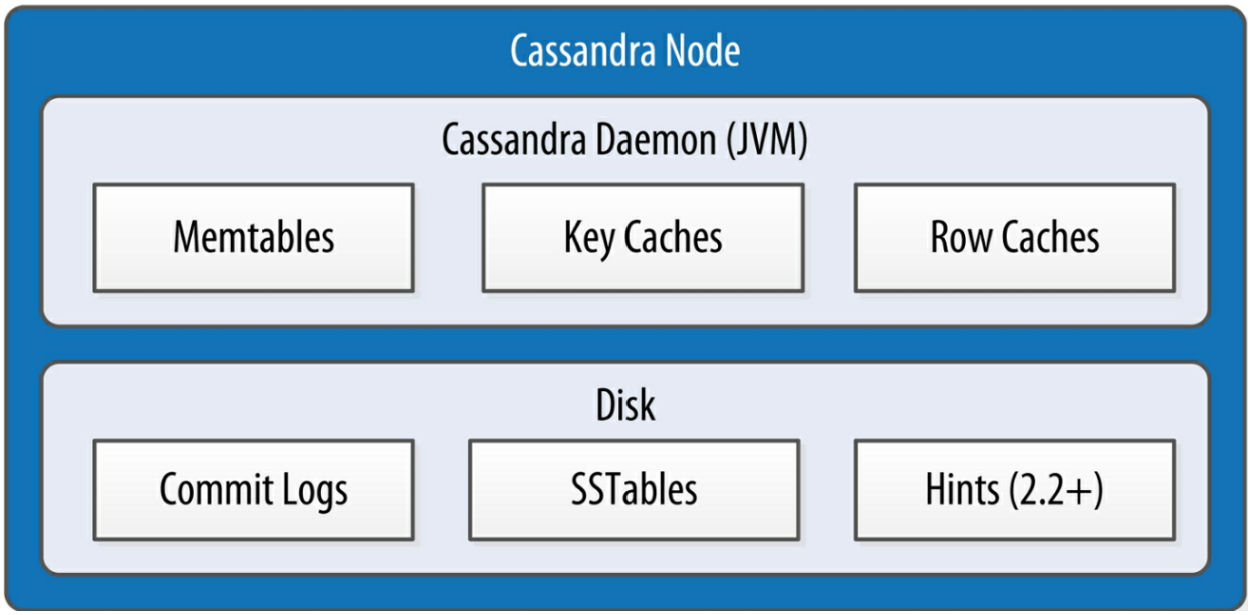
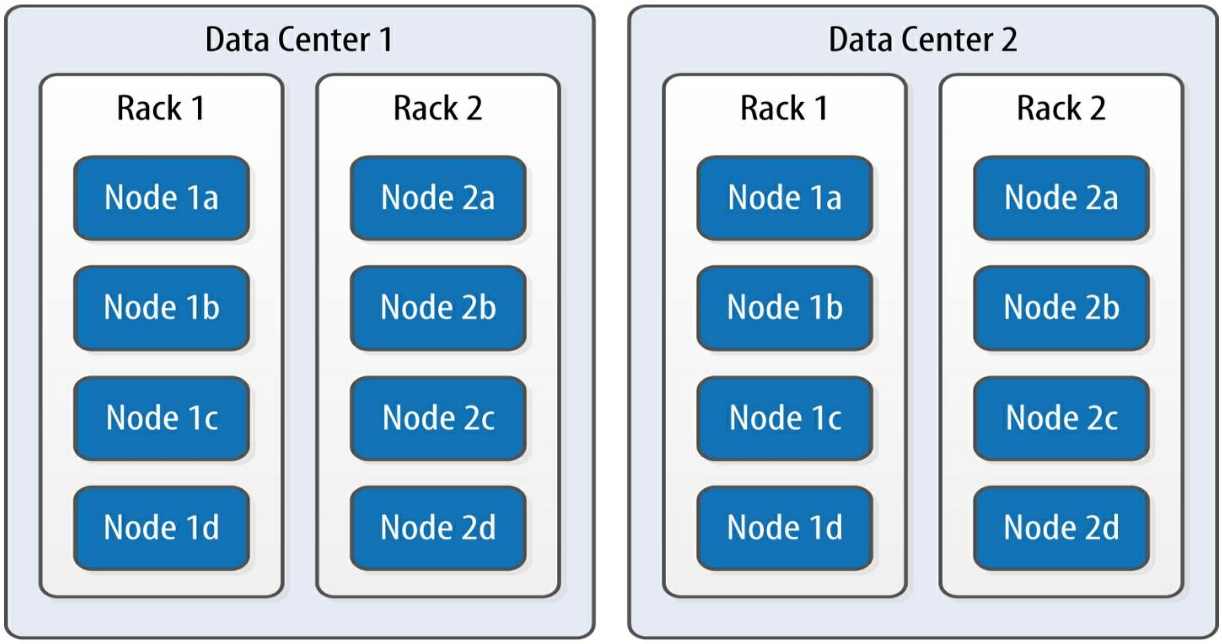
Architecture of Datastax:

1. Node - Physical server running DSE or Virtual Machine with DSE or a DSE container
2. Cluster - a group of nodes, ideally in Odd numbers of nodes
3. Datacenters - Group nodes in close proximity or a particular location
4. Replication - RF=3, Fault tolerence
5. Commit Log (disk) - Data is first written to commit log before to SSTables (per node)
6. MemTable - In memory structure were DSE write the data
7. SSTable - Sorted Strings Table (.db). Immutable. for each table
8. Tombstone - deletion is only marking the row as deleted. during compaction it is deleted
9. Gossip Protocol : (in 9 sec a node failure is detected).
10. Partitioner : murmur3partitioner
11. Replication Factor : replicas of each partition
12. Replica Placement Stratergy : NetworkTolpologyStratergy (SimpleStartergy)
13. Snitch : IP Addresses to Physical or logical location nodes
14. **Bloom Filters:** Probabilistic data structures to quickly check if data exists in an SSTable, reducing disk reads
15. Key Cache & Row Cache : Speed up reads by storing frequently accessed partition keys and full rows in memory

Node Structure:



Datastax Multi DC Cluster:

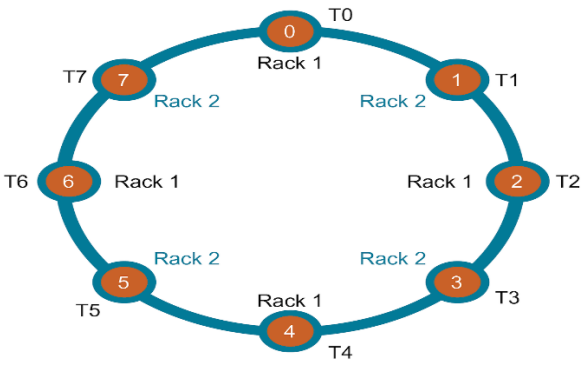


Data Distribution Across Nodes:

Ring: Alll the data managed by the cluster
Token: It is a hash value that derived from Primary Key. Range of token is from -2^{63} to $2^{63}-1$. Token determines which node in the ring contains this data

Data Distribution Tokens (Partitioning)

- **How it Works**: When data is inserted, Cassandra hashes the primary key (partition key + clustering columns) to get a token. This token falls within a large range (0 to 2^{127}).
- **Node Responsibility**: Each node in the cluster owns a segment (range) of these tokens, and it stores all data whose token falls within its assigned range.
- **Token Ranges**: Nodes are assigned initial tokens (e.g., 0, 25, 50, 75 in a simple example) to divide the total token space, ensuring balanced data distribution.



Virtual Nodes (Vnodes) use consistent hashing to divide the data ring into many small, manageable token ranges. Vnodes are assigned ownership of token ranges.
Each physical node will be assigned several Vnodes.

Partitioner : A partitioner determines how data is distributed across the nodes in the cluster. Partitioner creates the hash value from primary key to generate the token for row.

Murmur3Partitioner : the default, recommended for even data distribution via fast hashing. Tokens 2^{63} to 2^{63}-1

RandomPartitioner:legacy, uses slower MD5 hashing for uniform distribution. 0 to 2^{127} tokens.

ByteOrderedPartitioner: legacy, lexical ordering, not recommended due to hot spots.

Replication:

Data Placement: When data is written, its partition key is hashed to a token. This token falls within a specific range, which determines which node is the primary replica owner for that data.

Ownership: Each node knows to which token ranges it the owner and to which ranges other nodes are responsible. (Gossip Protocol).

Primary Replica: The node which owns the range is primary replica and based on RF additional copies of data are placed on other nodes.

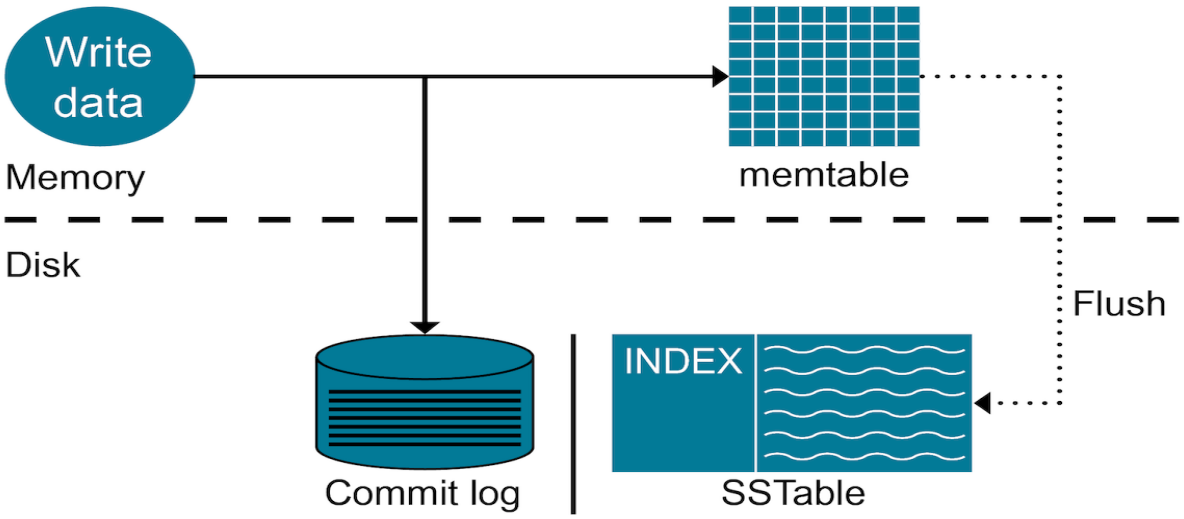
Dynamic Distribution: When nodes are added or removed from the cluster, the token ranges are automatically and dynamically redistributed to ensure the data load remains balanced across all available nodes.

Replication Factor: Determines the number of additional copies of the Range to saved for fault tolerance.

Replication Strategy: Determines the placement of replicas in different Datacenters and Zones.

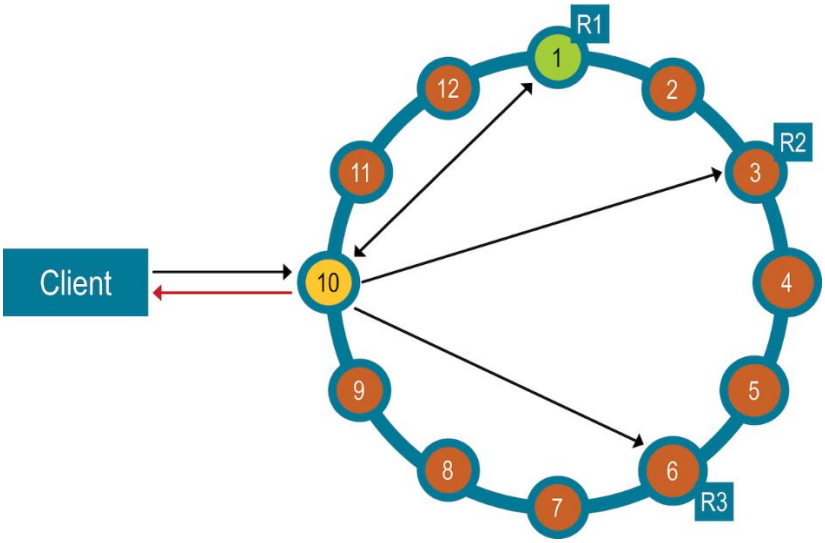
- **SimpleStrategy**: Places replicas sequentially on different nodes (often used for single data centers).
- **NetworkTopologyStrategy**: Places replicas across different racks and data centers, providing better disaster recovery.

Datastax Write Path

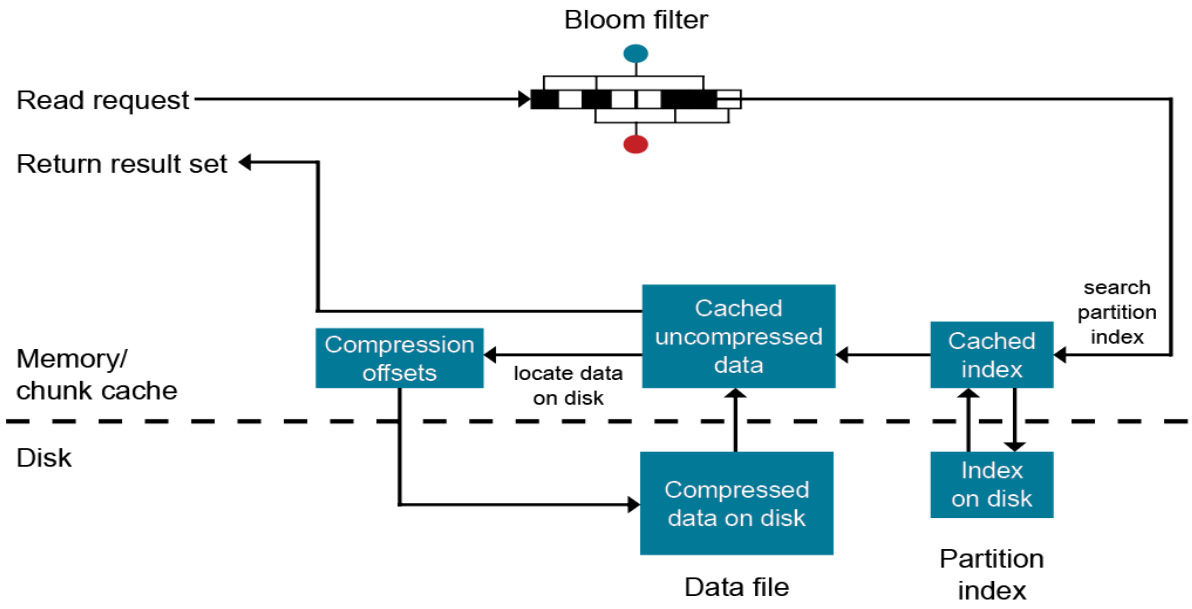


Datastax Read Path:

Client can connect to any node and request the data. This node becomes the coordinator node. It selects the replica nearest to it serve the read the request.



Read Diagram:



Consistency Level:

Read Consistency Levels

Level	Description	Usage
ALL	Returns the record after all replicas have responded. The read operation will fail if a replica does not respond.	Provides the highest consistency of all levels and the lowest availability of all levels.
EACH_QUORUM	Not supported for reads.	
QUORUM	Returns the record after a quorum of replicas from all datacenters has responded.	Used in either single or multiple datacenter clusters to maintain strong consistency across the cluster. Ensures strong consistency if you can tolerate some level of failure.
LOCAL_QUORUM	Returns the record after a quorum of replicas in the current datacenter as the coordinator has reported. Avoids latency of inter-datacenter communication.	Used in multiple datacenter clusters with a rack-aware replica placement strategy (NetworkTopologyStrategy) and a properly configured snitch. Fails when using SimpleStrategy.
ONE	Returns a response from the closest replica, as determined by the snitch. By default, a read repair runs in the background to	Provides the highest availability of all the levels if you can tolerate a comparatively high probability of stale data being read. The replicas contacted for reads may not

Read Consistency Levels

Level	Description	Usage
	make the other replicas consistent.	always have the most recent write.
TWO	Returns the most recent data from two of the closest replicas.	Similar to ONE.
THREE	Returns the most recent data from three of the closest replicas.	Similar to TWO.
LOCAL_ONE	Returns a response from the closest replica in the local datacenter.	Same usage as described in the table about write consistency levels.
SERIAL	Allows reading the current (and possibly uncommitted) state of data without proposing a new addition or update. If a SERIAL read finds an uncommitted transaction in progress, it will commit the transaction as part of the read. Similar to QUORUM.	To read the latest value of a column after a user has invoked a lightweight transaction to write to the column, use SERIAL. Cassandra then checks the inflight lightweight transaction for updates and, if found, returns the latest data.
LOCAL_SERIAL	Same as SERIAL, but confined to the datacenter. Similar to LOCAL_QUORUM.	Used to achieve linearizable consistency for lightweight transactions.

Repair is a critical background maintenance process that synchronizes data across replica nodes to resolve inconsistencies.

Key aspects of Repair in DataStax:

- **Purpose:** To bring replica nodes back into sync, correcting data drift that occurs when nodes are down, data writes are missed, or corruption happens.
- **Mechanism:** It uses anti-entropy, often involving Merkle trees, to compare data ranges between nodes without sending all data, efficiently finding and fixing differences.
- **Types of Repair:**
    - **Full Repair:** Repairs all data for a node or cluster.
    - **Incremental Repair:** Repairs only data that has changed since the last repair, saving resources by skipping already-repaired data.
    - **Hinted Handoff:** Stores hints for writes to offline nodes and delivers them when the node returns.
    - **Read Repair:** Performed during reads when inconsistencies are detected at a certain consistency level (e.g., QUORUM), fixing data on the fly.

System Key spaces:

- **system:** This keyspace contains core Cassandra system tables, including schema definitions for all keyspaces and tables, information about nodes in the cluster, and other fundamental metadata.
- **system_schema:** This keyspace, introduced in later versions of Cassandra, specifically stores schema definitions for user-defined types, functions, aggregates, and tables.
- **system_auth:** This keyspace stores information related to authentication and authorization in DSE. It contains tables for user roles, permissions, and credentials, which are essential for securing the database.
- **system_distributed:** This keyspace is used to store data related to distributed operations and features, such as distributed transactions and other internal coordination mechanisms within a DSE cluster.
- **dse_security:** This DSE-specific keyspace holds security-related information for various DSE features, including DSE Spark, Kerberos digest data, and role options. It is critical for managing the security aspects of DSE.
- **dse_advrep:** It stores metadata and configuration related to advanced replication operations between DSE clusters.
- **dse_perf:** This keyspace is used to store performance-related metrics and data collected by DSE, which can be useful for monitoring and tuning the cluster.
- **dse_leases:** This keyspace is used for internal locking mechanisms and distributed coordination within DSE, particularly for features like DSE Search and DSE Analytics.