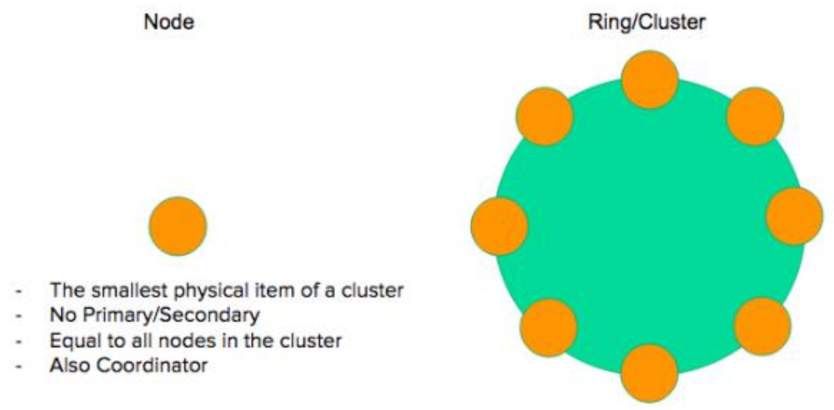
Architecture of Cassandra

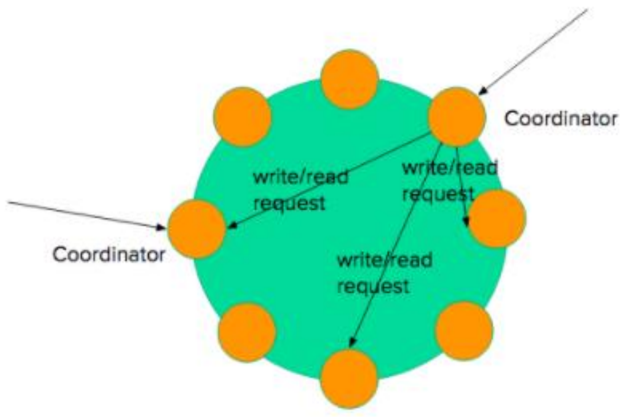
The Apache Cassandra architecture is designed to provide scalability, availability, and reliability to store massive amounts of data. After reading this document, you will have a basic understanding of the components.

Apache Cassandra Topology

Cassandra is based on a distributed system architecture. In its simplest form, Cassandra can be installed on a single machine or container. A single Cassandra instance is called a node. Cassandra supports horizontal scalability achieved by adding more than one node as a part of a Cassandra cluster.


Node
Ring/Cluster
- The smallest physical item of a cluster
- No Primary/Secondary
- Equal to all nodes in the cluster
- Also Coordinator

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. Each Cassandra node can perform all database operations and can serve client requests without the need for a primary node.

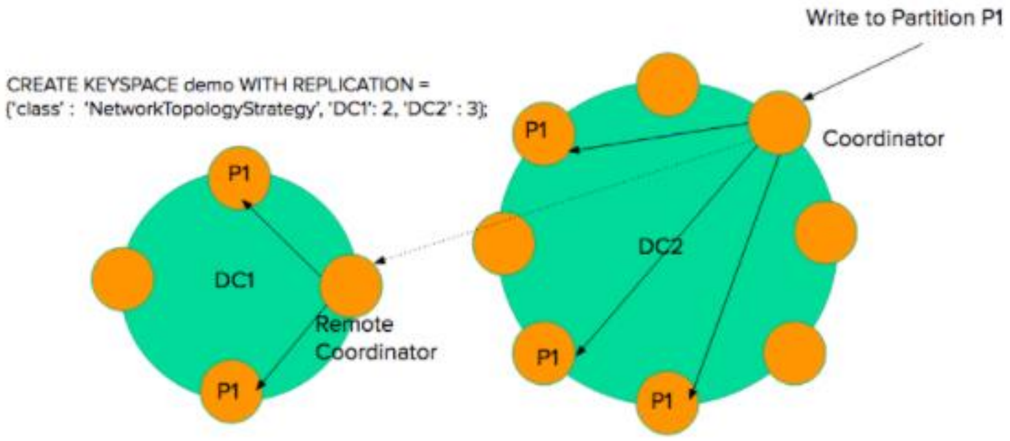


How do the nodes in this peer-to-peer architecture (no primary node) know to which node to route a request and if a certain node is down or up? Through Gossip.

Gossip is the protocol used by Cassandra nodes for peer-to-peer communication. The gossip protocol informs a node about the state 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 make efficient communication, thus shortly each node can build the entire metadata of the cluster (which nodes are up/down, what are the tokens allocated to each node, etc..).

Multi Data Centers Deployment

A Cassandra cluster can be a single data center deployment (like in the above pics), but most of the time Cassandra clusters are deployed in multiple data centers. A multi data-center deployment looks like below – where you can see depicted a 12 nodes Cassandra cluster, topology wise installed in 2 datacenters. Since replication is being set at keyspace level, demo keyspace specifies a replication factor 5: 2 in data center 1 and 3 in data center 2.



```
CREATE KEYSPACE demo WITH REPLICATION =
{'class' :  'NetworkTopologyStrategy', 'DC1': 2, 'DC2' : 3};
```

Note: since a Cassandra node can be as well a coordinator of operations, in our example since the operation came in data center 2 the node receiving the operation becomes the coordinator of the operation, while a node in data center 1 will become the remote coordinator – taking care of the operation in only data center 1.

Components of a Cassandra Node

There are several components in Cassandra nodes that are involved in the write and read operations. Some of them are listed below:

Memtable

Memtables are in-memory structures where Cassandra buffers writes. In general, there is one active Memtable per table. Eventually, Memtables are flushed onto disk and become immutable SSTables.

This can be triggered in several ways:

- The memory usage of the Memtables exceeds a configured threshold.
- The CommitLog approaches its maximum size, and forces Memtable flushes in order to allow Commitlog segments to be freed.
- When we set a time to flush per table.

CommitLog

Commitlogs are an append-only log of all mutations local to a Cassandra node. Any data written to Cassandra will first be written to a commit log before being written to a Memtable. This provides durability in the case of unexpected shutdown. On startup, any mutations in the commit log will be applied to Memtables.

SSTables

SSTables are the immutable data files that Cassandra uses for persisting data on disk.
As SSTables are flushed to disk from Memtables or are streamed from other nodes, Cassandra triggers compactions which combine multiple SSTables into one. Once the new SSTable has been written, the old SSTables can be removed.

Each SSTable is comprised of multiple components stored in separate files, some of which are listed below:

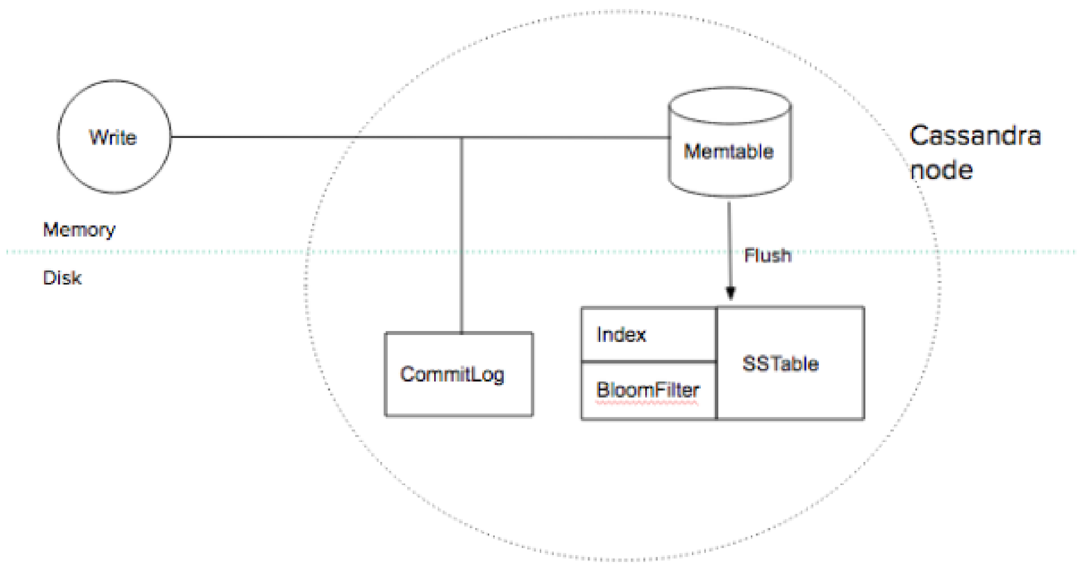
- Data.db: The actual data.
- Index.db: An index from partition keys to positions in the Data.db file.
- Summary.db: A sampling of (by default) every 128th entry in the Index.db file.
- Filter.db: A Bloom Filter of the partition keys in the SSTable.

CompressionInfo.db: Metadata about the offsets and lengths of compression chunks in the Data.db file.

Write Process at Node Level

Cassandra processes data at several stages on the write path, starting with the immediate logging of a write and ending with a write of data to disk:

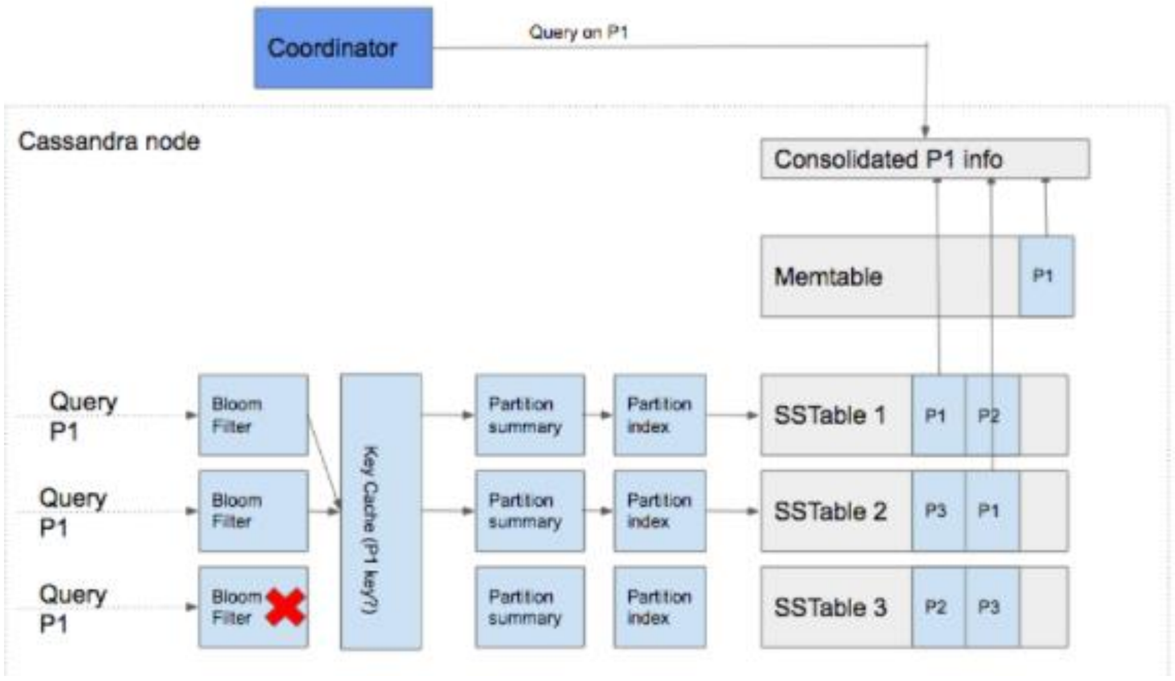
- Logging data in the commit log
- Writing data to the Memtable
- Flushing data from the Memtable
- Storing data on disk in SSTables



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 both memory (Memtable) and disk (SSTables). Since data on disk can be fragmented in several SSTables, the read process needs to identify which SSTables most likely contain info about the partitions we are querying - this selection is done by the Bloom Filter information. The steps are described below:

- Checks the Memtable
- Checks Bloom filter
- Checks partition key cache, if enabled
- If the partition is not in the cache, the partition summary is checked
- Then the partition index is accessed
- Locates the data on disk
- Fetches the data from the SSTable on disk
- Data is consolidated from Memtable and SSTables before being sent to coordinator


Coordinator
Query on P1
Cassandra node
Consolidated P1 info
Memtable
P1
Query P1
Query P1
Query P1
Bloom Filter
Bloom Filter
Bloom Filter
Key Cache (P1 key?)
Partition summary
Partition summary
Partition summary
Partition index
Partition index
Partition index
SSTable 1
P1
P2
SSTable 2
P3
P1
SSTable 3
P2
P3