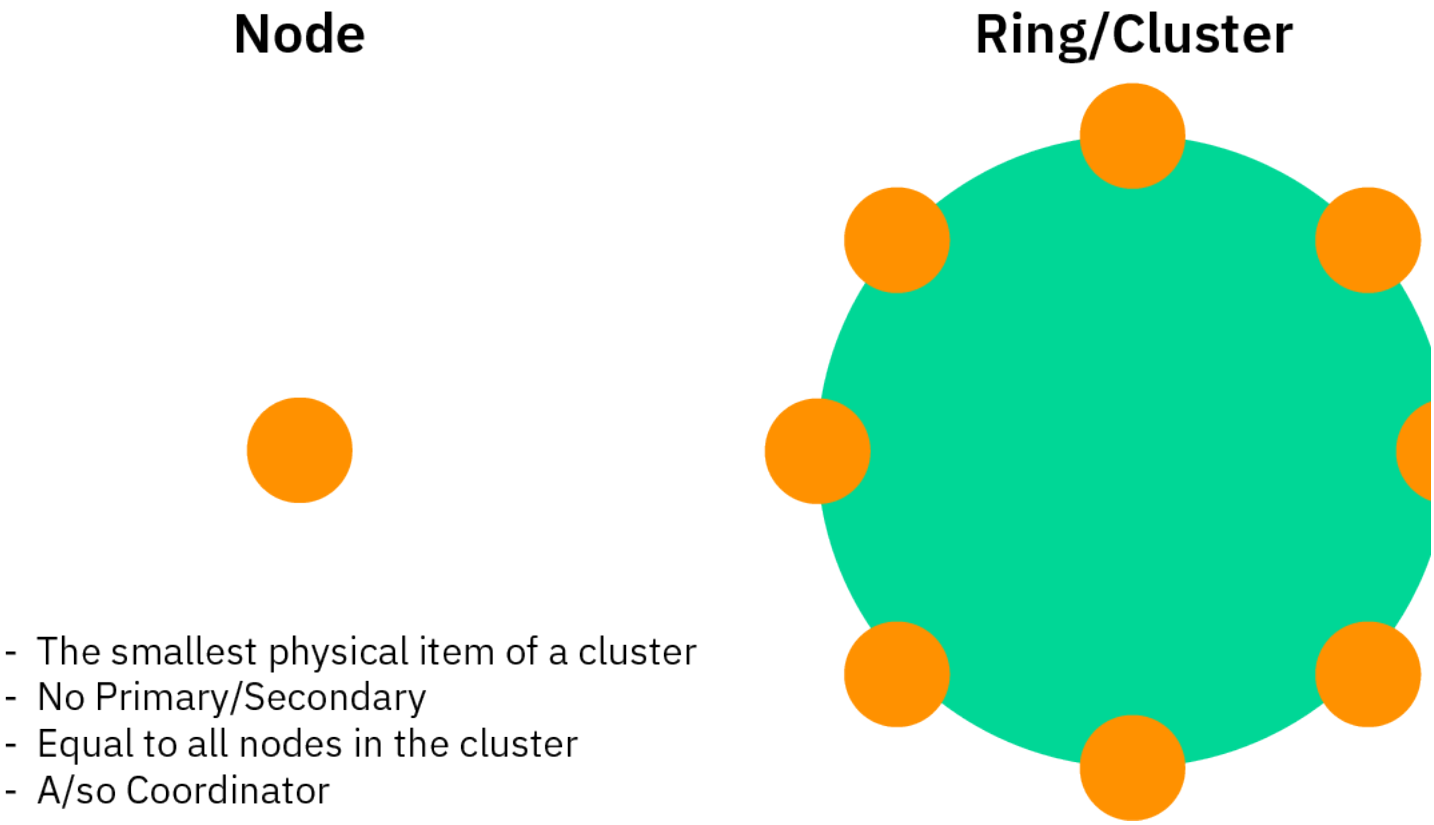
Architecture of Cassandra

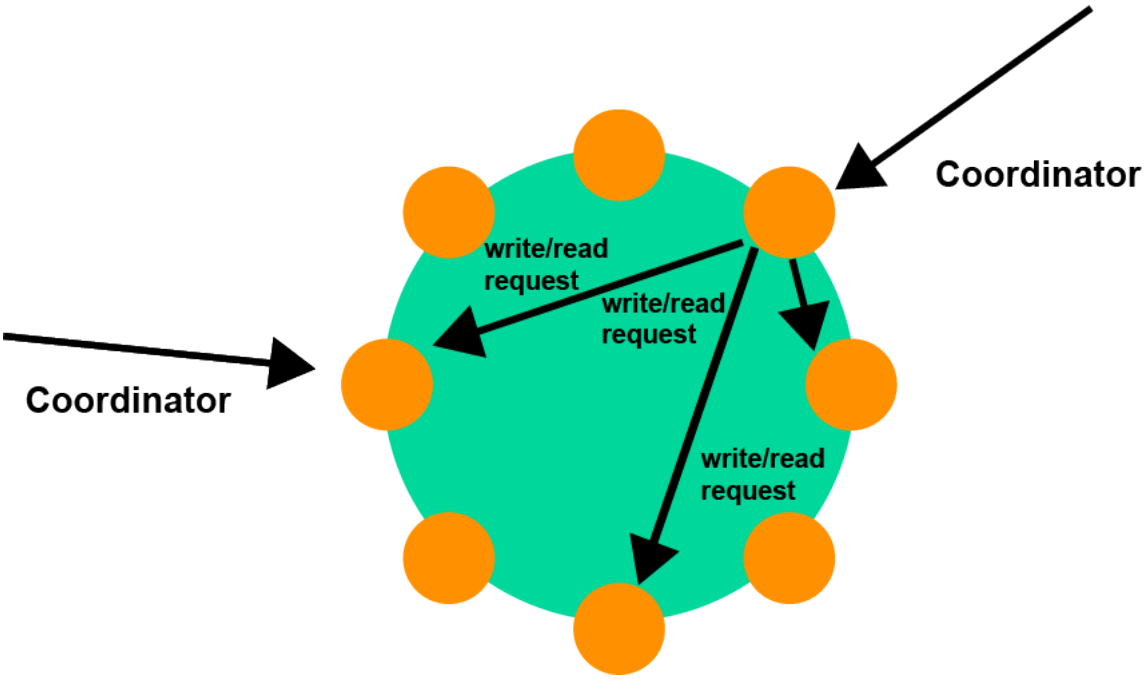
Cassandra is a popular open-source distributed NoSQL database system known for its scalability, high performance, and fault tolerance. While it doesn't have a traditional architectural structure like a building, it does have a robust design that allows it to handle large amounts of data across multiple nodes. After reading this document, you will have a basic understanding of the components.

Apache Cassandra topology

Cassandra is a distributed system architecture. The basic component of Cassandra's architecture is the standalone unit, node. It is also known as a single Cassandra. Nodes can be added or removed without affecting the system's availability. Each node operates independently and communicates with other nodes through a peer-to-peer protocol.



As well as being a distributed system, Cassandra is designed to be a peer-to-peer architecture, with each node connected to all other nodes. Every Cassandra node can run all database operations and handle client requests independently, eliminating the need for a primary node.



How do the nodes in this peer-to-peer architecture know to which node to route a request without a primary node? What if a certain node is down or up? Through the gossip protocol.

The gossip protocol enables nodes to exchange details and information, updating each node about the status of all other nodes. A node performs gossip communications with up to three other nodes every second. The gossip messages follow a specific format and use version numbers to communicate efficiently. So, each node can build the entire metadata of the cluster (which nodes are up/down, what the tokens allocated to each node are, and so on).

Components of a Cassandra node

Several components in Cassandra nodes are involved in the write and read operations. Some of them are listed next:

Memtable

Memtables serve as in-memory structures within Cassandra, buffering write operations before being written onto disk. Typically, each table has an active Memtable. Eventually, these Memtables are flushed to disk, transforming into immutable SSTables (Sorted String Tables).

The triggering of Memtable flushes can occur through various methods:

- Exceeding a predefined threshold for Memtable memory usage.
- Approaching the maximum size of the CommitLog, which prompts Memtable flushes to free up CommitLog segments.
- Setting specific time intervals to trigger flushes on a per-table basis.

These triggers initiate the process where the buffered data in Memtables is persisted onto disk as SSTables, ensuring data durability and efficient retrieval in the Cassandra database system.

Commit log

Commit logs in Cassandra function as append-only logs, capturing all local mutations on a specific Cassandra node. Before you write data to a Memtable, you must record it in a commit log. This process ensures durability in the event of an unexpected shutdown. Upon restarting, any mutations in the commit log are applied to the Memtables, guaranteeing data consistency and recovery in the Cassandra database system.

SSTables

SSTables (Sorted String Tables) are the immutable data files in Cassandra, storing data persistently on disk. These files are created by flushing memtables or streaming from other nodes. When you generate SSTables, Cassandra initiates the compaction processes to merge multiple SSTables into one. You should be able to see the new SSTable while the older SSTables become eligible for removal.

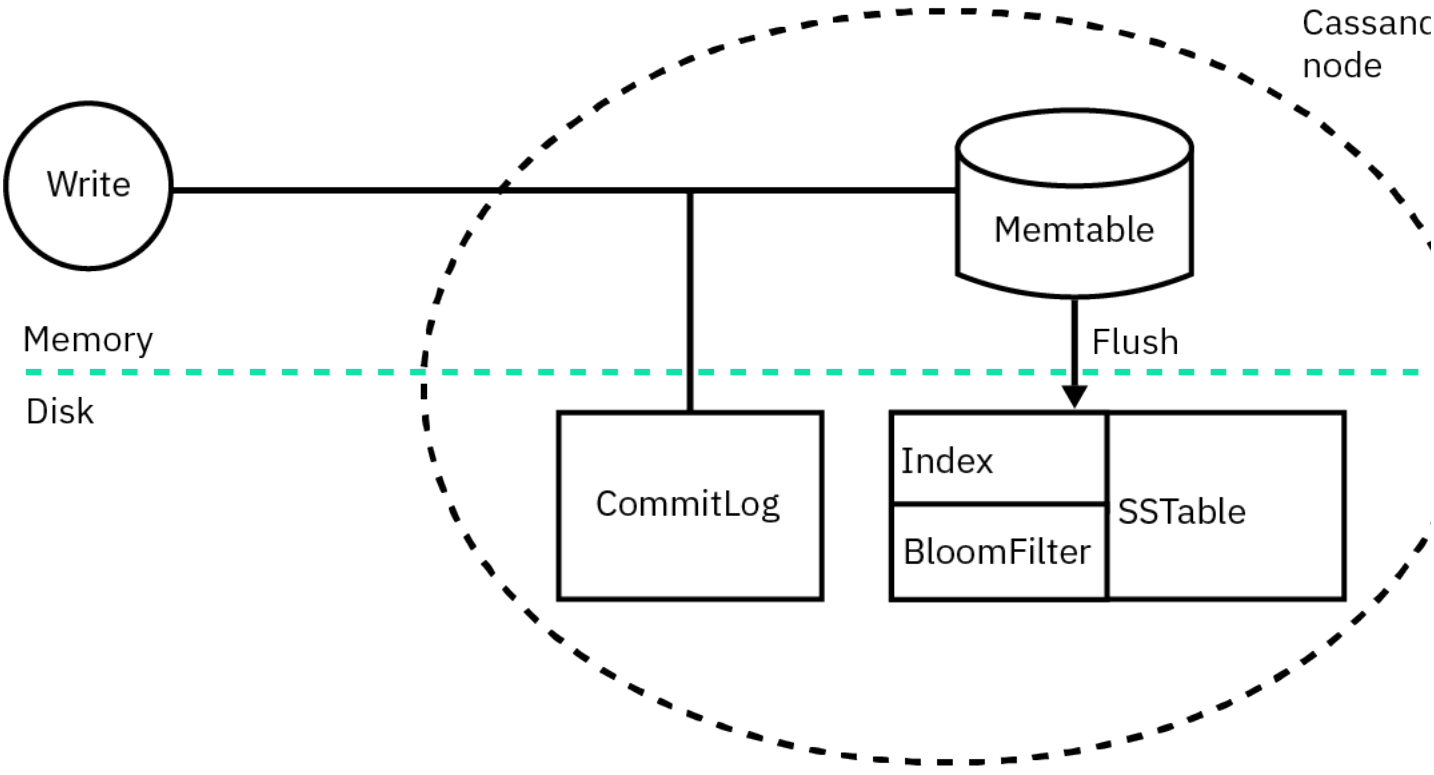
An SSTable comprises various distinct components stored in separate files, some of which include:
Data.db: This file contains the actual data stored by Cassandra.
Index.db: An index file that maps partition keys to specific positions within the Data.db file, aiding in efficient data retrieval.
Summary.db: This file provides a sample subset (typically every 128th entry) of the information contained in the Index.db file, offering an overview for quicker data access.
Filter.db: Cassandra employs a Bloom Filter in this file, which serves as a probabilistic data structure, assisting in determining if a partition key exists in the SSTable without requiring a disk seek.
CompressionInfo.db: This file holds metadata regarding the offsets and lengths of compressed chunks within the Data.db file, facilitating the decompression of stored data.

These distinct files within an SSTable collectively form an organized structure that enables efficient data storage, indexing, retrieval, and compression within the Cassandra database system.

Write process at node level

Writes are distributed across nodes in the cluster, utilizing a coordinator node that manages the operation and replicates data to appropriate replicas based on the configured consistency level.

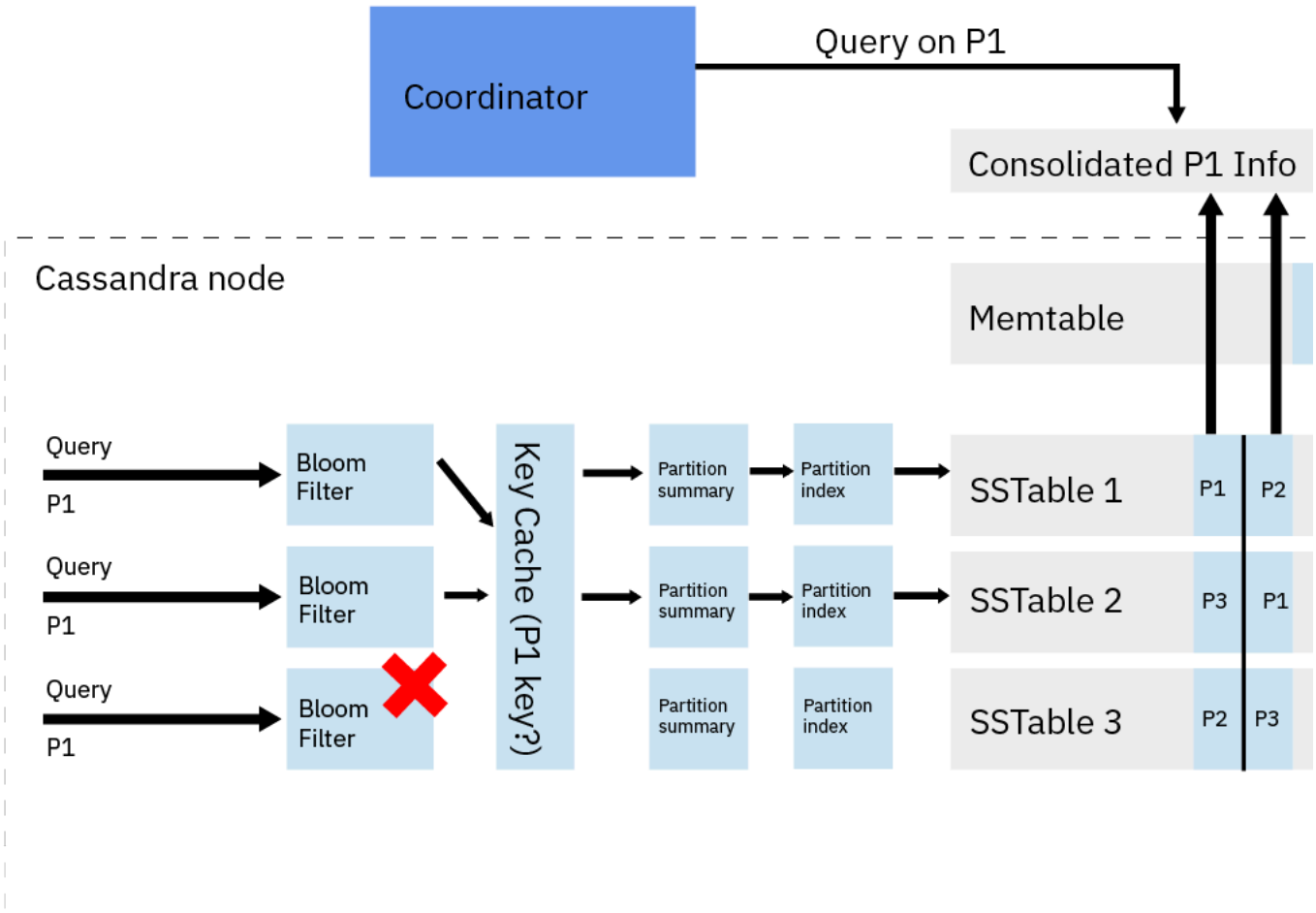
- Logging data in the commit log
- Writing data to the Memtable
- Flushing data from the Memtable
- Storing data on disk in SSTables



Read at node level

While writes in Cassandra are very simple and fast operations done in memory, the read is a bit more complicated since it needs to consolidate data from memory (Memtable) and disk (SSTables). Since you can fragment data on disk into several SSTables, the reading process needs to identify which SSTables most likely contain info about the querying partitions. The Bloom Filter information makes this selection through the following steps:

1. Checks the Memtable
2. Checks Bloom filter
3. Checks partition key cache, if enabled
4. If the partition is not in the cache, the partition summary is checked
5. Then, the partition index is accessed
6. Locates the data on the disk
7. Fetches the data from the SSTable on disk
8. Data is consolidated from Memtable and SSTables before being sent to the coordinator



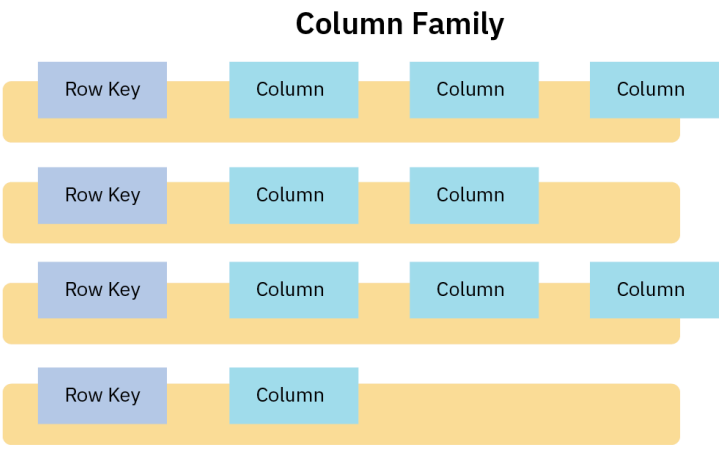
Data distribution

Partitioner: Cassandra uses a partitioner to distribute data across nodes. This consistent hashing algorithm ensures an even data distribution across the cluster, preventing hotspots and facilitating horizontal scaling.

![Partitioner](/images/partition.png width="600")
Token Ring: Cassandra employs a token ring mechanism, assigning a range of tokens to each node in the cluster. Each token corresponds to a particular data range, allowing nodes to locate and manage data efficiently.

Data model

Column family data model: Cassandra follows a column-family-based data model, organizing data into rows and columns. It stores data in tables composed of rows indexed by a unique key. Each row contains columns with different attributes.



Replication and consistency

Replication: Data replication is configurable in Cassandra. Each piece of data is replicated across multiple nodes (replication factor) to ensure fault tolerance and high availability, even in node failures.

Consistency levels: Cassandra offers tunable consistency levels for read and write operations. Users can choose between different consistency levels, balancing consistency and performance based on application requirements.

Read and Write Operations

Distributed writes: Writes are distributed across nodes in the cluster, utilizing a coordinator node that manages the operation and replicates data to appropriate replicas based on the configured consistency level.

Read optimization: Cassandra supports efficient reads by allowing read operations from multiple replicas. It uses the "hinted handoff" mechanism to ensure consistency in case some replicas are temporarily unavailable.

Architecture Components

Gossip protocol: Cassandra uses the gossip protocol for inter-node communication. Nodes exchange information about the cluster's state, such as membership changes or node failures, ensuring each node stays updated.

Compaction and compression: Cassandra periodically performs compaction to merge and organize data files, optimizing storage and improving performance. It also supports data compression to reduce disk usage.

Snitches and load balancing: Snitches determine network topology, optimizing data replication and routing. Load balancing ensures that queries are evenly distributed across nodes to prevent uneven distribution.

Security

Security features: Cassandra offers security features like authentication, encryption, and access control mechanisms to ensure data integrity, confidentiality, and protection against unauthorized access.

Summary

Cassandra's architecture is designed for high availability, fault tolerance, scalability, and efficient data management across distributed environments, making it a powerful choice for handling large-scale data-intensive applications.