allowed to hold the lock at a time

Crash detection

- “Ephemeral” data on a node’s availability automatically goes away if the node disconnects, or fails to refresh itself after some time-out period.

Group management

Metadata

- List of outstanding tasks, task assignments

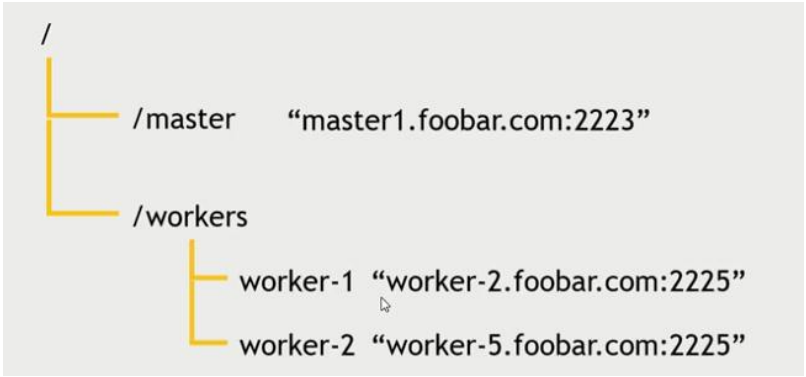
But ZooKeeper's API is not about these primitives.

- Instead they have built a more general purpose system that makes it easy for applications to implement them.

Zookeeper's API

Really a little distributed file system

- With strong consistency guarantees
- Replace the concept of “file” with “znode” and you’ve pretty much got it
- ZooKeeper provides a **general API (znodes, watches, ephemeral nodes)** that lets applications *implement* things like master election, crash detection, and group management on top of it.



Here's the ZooKeeper API:

- Create, delete, exists, setData, getData, getChildren

Persistent and ephemeral znodes

Persistent znodes remain stored until explicitly deleted

- i.e., assignment of tasks to workers must persist even if the master crashes

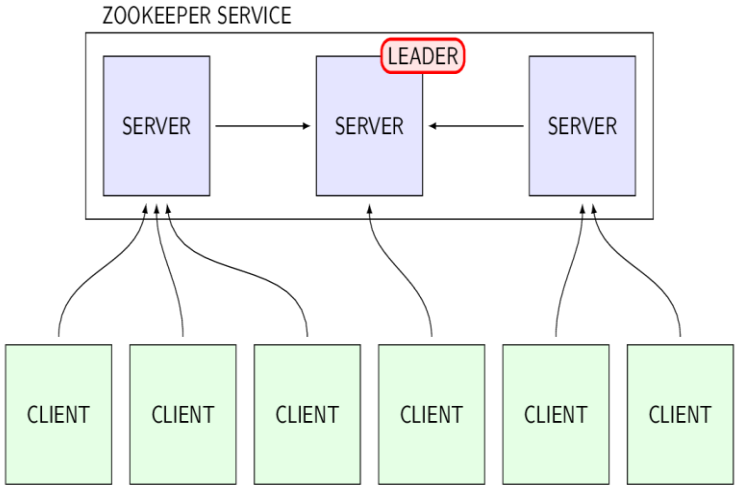
Ephemeral znodes go away if the client that created it crashes or loses its connection to ZooKeeper

- i.e., if the master crashes, it should release its lock on the znode that indicates which node is the master!

ZooKeeper Architecture Service

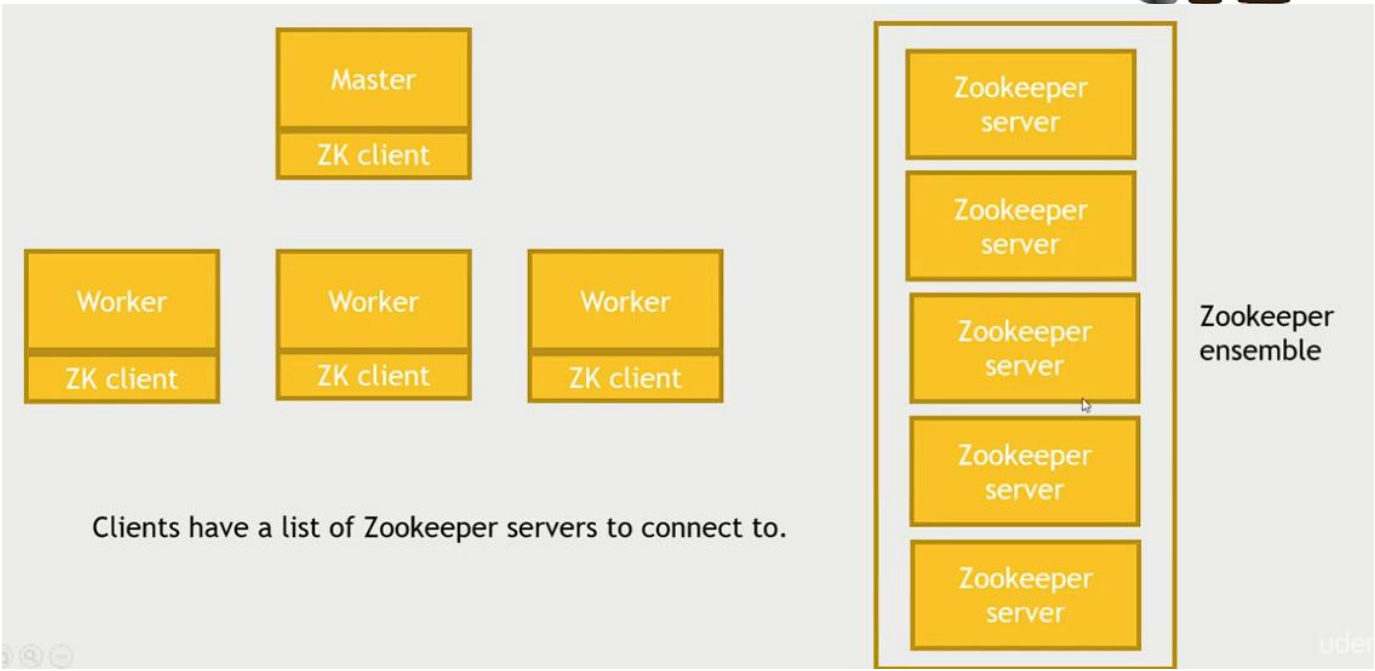
ZooKeeper uses a replicated client-server model built for high availability. The core components are:

- **Ensemble**: A cluster of servers (usually an odd number, like 3 or 5).
- **Master**: One server is elected to handle all **write** requests, ensuring data is updated in the correct order.
- **Followers**: All other servers replicate the leader's data and handle all **read** requests.
- **Client**: Your application connects to any server. Writes are forwarded to the leader, while reads are served locally for speed.



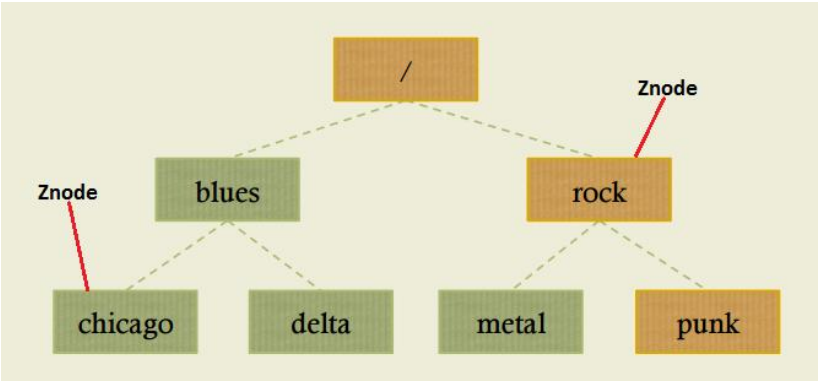
ZooKeeper Architecture Service

APACHE ZooKeeper™



ZooKeeper's Data Model (Znodes)

- ZooKeeper organizes data in a hierarchical namespace, similar to a file system.
- Each node in this hierarchy is called a **Znode**, which can store small amounts of data.
- **Ephemeral Znodes** exist only as long as the session that created them is active.
- Big Data applications use this to monitor node health; if a node's ephemeral znode disappears, it's considered offline.



ZooKeeper's Reliability Guarantees

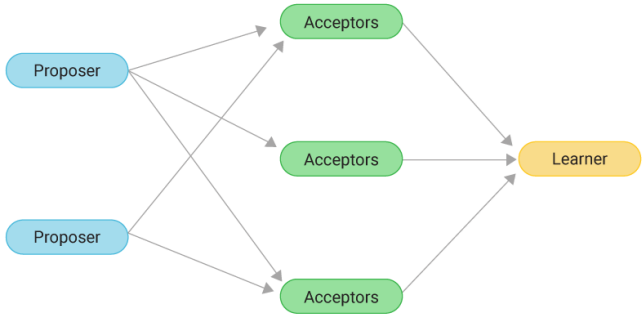
- ZooKeeper provides strong guarantees, like **sequential consistency** and **atomicity**.
- This means all updates are applied in order and either fully succeed or fully fail.
- For a big data pipeline, this ensures that critical configuration changes are not partially applied.
- These guarantees are fundamental to building fault-tolerant Big Data applications.

The Consensus Problem in Big Data

- Consensus is the challenge of getting multiple servers in a cluster to agree on a single value.
- For example, in a high-availability setup, all nodes must agree on which node is the active HMaster.
- This is difficult because servers can fail or network messages can be delayed.
- Without consensus, a cluster could suffer from a "split-brain" scenario, leading to data corruption.

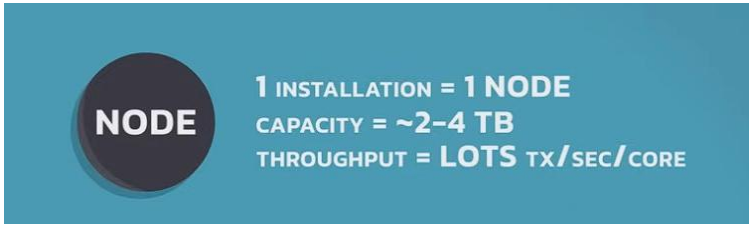
The Paxos Algorithm for Consensus

- Paxos is a proven algorithm that allows a distributed system to reach consensus.
- It uses a two-phase protocol (prepare/propose) to ensure only one value is chosen.
- While complex, it provides the mathematical foundation for the fault-tolerance in ZooKeeper.
- This fault-tolerance is what makes Big Data systems resilient to partial hardware failures.



Introduction to Cassandra

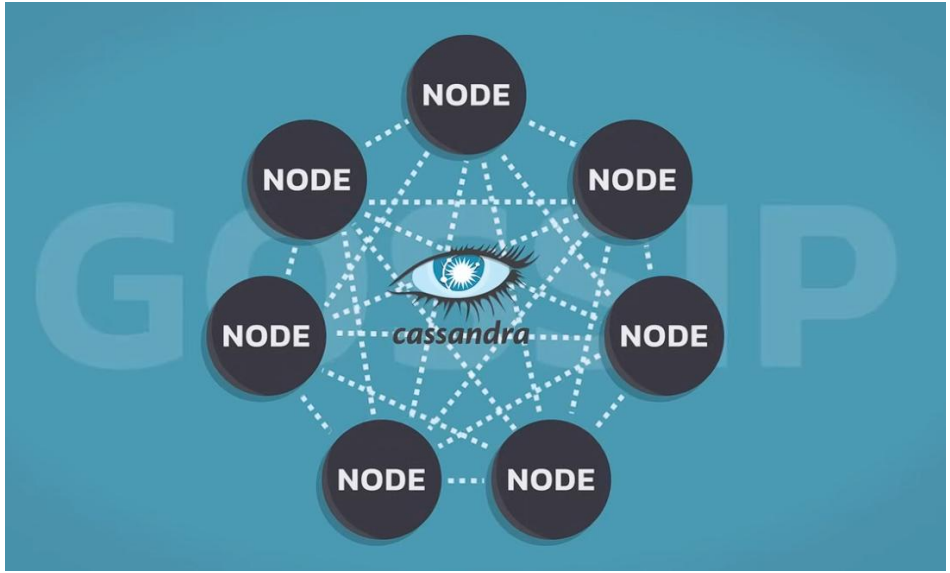
- Cassandra is a decentralized NoSQL database built for extreme scale and high availability.
- It excels at handling high-velocity data ingestion, common in IoT and web analytics.
- Its masterless "ring" architecture means there is no single point of failure.
- This makes it a popular choice for mission-critical Big Data applications that cannot afford downtime.
- Petabyte Database, high availability, performance and vendorr independant



Introduction to Cassandra

Cassandra: Nodes & Gossip

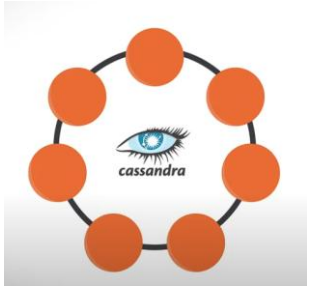
- **Nodes:**
  - All peers, no master.
  - Connected in a ring.
  - Data is partitioned & replicated.
- **Gossip:**
  - Nodes share info randomly.
  - Spreads cluster state (alive, dead, metadata).
  - Works like word-of-mouth.



Partitions & Partition Key (Cassandra)

Partition Key:

- Determines **which node stores the data**.
- Example: If we store **companies**, the partition key could be **country name**.
  - `"USA"` → goes to Node A
  - `"France"` → goes to Node B
  - `"India"` → goes to Node C



Partition:

- A **group of rows** that share the same partition key
- Example: All companies in **USA** are in the **USA partition**.
- Helps Cassandra **quickly find and store data**.

Why it matters:

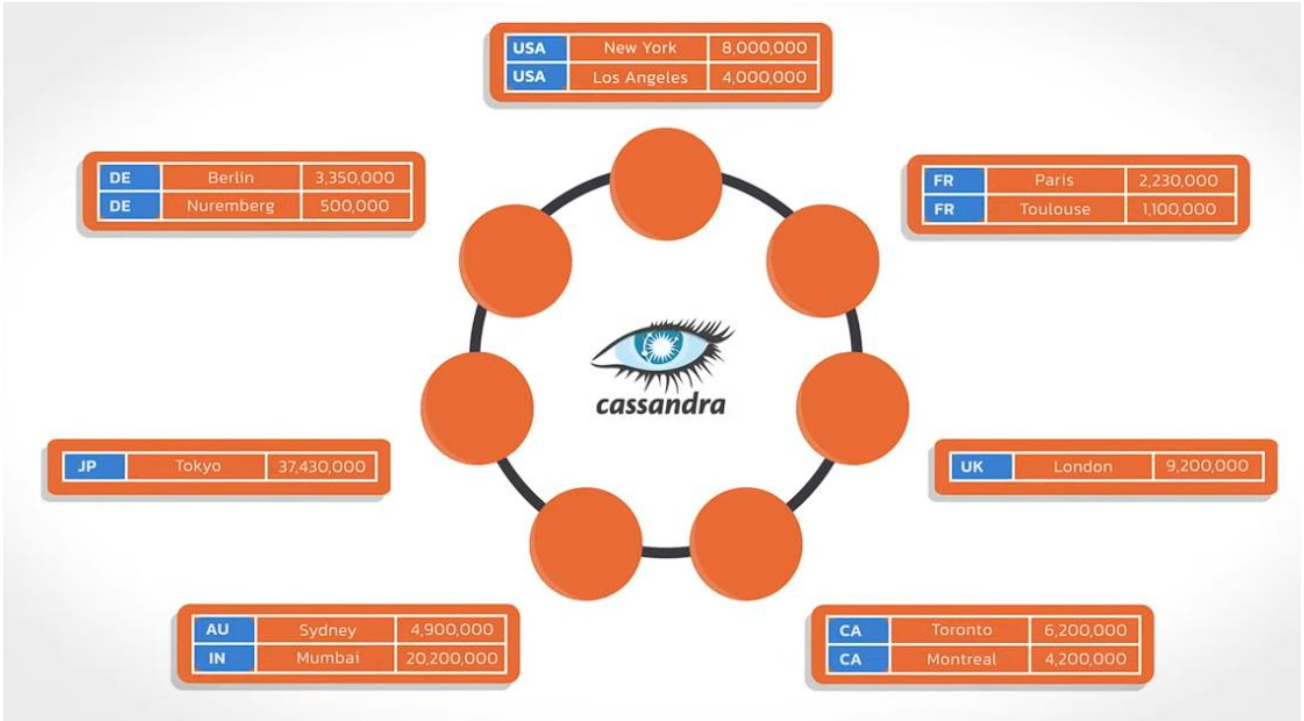
- Ensures data is **evenly distributed** across nodes.
- Makes reads/writes **fast and organized**.

PARTITIONS

Partition Key

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

Partitions & Partition Key (Cassandra)



Cassandra: Replication Factor & Coordinator

Replication Factor (RF):

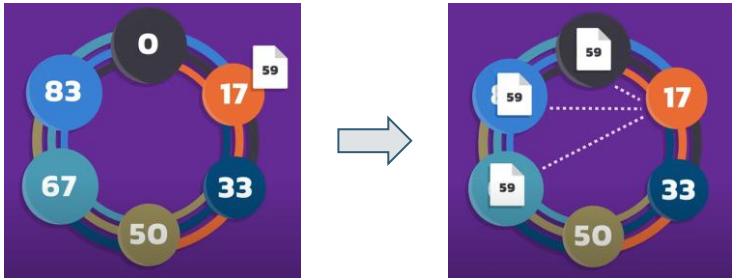
- Number of copies of data stored across different nodes.
- Example: RF = 3 → each row stored on **3 different nodes**.
- Ensures **fault tolerance** (data safe if nodes fail).

Coordinator Node:

- **Any node** that receives a client request acts as coordinator.
- Decides which replica nodes to contact for reads/writes.
- Ensures consistency across replicas.

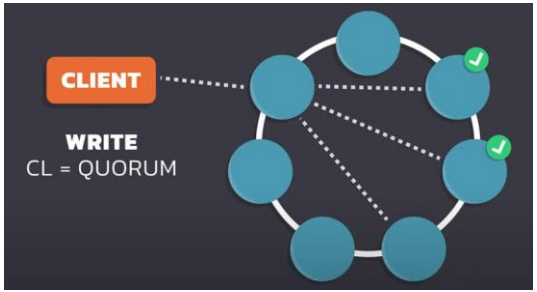
Why it matters:

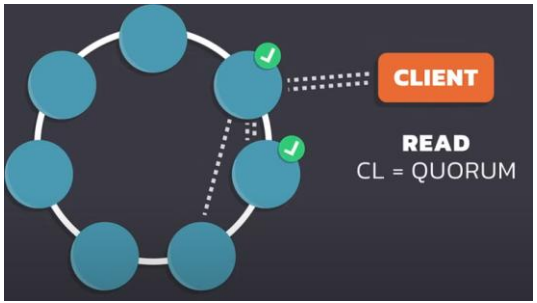
- RF + coordinator → **reliable, consistent, fault-tolerant system**.



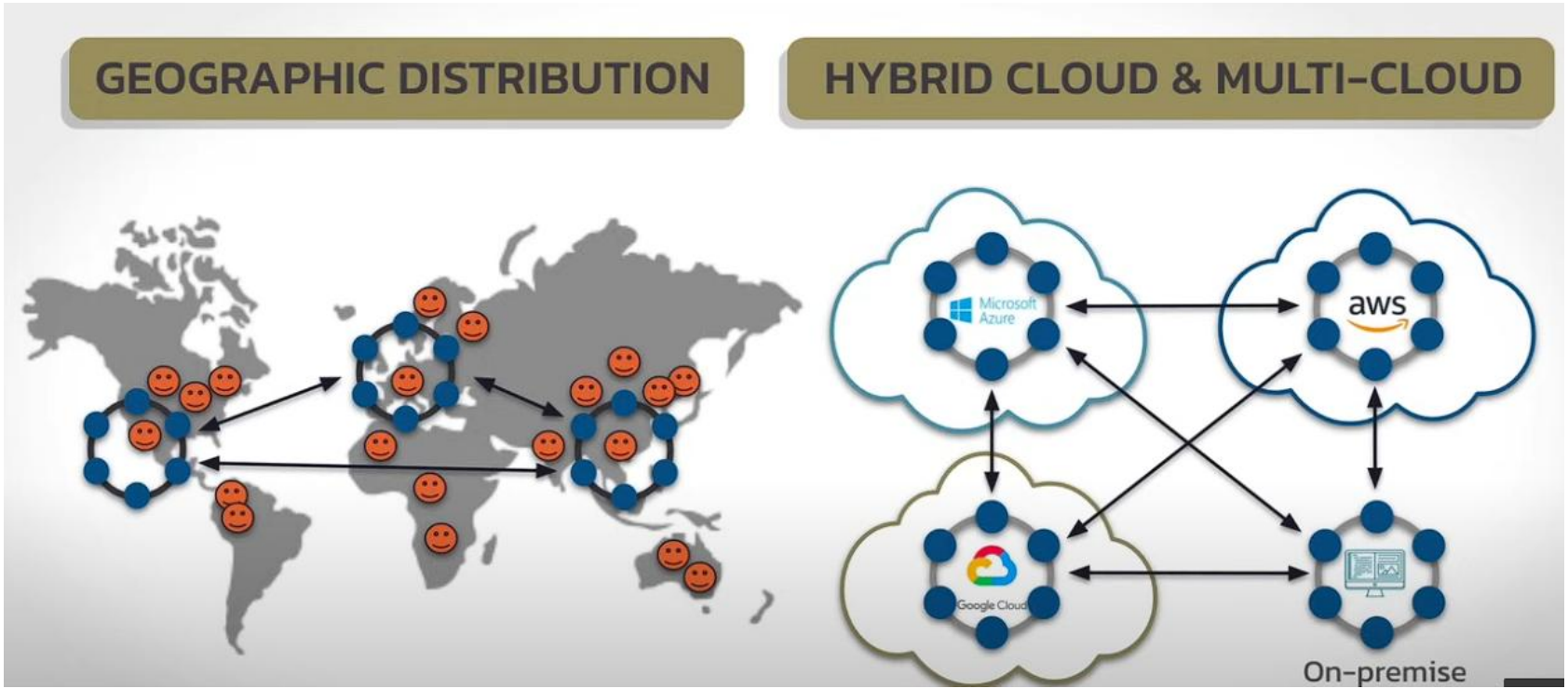
Cassandra: Quorum Reads & Writes

- **Write Quorum:** Majority of replicas must confirm (RF = 3 → 2 nodes).
- **Read Quorum:** Majority of replicas respond to return latest data.
- Ensures **strong consistency** across the cluster.



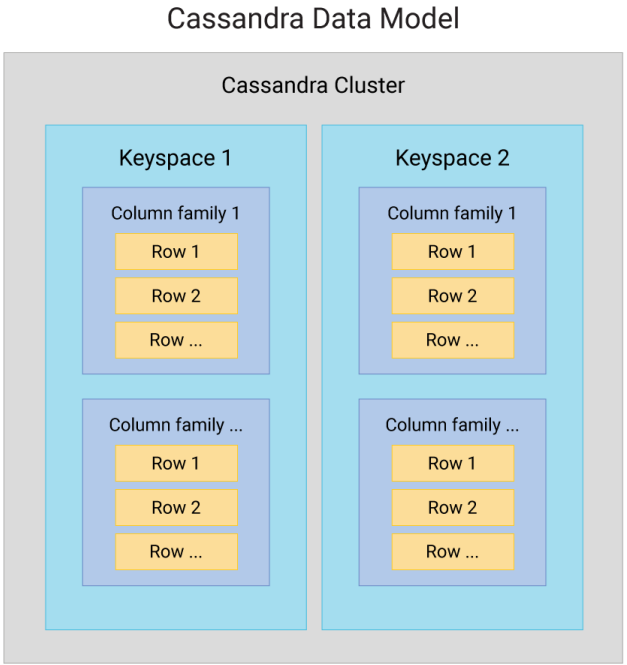


Cassandra: Quorum Reads & Writes



Cassandra's Wide-Column Data Model

- Cassandra organizes data into Keyspaces, Tables, Rows, and Columns.
- Unlike a relational database, different rows in the same table can have different sets of columns.
- This schema flexibility is perfect for Big Data analytics where data structures may evolve.
- It allows you to add new metrics or attributes without costly database migrations.



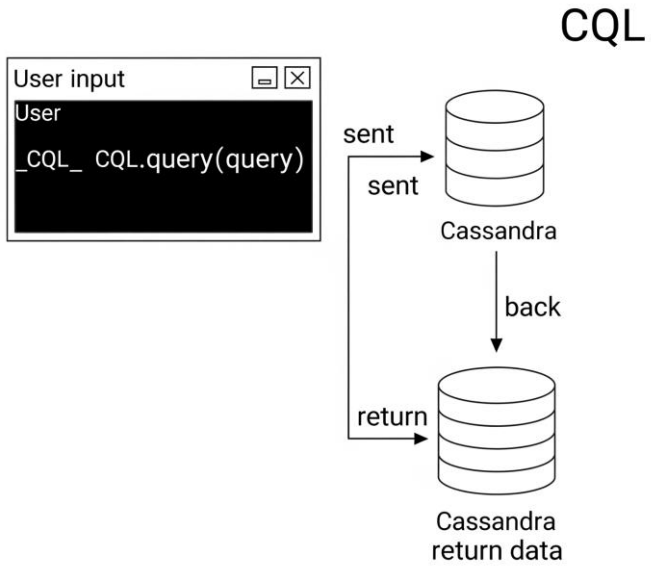
The Importance of Denormalization

- Cassandra does not support joins, which are expensive operations in distributed systems.
- Instead, you practice **denormalization**: creating tables specifically designed to answer a query.
- This means you might store the same piece of data in multiple tables in different forms.
- This "query-first" design approach is key to achieving high performance at Big Data scale.

Denormalization
Used to increase redundancy to execute queries
Data integrity is not supported
Reduces the number of tables
Doesn't optimize disk space usage

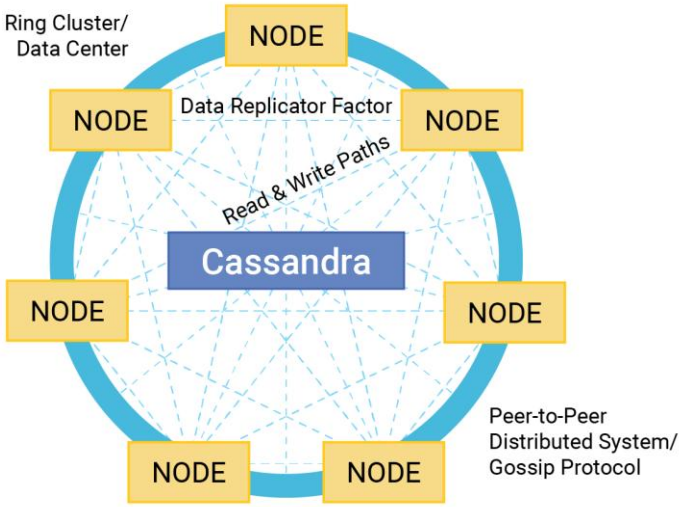
Interacting with Cassandra Query Language (CQL)

- CQL is the primary way to interact with Cassandra, and its syntax is very similar to SQL.
- This makes the transition easier for developers and analysts familiar with relational databases.
- You can use familiar commands like `SELECT`, `INSERT`, `UPDATE`, and `DELETE`.
- Example: `SELECT event_name, event_time FROM user_activity WHERE user_id = 'some_user';`.



Cassandra Internals : The Peer-to-Peer "Ring" Architecture

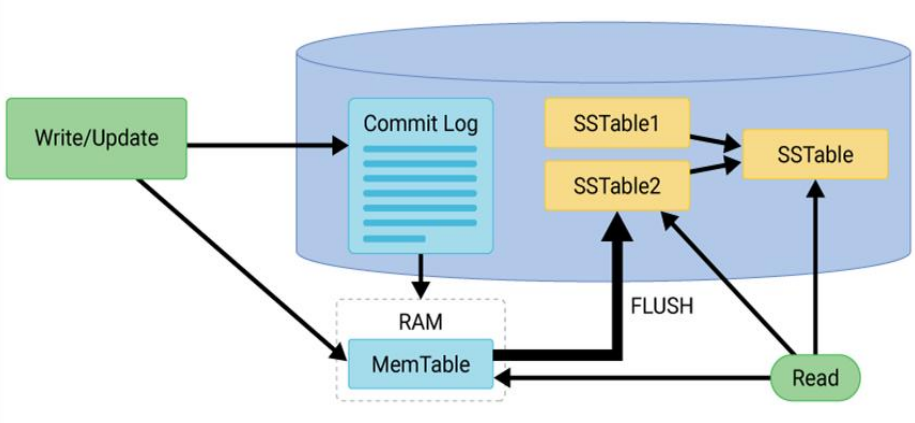
- In Cassandra, all nodes are equal; there is no master node that manages the cluster.
- Nodes communicate using a **gossip protocol** to share information about their health and status.
- This decentralized design eliminates single points of failure.
- If a node goes down, the cluster continues to operate, ensuring high availability for Big Data services.



Database

The Write Path: Memtable & Commit Log

- When a write occurs, Cassandra first appends it to a **Commit Log** on disk for durability.
- Simultaneously, it writes the data to an in-memory structure called a **Memtable**.
- This process is extremely fast because it avoids slow, random disk I/O.
- This write-optimized design is why Cassandra can handle massive ingestion rates from Big Data sources.



The Write Path: SSTables (Sorted String Table)

- When the Memtable fills up, its sorted data is flushed to a new, immutable file on disk called an **SSTable**.
- This is a core component of the **Log-Structured Merge-Tree (LSM-Tree)** data structure.
- LSM-trees are optimized for systems with very high write volumes.
- This makes them a perfect fit for the demands of modern Big Data applications.