Cassandra: Introduction

Let’s explore Cassandra and its use cases.

We'll cover the following

- Goal
- Background
- What is Cassandra?
- Cassandra use cases

Goal#

Design a distributed and scalable system that can store a huge amount of structured data, which is indexed by a row key where each row can have an unbounded number of columns.

Background#

Cassandra is an open-source Apache project. It was originally developed at Facebook in 2007 for their inbox search feature. The Apache Cassandra architecture is designed to provide **scalability**, **availability**, and **reliability** to store large amounts of data. Cassandra combines the distributed nature of **Amazon’s Dynamo** which is a key-value store and the data model of **Google’s BigTable** which is a column-based data store. With Cassandra’s **decentralized architecture**, there is **no single point of failure** in a cluste and its performance can scale linearly with the addition of nodes.

What is Cassandra?#

Cassandra is a **distributed**, **decentralized**, **scalable**, and **highly available** NoSQL database. In terms of CAP theorem, Cassandra is typically classified as an AP (*i.e., available and partition tolerant*) system which means that availability and partition tolerance are generally considered more important than the consistency. Cassandra can be tuned with replication-factor and consistency levels to meet strong consistency requirements, but this comes with a performance cost. In other words, data can be highly available with low consistency guarantees, or it can be highly consistent with lower availability. Cassandra uses **peer-to-peer architecture**, with each node connected to all other nodes. Each Cassandra node performs all database operations and can serve client requests without the need for any leader node.

> **Disclaimer:** All of the following lessons are Cassandra version agnostic and try to explore the general design and architectural layout of different Cassandra components and operations.

Cassandra use cases#

By default, Cassandra is not a strongly consistent database (it is eventually consistent), hence, any application where consistency is not a concern can utilize Cassandra. Though Cassandra can support strong consistency, it comes with a performance impact. Cassandra is **optimized for high throughput** and **faster writes**, and can be used for collecting big data for performing real-time analysis. Here are some of its top use cases:

- **Storing key-value data with high availability** - Reddit and Digg use

Cassandra as a persistent store for their data. Cassandra's ability to scale linearly without any downtime makes it very suitable for their growth needs.

- **Time series data model** - Due to its data model and log-structured storage engine, Cassandra benefits from high-performing write operations. This also makes Cassandra well suited for storing and analyzing sequentially captured metrics (i.e., measurements from sensors, application logs, etc.). Such usages take advantage of the fact that columns in a row are determined by the application, not a predefined schema. Each row in a table can contain a different number of columns, and there is no requirement for the column names to match.

- **Write-heavy applications** - Cassandra is especially suited for write-intensive applications such as time-series streaming services, sensor logs, and Internet of Things (IoT) applications.

Back
Mock Interview: Dynamo

Next
High-level Architecture

Completed

Report an Issue

High-level Architecture

This lesson gives a brief overview of Cassandra's architecture.

We'll cover the following

- Cassandra common terms
- High-level architecture
  - Data partitioning
  - Cassandra keys
  - Clustering keys
  - Partitioner
  - Coordinator node

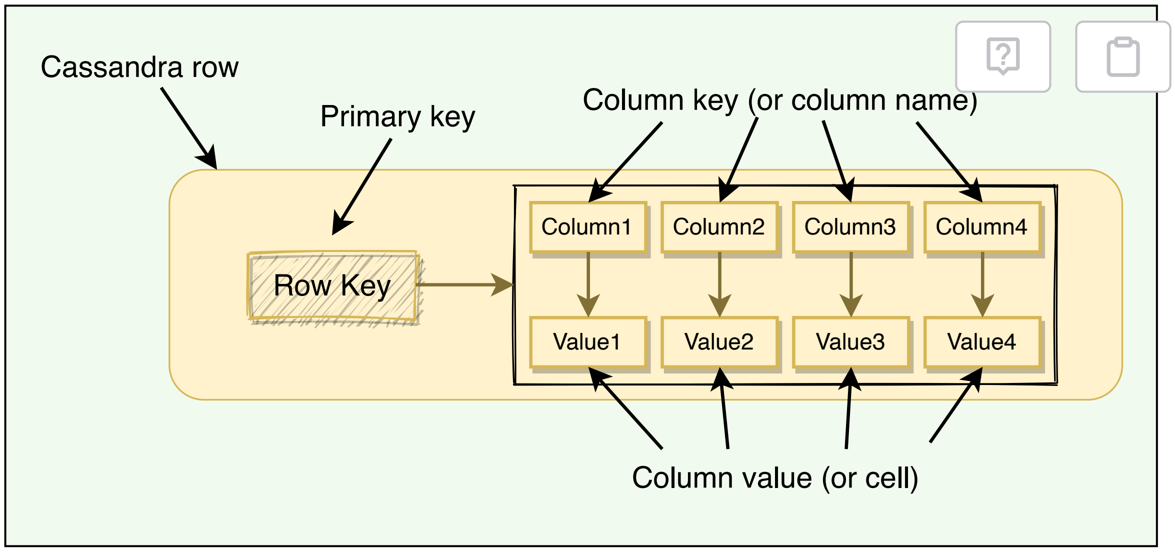
Cassandra common terms#

Before digging deep into Cassandra's architecture, let's first go through some of its common terms:

Column: A column is a key-value pair and is the most basic unit of data structure.

- **Column key:** Uniquely identifies a column in a row.
- **Column value:** Stores one value or a collection of values.

Row: A row is a container for columns referenced by primary key. Cassandra does not store a column that has a null value; this saves a lot of space.



Components of a Cassandra row

Table: A table is a container of rows.

Keyspace: Keyspace is a container for tables that span over one or more Cassandra nodes.

Cluster: Container of Keyspaces is called a cluster.

Node: Node refers to a computer system running an instance of Cassandra. A node can be a physical host, a machine instance in the cloud, or even a Docker container.

NoSQL: Cassandra is a NoSQL database which means we cannot have `joins` between tables, there are no `foreign keys`, and while querying, we cannot add any column in the `where` clause other than the primary key. These constraints should be kept in mind before deciding to use Cassandra.

High-level architecture#

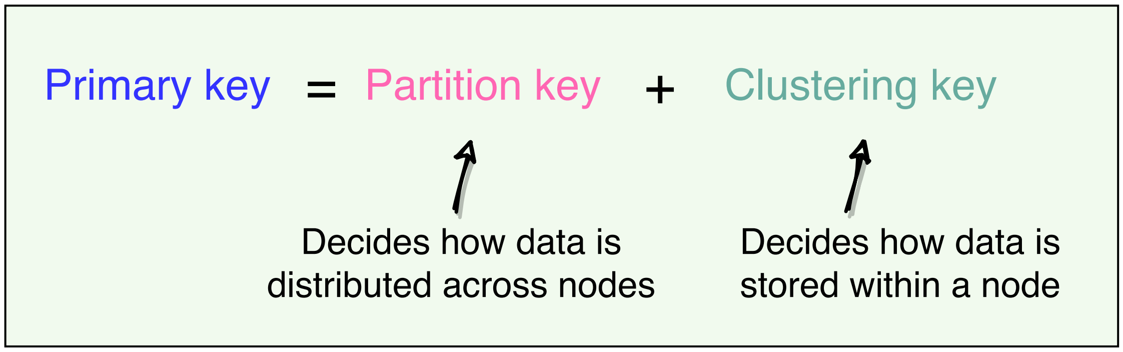
Data partitioning#

Cassandra uses **consistent hashing** for data partitioning. Please take a look at Dynamo's data partitioning; all consistent hashing details described in it applies to Cassandra too.

Let’s look into mechanisms that Cassandra applies to uniquely identify rows.

Cassandra keys#

The **Primary key** uniquely identifies each row of a table. In Cassandra primary key has two parts:


Primary key = Partition key + Clustering key
Decides how data is distributed across nodes
Decides how data is stored within a node

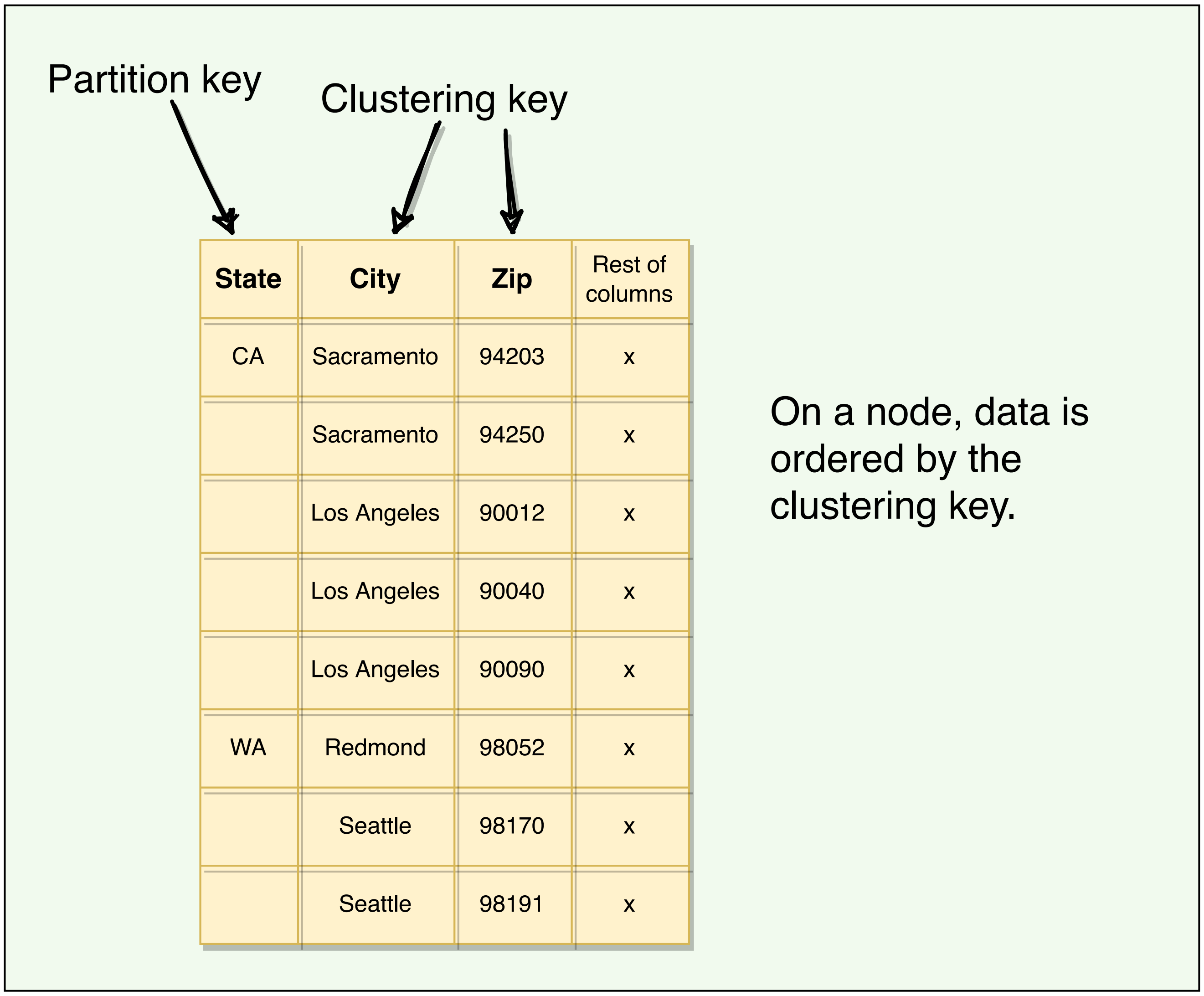
Parts of a primary key

The partition key decides which node stores the data, and the clustering key decides how the data is stored within a node. Let’s take the example of a table with PRIMARY KEY (`city_id`, `employee_id`). This primary key has two parts represented by the two columns:

1. `city_id` is the partition key. This means that the data will be partitioned by the `city_id` field, that is, all rows with the same `city_id` will reside on the same node.
2. `employee_id` is the clustering key. This means that within each node, the data is stored in sorted order according to the `employee_id` column.

Clustering keys#

As described above, clustering keys define how the data is stored within a node. We can have multiple clustering keys; all columns listed after the partition key are called clustering columns. Clustering colum ec e order that the data is arranged on a node.

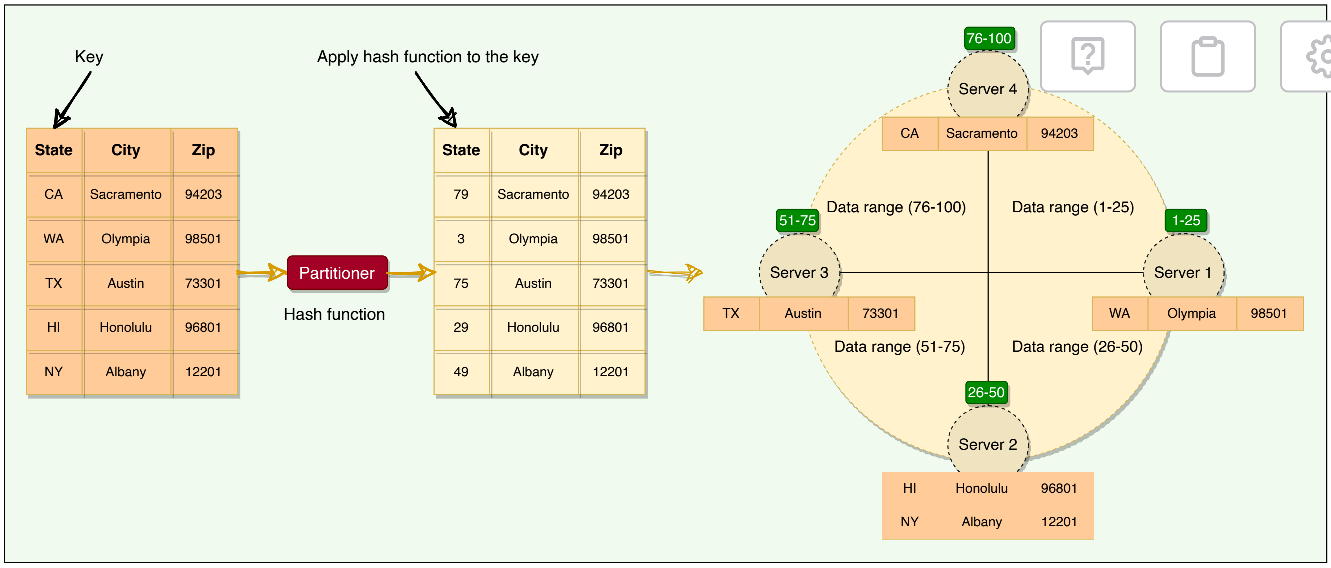


State	City	Zip	Rest of columns
CA	Sacramento	94203	x
	Sacramento	94250	x
	Los Angeles	90012	x
	Los Angeles	90040	x
	Los Angeles	90090	x
WA	Redmond	98052	x
	Seattle	98170	x
	Seattle	98191	x

Clustering key

Partitioner#

Partitioner is the component responsible for determining how data is distributed on the Consistent Hash ring. When Cassandra inserts some data into a cluster, the partitioner performs the first step, which is to apply a hashing algorithm to the partition key. The output of this hashing algorithm determines within which range the data lies and hence, on which node the data will be stored.



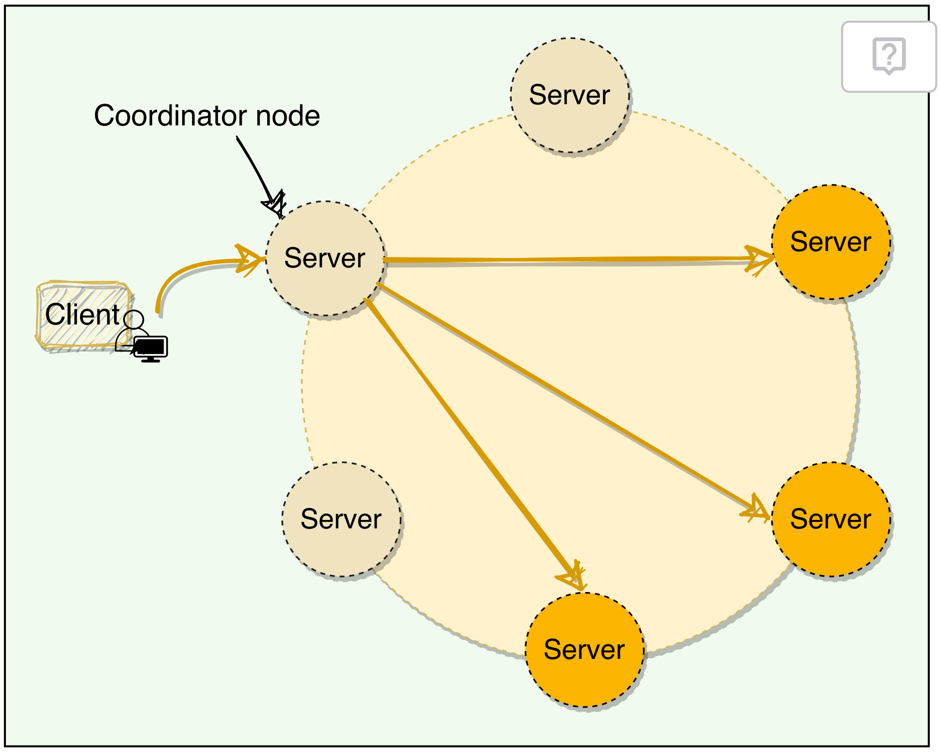
Distributing data on the Consistent Hashing ring

By default, Cassandra uses the Murmur3 hashing function. Murmur3 will always produce the same hash for a given partition key. This means that we can always find the node where a specific row is stored. Cassandra does allow custom hashing functions, however, once a cluster is initialized with a particular partitioner, it cannot be changed later. In Cassandra's default configuration, a token is a 64-bit integer. This gives a possible range for tokens from -2^{63} to $2^{63} - 1$.

All Cassandra nodes learn about the token assignments of other nodes through gossip (discussed later). This means any node can handle a request for any other node's range. The node receiving the request is called the **coordinator**, and any node can act in this role. If a key does not belong to the coordinator's range, it forwards the request to the replicas responsible for that range.

Coordinator node#

A client may connect to any node in the cluster to initiate a read or write query. This node is known as the coordinator node. The coordinator identifies the nodes responsible for the data that is being written or read a
forwards the queries to them.



Client connecting to the coordinator node

As of now, we discussed the core concepts of Cassandra. Let's dig deeper into some of its advanced distributed concepts.

← Back

Cassandra: Introduction

Next →

Replication

✓ Completed

Report an Issue

Replication

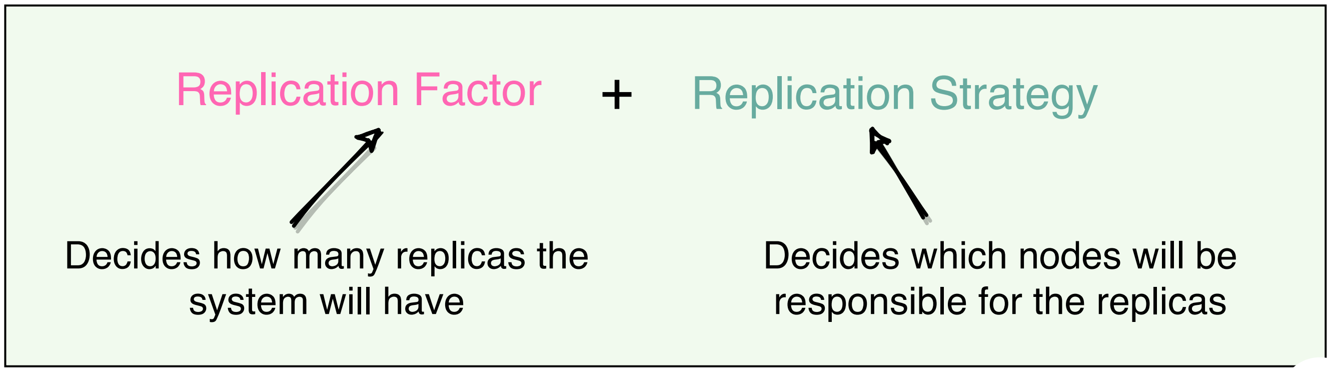
Let's explore Cassandra's replication strategy.

We'll cover the following

- Replication factor
- Replication strategy
  - Simple replication strategy
  - Network topology strategy

Each node in Cassandra serves as a replica for a different range of data. Cassandra stores multiple copies of data and spreads them across various replicas, so that if one node is down, other replicas can respond to queries for that range of data. This process of replicating the data on to different nodes depends upon two factors:

- Replication factor
- Replication strategy


Replication Factor
+
Replication Strategy
Decides how many replicas the system will have
Decides which nodes will be responsible for the replicas

Replication factor#

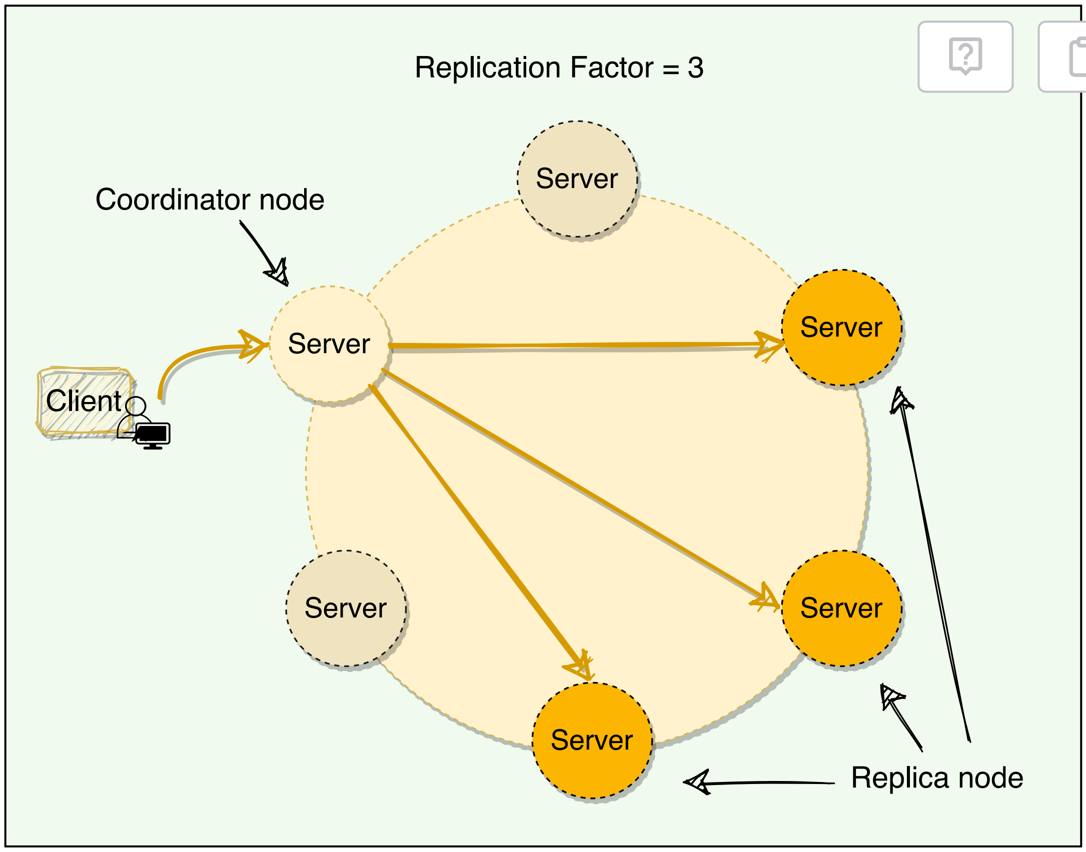
The replication factor is the number of nodes that will receive the copy of the same data. This means, if a cluster has a replication factor of 3, each row will be stored on three different nodes. Each keyspace in Cassandra can have a different replication factor.

Replication strategy#

The node that owns the range in which the hash of the partition key falls will be the first replica; all the additional replicas are placed on the consecutive nodes. Cassandra places the subsequent replicas on the next node in a clockwise manner. There are two replication strategies in Cassandra:

Simple replication strategy#

This strategy is used only for a single data center cluster. Under this strategy, Cassandra places the first replica on a node determined by the partitioner and the subsequent replicas on the next node in a clockwise manner.

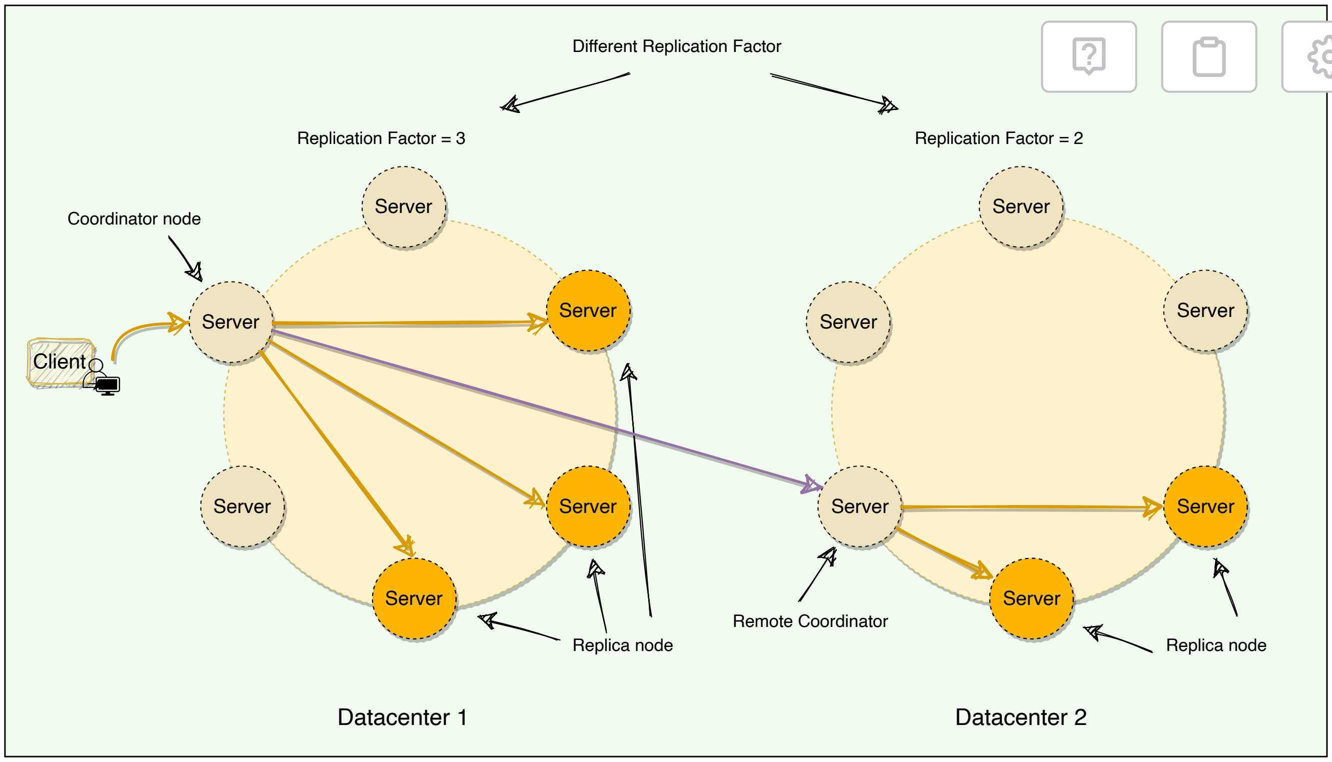


Simple replication with 3 replicas

Network topology strategy#

This strategy is used for multiple data-centers. Under this strategy, we can specify different replication factors for different data-centers. This enables us to specify how many replicas will be placed in each data center.

Additional replicas, in the same data-center, are placed by walking the ring clockwise until reaching the first node in another rack. This is done to guard against a complete rack failure, as nodes in the same rack (or similar physical grouping) tend to fail together due to power, cooling, or network issues.



Network topology strategy for replication

Back

High-level Architecture

Next

Cassandra Consistency Levels

Completed

Report an Issue

Cassandra Consistency Levels

Let's explore how Cassandra manages data consistency.

We'll cover the following

- What are Cassandra’s consistency levels?
- Write consistency levels
  - Hinted handoff
- Read consistency levels
- Snitch

What are Cassandra’s consistency levels?#

Cassandra’s consistency level is defined as the minimum number of Cassandra nodes that must fulfill a read or write operation before the operation can be considered successful. Cassandra allows us to specify different consistency levels for read and write operations. Also, Cassandra has tunable consistency, i.e., we can increase or decrease the consistency levels for each request.

There is always a tradeoff between consistency and performance. A higher consistency level means that more nodes need to respond to a read or write query, giving the user more assurance that the values present on each replica are the same.

Write consistency levels#

For write operations, the consistency level specifies how many replica nodes must respond for the write to be reported as successful to the client. The consistency level is specified per query by the client. Because Cassandra is eventually consistent, updates to other replica nodes may continue in the background. Here are different write consistency levels that Cassandra offers:

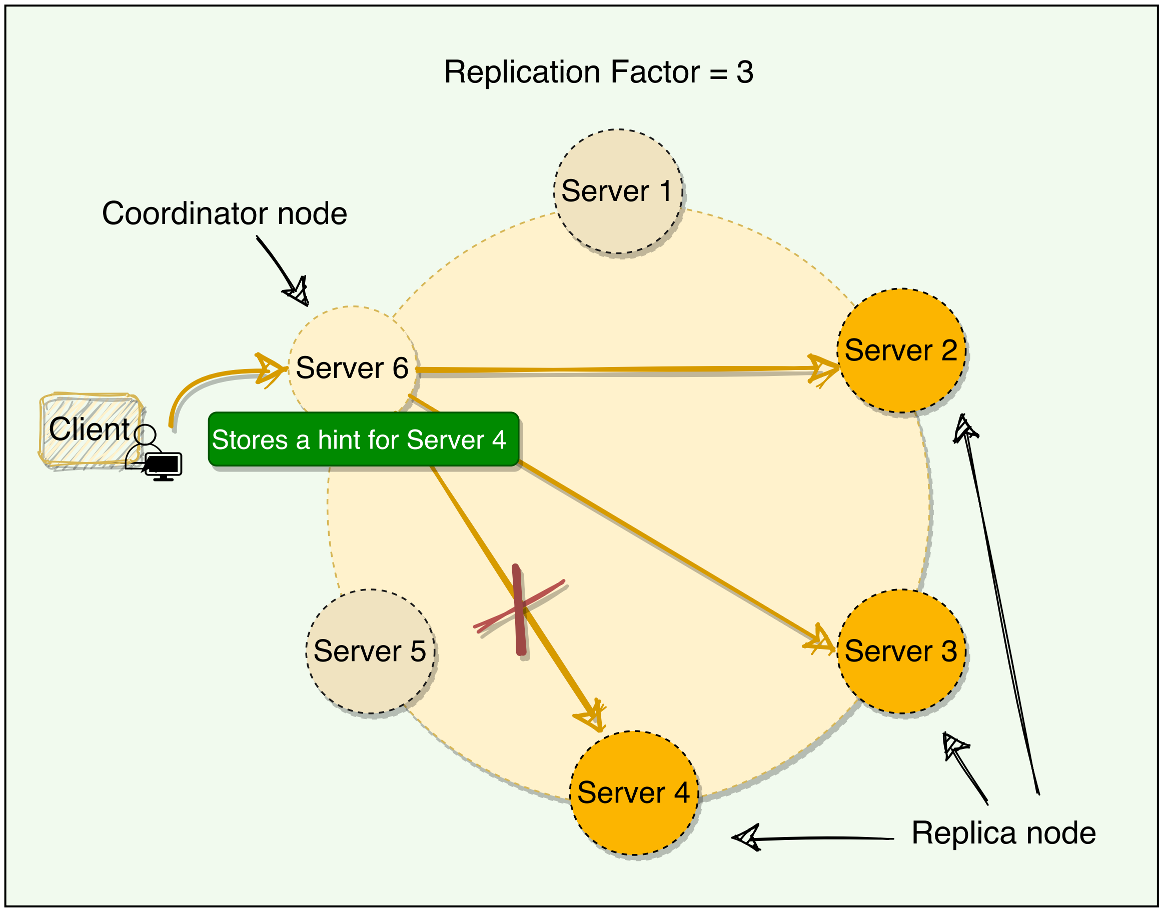
- **One** or **Two** or **Three**: The data must be written to at least the specified number of replica nodes before a write is considered successful.
- **Quorum**: The data must be written to at least a quorum (or majority) of replica nodes. Quorum is defined as $floor(RF/2+1)$, where RF represents the replication factor. For example, in a cluster with a replication factor of five, if three nodes return success, the write is considered successful.
- **All**: Ensures that the data is written to all replica nodes. This consistency level provides the highest consistency but lowest availability as writes will fail if any replica is down.
- **Local_Quorum**: Ensures that the data is written to a quorum of nodes in the same datacenter as the coordinator. It does not wait for the response from the other data-centers.
- **Each_Quorum**: Ensures that the data is written to a quorum of nodes in each datacenter.
- **Any**: The data must be written to at least one node. In the extreme case, when all replica nodes for the given partition key are down, the write can still succeed after a hinted handoff (discussed below) has been written. 'Any' consistency level provides the lowest latency and highest availability, however, it comes with the lowest consistency. If all replica nodes are down at write time, an 'Any' write is not readable until the

replica nodes for that partition have recovered and the d written on them.

How does Cassandra perform a write operation? For a write, the coordinator node contacts all replicas, as determined by the replication factor, and considers the write successful when a number of replicas equal to the consistency level acknowledge the write.

Hinted handoff#

Depending upon the consistency level, Cassandra can still serve write requests even when nodes are down. For example, if we have the replication factor of three and the client is writing with a quorum consistency level. This means that if one of the nodes is down, Cassandra can still write on the remaining two nodes to fulfill the consistency level, hence, making the write successful.



Hinted handoff

Now when the node which was down comes online again, how should we write data to it? Cassandra accomplishes this through hinted handoff.

When a node is down or does not respond to a write request, the coordinator node writes a hint in a text file on the local disk. This hint contains the data itself along with information about which node the data belongs to. When the coordinator node discovers from the Gossiper (will be discussed later) that a node for which it holds hints has recovered, it forwards the write requests for each hint to the target. Furthermore, each node every ten minutes checks to see if the failing node, for which it is holding any hints, has recovered.

With consistency level 'Any,' if all the replica nodes are down, the coordinator node will write the hints for all the nodes and report success to the client. However, this data will not reappear in any subsequent reads until one of the replica nodes comes back online, and the coordinator node successfully forwards the write requests to it. This is assuming that the coordinator node is up when the replica node comes back. This also means that we can lose our data if the coordinator node dies and never comes back. For this reason, we should avoid using the 'Any' consistency level.

If a node is offline for some time, the hints can build up considerably on other nodes. Now, when the failed node comes back online, other nodes tend to flood that node with write requests. This can cause issues on the node, as it is already trying to come back after a failure. To address this problem, Cassandra limits the storage of hints to a configurable time window. It is also possible to disable hinted handoff entirely.

Cassandra, by default, stores hints for three hours. After three hours, older hints will be removed, which means, if now the failed node recovers, it will have stale data. Cassandra can fix this stale data while serving a read request. Cassandra can issue a **Read Repair** when it sees stal a; il go through this while discussing the read path.

One thing to remember: When the cluster cannot meet the consistency level specified by the client, Cassandra fails the write request and does not store a hint.

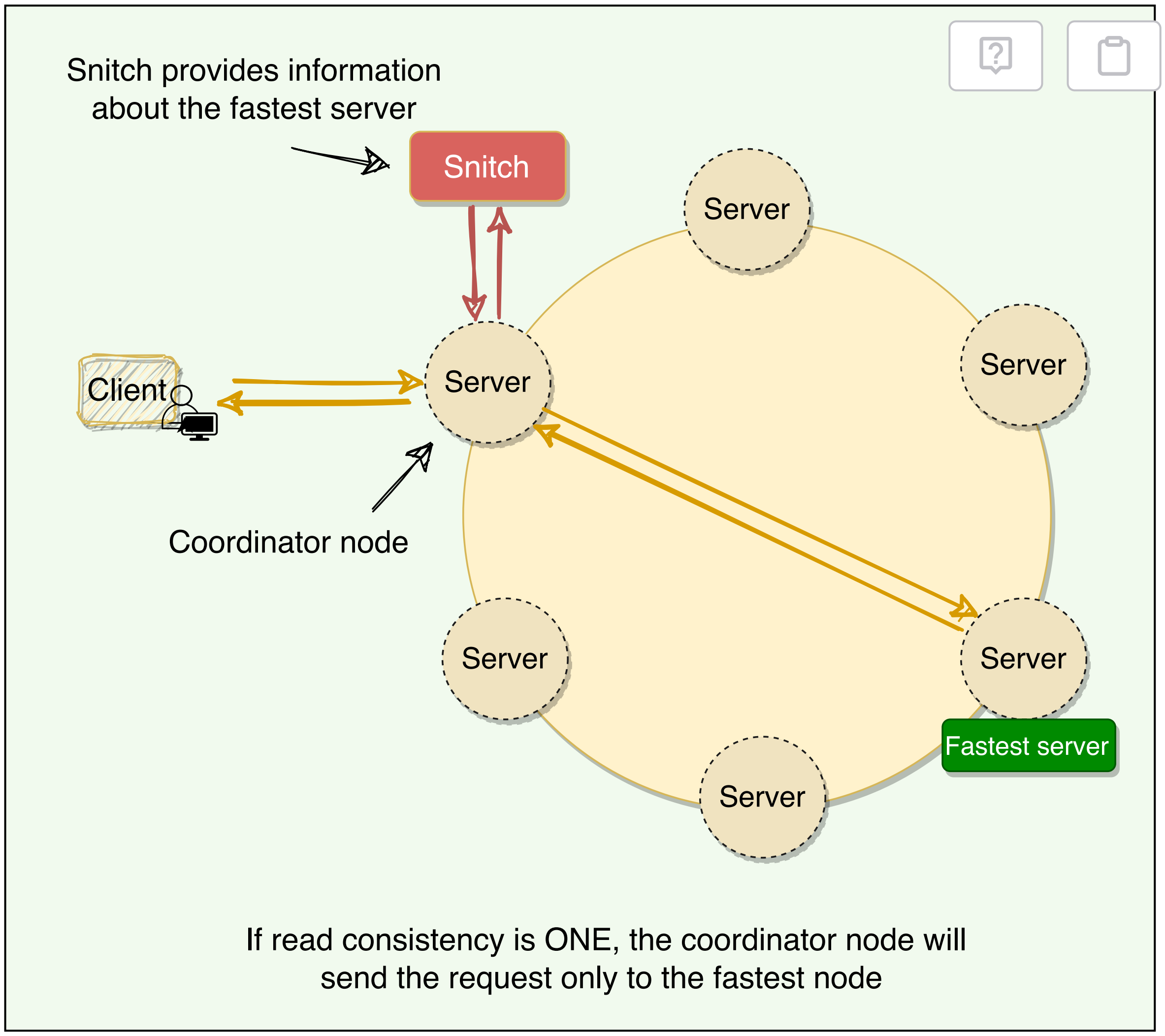
Read consistency levels#

The consistency level for read queries specifies how many replica nodes must respond to a read request before returning the data. For example, for a read request with a consistency level of quorum and replication factor of three, the coordinator waits for successful replies from at least two nodes.

Cassandra has the same consistency levels for read operations as that of write operations except for Each_Quorum (because it is very expensive).

To achieve strong consistency in Cassandra: $R + W > RF$ gives us strong consistency. In this equation, R, W, and RF are the read replica count, the write replica count, and the replication factor, respectively. All client reads will see the most recent write in this scenario, and we will have strong consistency.

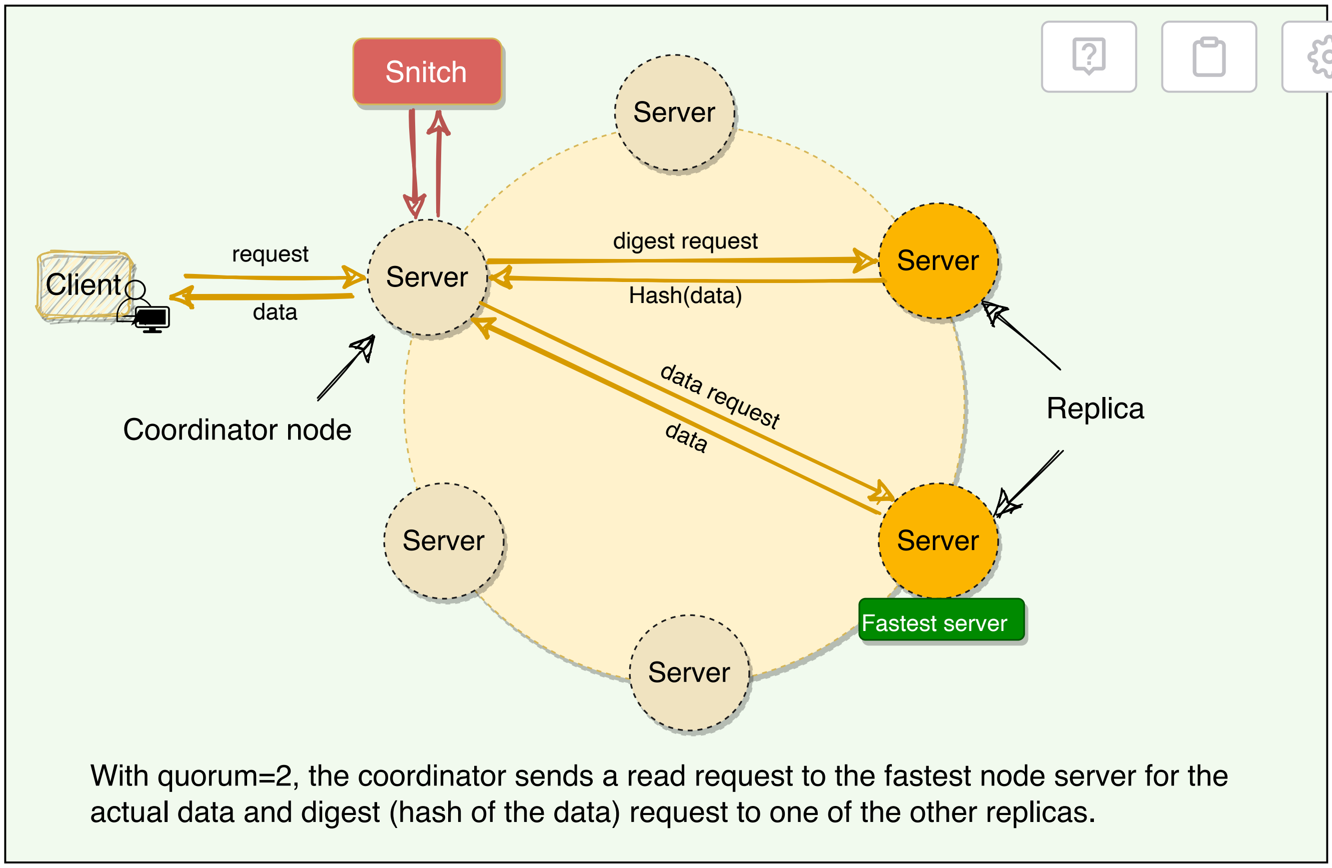
Snitch: The Snitch is an application that determines the proximity of nodes within the ring and also tells which nodes are faster. Cassandra nodes use this information to route read/write requests efficiently. We will discuss this in detail later.


Snitch provides information about the fastest server
Snitch
Server
Server
Client
Server
Coordinator node
Server
Server
Fastest server
Server
If read consistency is ONE, the coordinator node will send the request only to the fastest node

Coordinator node forwards the read request to the fastest server

How does Cassandra perform a read operation? The coordinator always sends the read request to the fastest node. For example, for Quorum=2, the coordinator sends the request to the fastest node and the digest of the data from the second-fastest node. The digest is a checksum of the data and is used to save network bandwidth.

If the digest does not match, it means some replicas do not have the latest version of the data. In this case, the coordinator reads the data from all the replicas to determine the latest data. The coordinator then returns the latest data to the client and initiates a **read repair** request. The read repair operation pushes the newer version of data to nodes with the older version.


Snitch
Server
Client
request
data
Server
digest request
Hash(data)
Server
Replica
Coordinator node
data request
data
Server
Server
Fastest server
Server
With quorum=2, the coordinator sends a read request to the fastest node server for the actual data and digest (hash of the data) request to one of the other replicas.

Read repair

While discussing Cassandra’s write path, we saw that the nodes could become out of sync due to network issues, node failures, corrupted disks, etc. The read repair operation helps nodes to resync with the latest data. Read operation is used as an opportunity to repair inconsistent data across replicas. The latest write-timestamp is used as a marker for the correct version of data. The read repair operation is performed only in a portion of the total reads to avoid performance degradation. Read repairs are opportunistic operations and not a primary operation for anti-entropy.

Read Repair Chance: When the read consistency level is less than ‘All,’ Cassandra performs a read repair probabilistically. By default, Cassandra tries to read repair 10% of all requests with DC local read repair. In this case, Cassandra immediately sends a response when the consistency level is met and performs the read repair asynchronously in the background.

Snitch#

Snitch keeps track of the network topology of Cassandra nodes. It determines which data-centers and racks nodes belong to. Cassandra uses this information to route requests efficiently. Here are the two main functions of a snitch in Cassandra:

- Snitch determines the proximity of nodes within the ring and also monitors the read latencies to avoid reading from nodes that have slowed down. Each node in Cassandra uses this information to route requests efficiently.
- Cassandra's replication strategy uses the information provided by the Snitch to spread the replicas across the cluster intelligently. Cassandra will do its best by not having more than one replica on the same "rack".

To understand Snitch's role, let's take the example of Cassandra's read operation. Let's assume that the client is performing a read with a quorum consistency level, and the data is replicated on five nodes. To support maximum read speed, Cassandra selects a single replica to query for the full object and asks for the digest of the data from two additional nodes in order to ensure that the latest version of the data is returned. The Snitch helps to identify the fastest replica, and Cassandra asks this replica for the full object.

← Back
Replication

Next →
Gossiper

Completed

Report an Issue

Gossiper

Let's explore how Cassandra uses gossip protocol to keep track of the state of the system.

We'll cover the following

- How does Cassandra use gossip protocol?
- Node failure detection

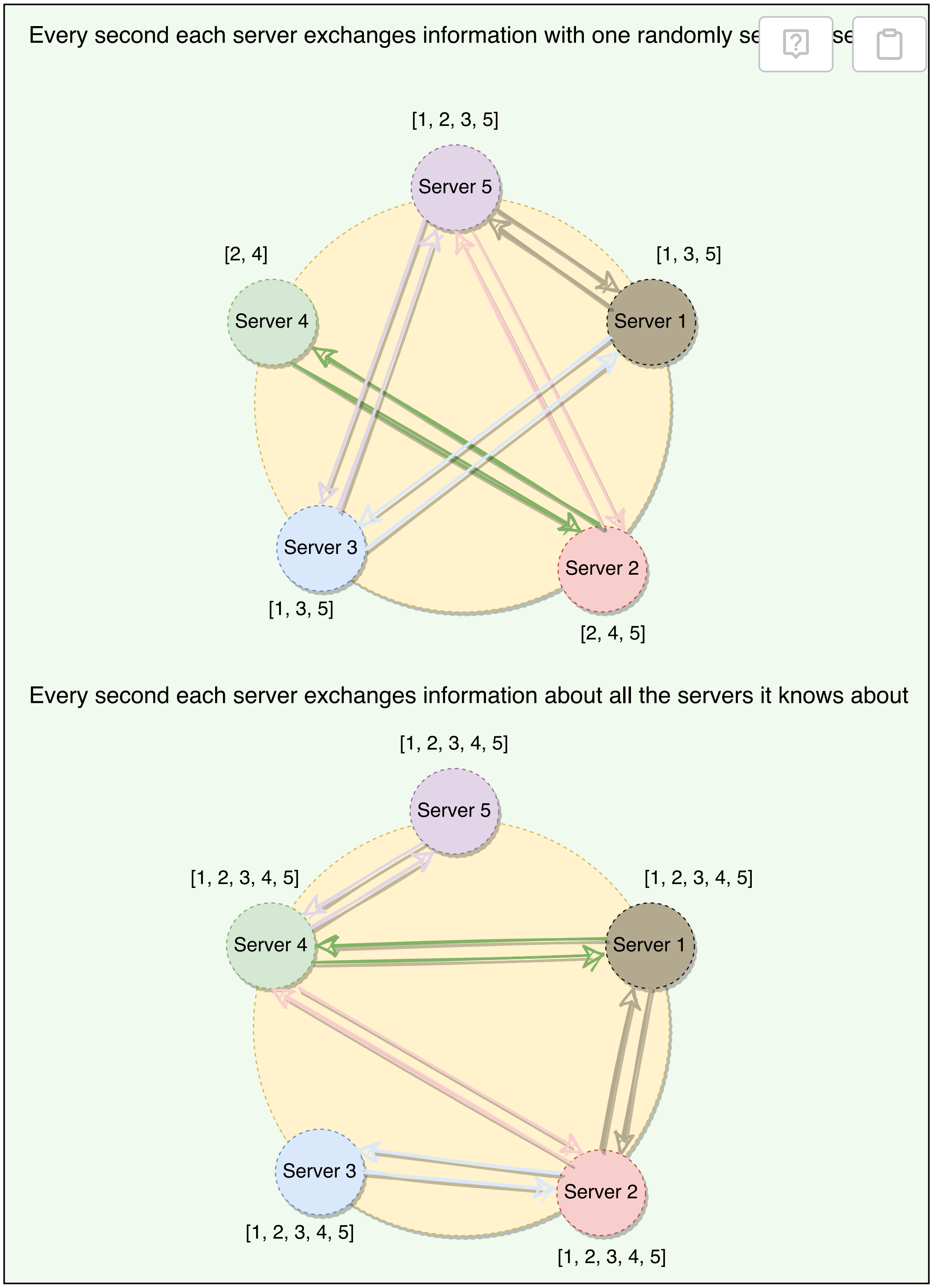
How does Cassandra use gossip protocol?#

Cassandra uses **gossip protocol** that allows each node to keep track of state information about the other nodes in the cluster. Nodes share state information with each other to stay in sync. Gossip protocol is a peer-to-peer communication mechanism in which nodes periodically exchange state information about themselves and other nodes they know about. Each node initiates a gossip round every second to exchange state information about themselves (and other nodes) with one to three other random nodes. This way, all nodes quickly learn about all other nodes in a cluster.

Each gossip message has a version associated with it, so that during a gossip exchange, older information is overwritten with the most current state for a particular node.

Generation number: In Cassandra, each node stores a generation number which is incremented every time a node restarts. This generation number is included in each gossip message exchanged between nodes a u o distinguish the current state of a node from its state before a restart. The generation number remains the same while the node is alive and is incremented each time the node restarts. The node receiving the gossip message can compare the generation number it knows and the gossip message's generation number. If the generation number in the gossip message is higher, it knows that the node was restarted.

Seed nodes: To prevent problems in gossip communications, Cassandra designates a list of nodes as the seeds in a cluster. This is critical for a node starting up for the first time. By default, a node remembers other nodes it has gossiped with between subsequent restarts. The seed node designation has no purpose other than bootstrapping the gossip process for new nodes joining the cluster. Thus, seed nodes are not a single point of failure, nor do they have any other special purpose in cluster operations other than the bootstrapping of nodes.



Gossip protocol

Node failure detection#

Accurately detecting failures is a hard problem to solve as w[illegible] with 100% surety that if a system is genuinely down or is just very slow in responding due to heavy load, network congestion, etc. Mechanisms like Heartbeating outputs a boolean value telling us if the system is alive or not; there is no middle ground. Heartbeating uses a fixed timeout, and if there is no heartbeat from a server, the system, after the timeout, assumes that the server has crashed. Here the value of the timeout is critical. If we keep the timeout short, the system will be able to detect failures quickly but with many false positives due to slow machines or faulty networks. On the other hand, if we keep the timeout long, the false positives will be reduced, but the system will not perform efficiently for being slow in detecting failures.

Cassandra uses an adaptive failure detection mechanism as described by **Phi Accrual Failure Detector**. This algorithm uses historical heartbeat information to make the threshold adaptive. A generic Accrual Failure Detector, instead of telling that the server is alive or not, outputs the suspicion level about a server; a higher suspicion level means there are higher chances that the server is down. Using Phi Accrual Failure Detector, if a node does not respond, its suspicion level is increased and could be declared dead later. As a node's suspicion level increases, the system can gradually decide to stop sending new requests to it. Phi Accrual Failure Detector makes a distributed system efficient as it takes into account fluctuations in the network environment and other intermittent server issues before declaring a system completely dead.

Now that we have discussed Cassandra's major components, let's see how Cassandra performs its read and write operations.

← Back

Cassandra Consistency Levels

Next →

Anatomy of Cassandra's Write Oper

Report an Issue

Anatomy of Cassandra's Write Operation

Let's dig deeper into the components involved in Cassandra's write path.

We'll cover the following

- Commit log
- MemTable
- SStable

Cassandra stores data both in memory and on disk to provide both high performance and durability. Every write includes a timestamp. Write path involves a lot of components, here is the summary of Cassandra's write path:

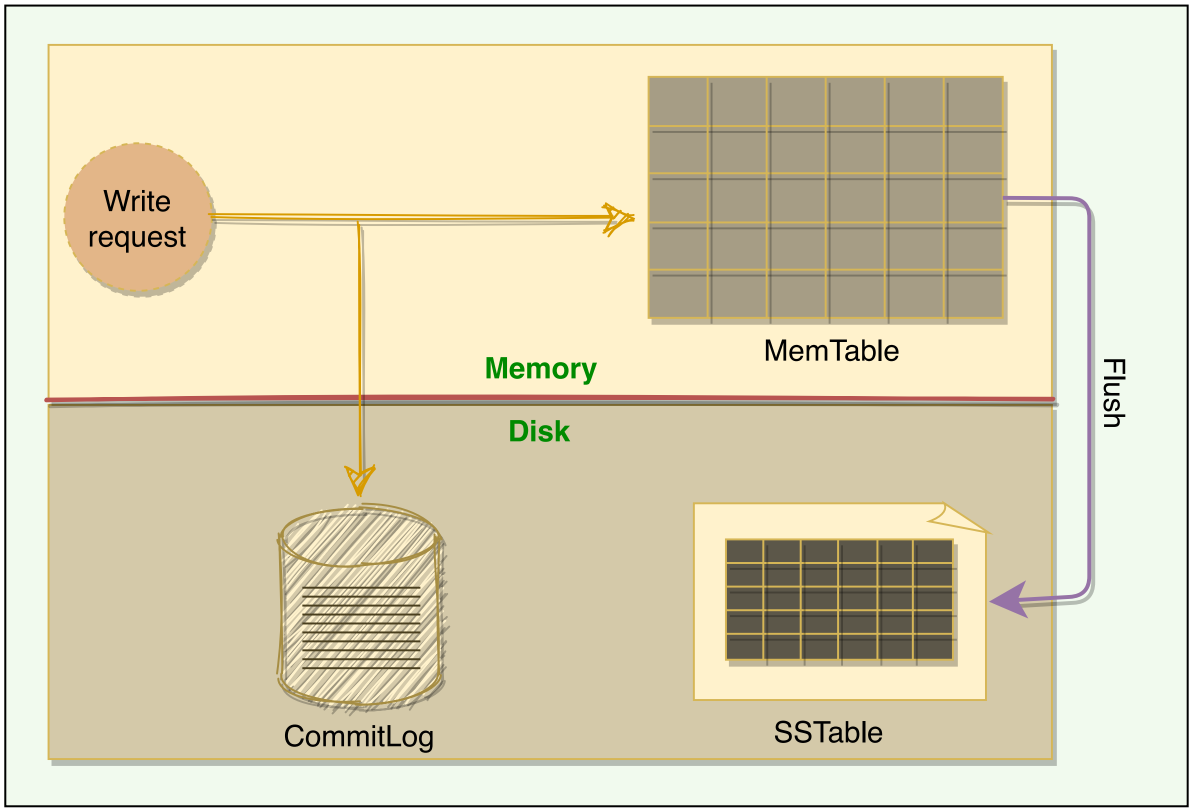
1. Each write is appended to a **commit log**, which is stored on disk.
2. Then it is written to **MemTable** in memory.
3. Periodically, MemTables are flushed to **SSTables** on the disk.
4. Periodically, compaction runs to merge SSTables.

Let's dig deeper into these parts.

Commit log#

When a node receives a write request, it immediately writes the data to a commit log. The commit log is a write-ahead log and is stored on disk. It is used as a crash-recovery mechanism to support Cassandra's durability goals.

A write will not be considered successful on the node until it [illegible] th commit log; this ensures that if a write operation does not make it to the in-memory store (*the MemTable, discussed in a moment*), it will still be possible to recover the data. If we shut down the node or it crashes unexpectedly, the commit log can ensure that data is not lost. That's because if the node restarts, the commit log gets replayed.



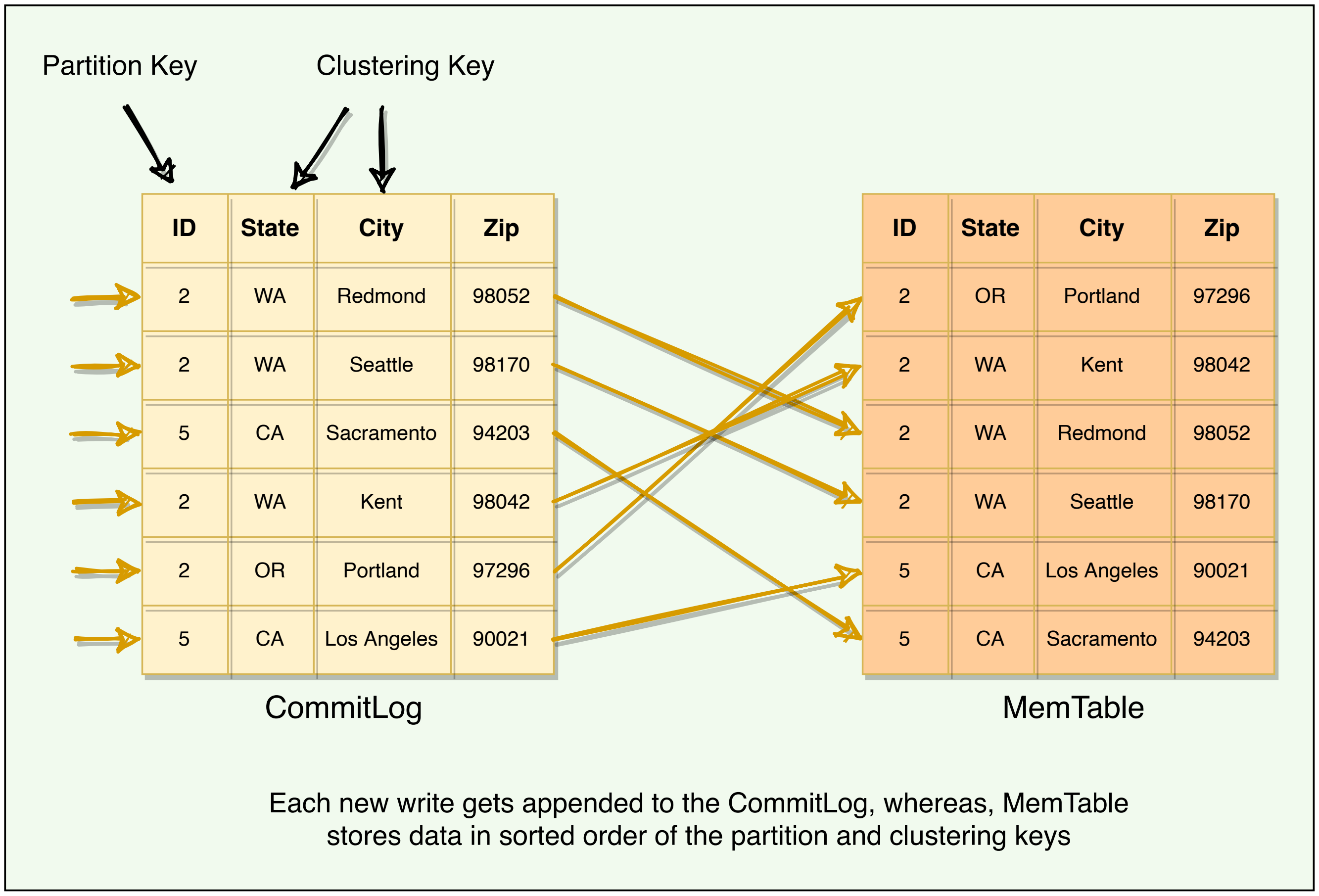
Cassandra's write path

MemTable#

After it is written to the commit log, the data is written to a memory-resident data structure called the MemTable.

- Each node has a MemTable in memory for each Cassandra table.
- Each MemTable contains data for a specific Cassandra table, and it resembles that table in memory.

- Each MemTable accrues writes and provides reads for d ot flushed to disk.
- Commit log stores all the writes in sequential order, with each new write appended to the end, whereas MemTable stores data in the sorted order of partition key and clustering columns.
- After writing data to the Commit Log and MemTable, the node sends an acknowledgment to the coordinator that the data has been successfully written.


Partition Key
Clustering Key
CommitLog
MemTable
Each new write gets appended to the CommitLog, whereas, MemTable stores data in sorted order of the partition and clustering keys

CommitLog:

ID	State	City	Zip
2	WA	Redmond	98052
2	WA	Seattle	98170
5	CA	Sacramento	94203
2	WA	Kent	98042
2	OR	Portland	97296
5	CA	Los Angeles	90021

MemTable:

ID	State	City	Zip
2	OR	Portland	97296
2	WA	Kent	98042
2	WA	Redmond	98052
2	WA	Seattle	98170
5	CA	Los Angeles	90021
5	CA	Sacramento	94203

Storing data to commit log and MemTable

SStable#

When the number of objects stored in the MemTable reaches a threshold, the contents of the MemTable are flushed to disk in a file called SSTable. At this point, a new MemTable is created to store subsequent data. This flushing is a non-blocking operation; multiple MemTables may exist for a [illegible] o current, and the rest waiting to be flushed. Each SStable contains data for a specific table.

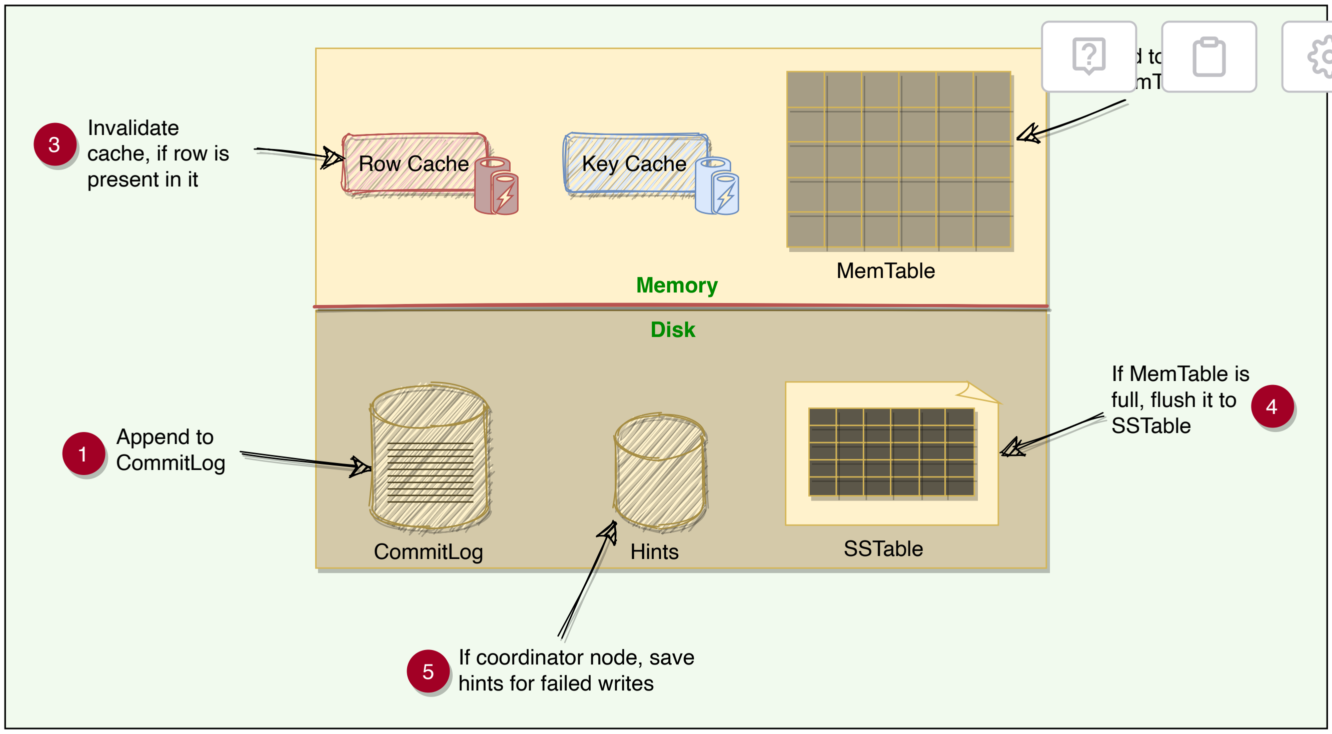
When the MemTable is flushed to SStables, corresponding entries in the Commit Log are removed.

Why are they called ‘SSTables’? The term ‘SSTables’ is short for ‘Sorted String Table’ and first appeared in Google’s Bigtable which is also a storage system. Cassandra borrowed this term even though it does not store data as strings on the disk.

Once a MemTable is flushed to disk as an SSTable, it is immutable and cannot be changed by the application. If we are not allowed to update SSTables, how do we delete or update a column? In Cassandra, each delete or update is considered a new write operation. We will look into this in detail while discussing Tombstones.

The current data state of a Cassandra table consists of its MemTables in memory and SSTables on the disk. Therefore, on reads, Cassandra will read both SSTables and MemTables to find data values, as the MemTable may contain values that have not yet been flushed to the disk. The MemTable works like a write-back cache that Cassandra looks up by key.

Generation number is an index number that is incremented every time a new SSTable is created for a table and is used to uniquely identify SSTables. Here is the summary of Cassandra’s write path:



3 Invalidate cache, if row is present in it
Row Cache
Key Cache
MemTable
Memory
Disk
1 Append to CommitLog
CommitLog
Hints
SSTable
If MemTable is full, flush it to SSTable 4
5 If coordinator node, save hints for failed writes

Anatomy of Cassandra's write path

← Back

Gossiper

Next →

Anatomy of Cassandra's Read Operati...

✓ Completed

Report an Issue

Anatomy of Cassandra's Read Operation

Let’s explore Cassandra’s read path.

We'll cover the following

- Caching
- Reading from MemTable
- Reading from SSTable
    - Bloom filters
    - How are SSTables stored on the disk?
    - Partition index summary file
    - Reading SSTable through key cache

Let’s dig deeper into the components involved in Cassandra’s read path.

Caching#

To boost read performance, Cassandra provides three optional forms of caching:

1. **Row cache:** The row cache, caches frequently read (or hot) rows. It stores a complete data row, which can be returned directly to the client if requested by a read operation. This can significantly speed up read access for frequently accessed rows, at the cost of more memory usage

2. **Key cache:** Key cache stores a map of recently read part[illegible] k[illegible] their SSTable offsets. This facilitates faster read access into SSTables stored on disk and improves the read performance. Key cache takes little memory compared to row cache (where the whole row is stored in memory) and provides a considerable improvement for read operations.
3. **Chunk cache:** Chunk cache is used to store uncompressed chunks of data read from SSTable files that are accessed frequently.

Reading from MemTable#

As we know, data is sorted by the partition key and the clustering columns. Let’s take an example. Here we have two partitions of a table with partition keys ‘2’ and ‘5’. The clustering columns are the state and city names. When a read request comes in, the node performs a binary search on the partition key to find the required partition and then return the row.

Reading data from MemTable

Here is the summary of Cassandra’s read path:

Anatomy of Cassandra's read path

Reading from SSTable#

Bloom filters#

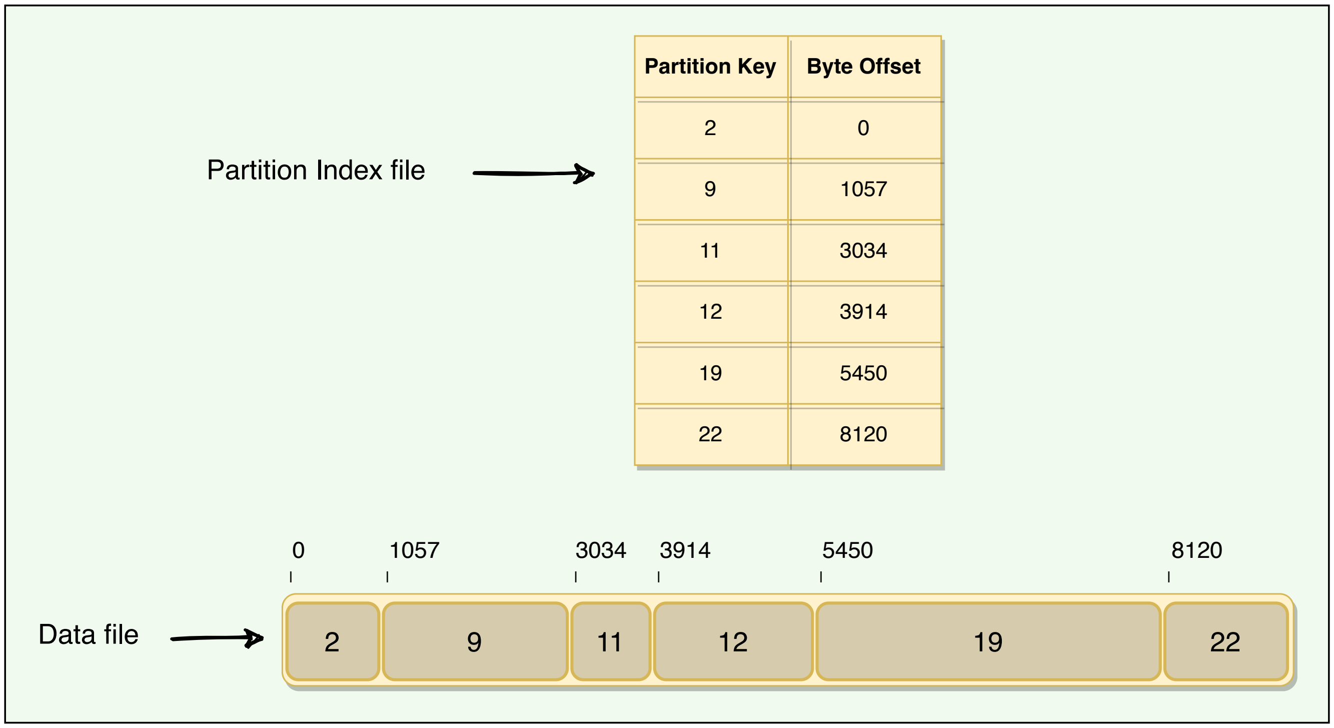
Each SStable has a Bloom filter associated with it, which tells if a particular key is present in it or not. Bloom filters are used to boost the performance of read operations. Bloom filters are very fast, non-deterministic algorithms for testing whether an element is a member of a set. They are non-deterministic because it is possible to get a false-positive read from a Bloom filter, but false-negative is not possible. Bloom filters work by mapping the values in a data set into a bit array and condensing a larger data set into a digest string using a hash function. The digest, by definition, uses a much smaller amount of memory than the original data would. The filters are stored in memory and are used to improve performance by reducing the need for disk access on key lookups. Disk access is typically much slower than memory access. So, in a way, a Bloom filter is a special kind of key cache.

Cassandra maintains a Bloom filter for each SSTable. When a [illegible]ry [illegible] performed, the Bloom filter is checked first before accessing the disk. Because false negatives are not possible, if the filter indicates that the element does not exist in the set, it certainly does not; but if the filter thinks that the element is in the set, the disk is accessed to make sure.

How are SSTables stored on the disk?#

Each SSTable consists of two files:

1. **Data File:** Actual data is stored in a data file. It has partitions and rows associated with those partitions. The partitions are in sorted order.
2. **Partition Index file:** Stored on disk, partition index file stores the sorted partition keys mapped to their SSTable offsets. It enables locating a partition exactly in an SSTable rather than scanning data.



Partition Key	Byte Offset
2	0
9	1057
11	3034
12	3914
19	5450
22	8120

Reading from an SSTable

Partition index summary file#

Stored in memory, the Partition Index Summary file stores th[illegible] of the Partition Index file. This is done for performance improvement.

Reading from partition index summary file

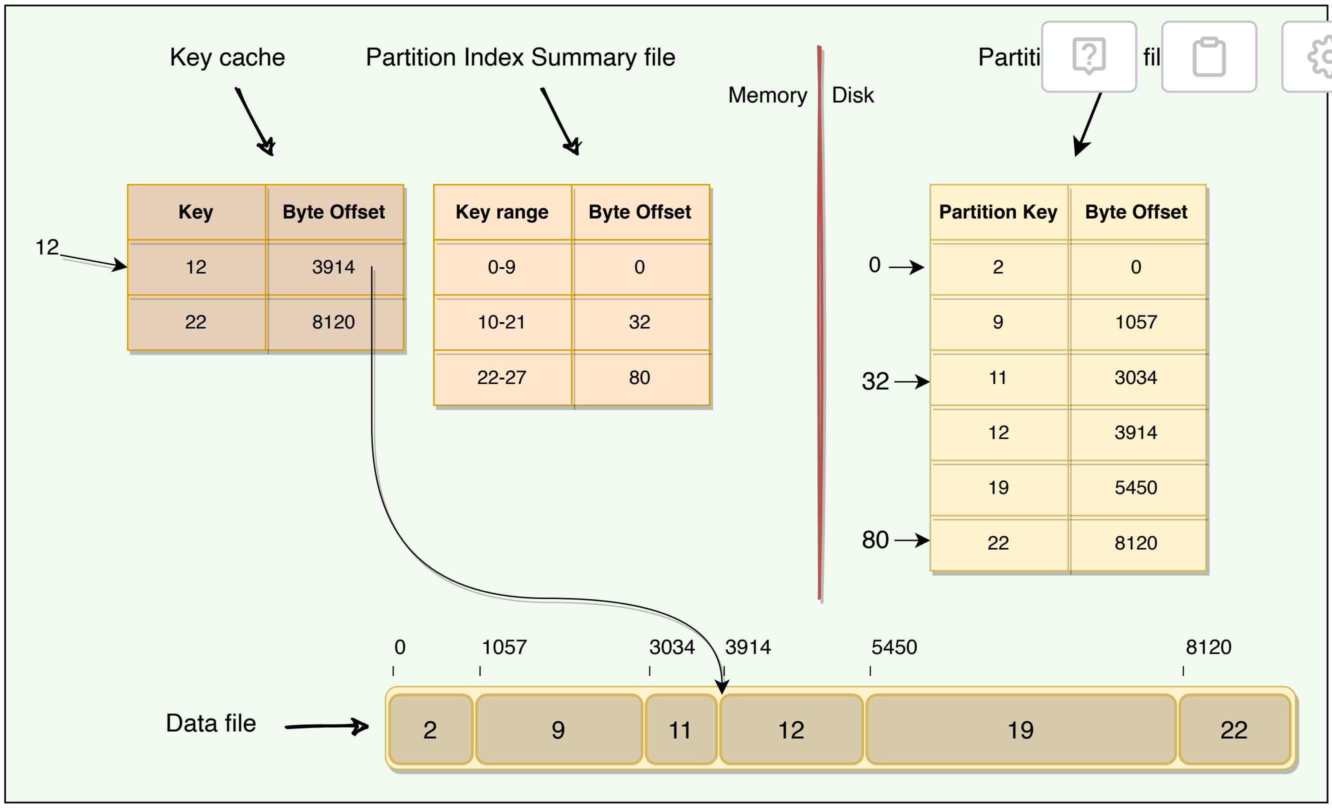
If we want to read data for key=12, here are the steps we need to follow (also shown in the figure below):

1. In the Partition Index Summary file, find the key range in which the key=12 lies. This will give us offset (=32) into the Partition Index file.
2. Jump to offset 32 in the Partition Index file to search for the offset of key=12. This will give us offset (=3914) into the SSTable file.
3. Jump to SSTable at offset 3914 to read the data for key=12

Reading from partition index summary file

Reading SSTable through key cache#

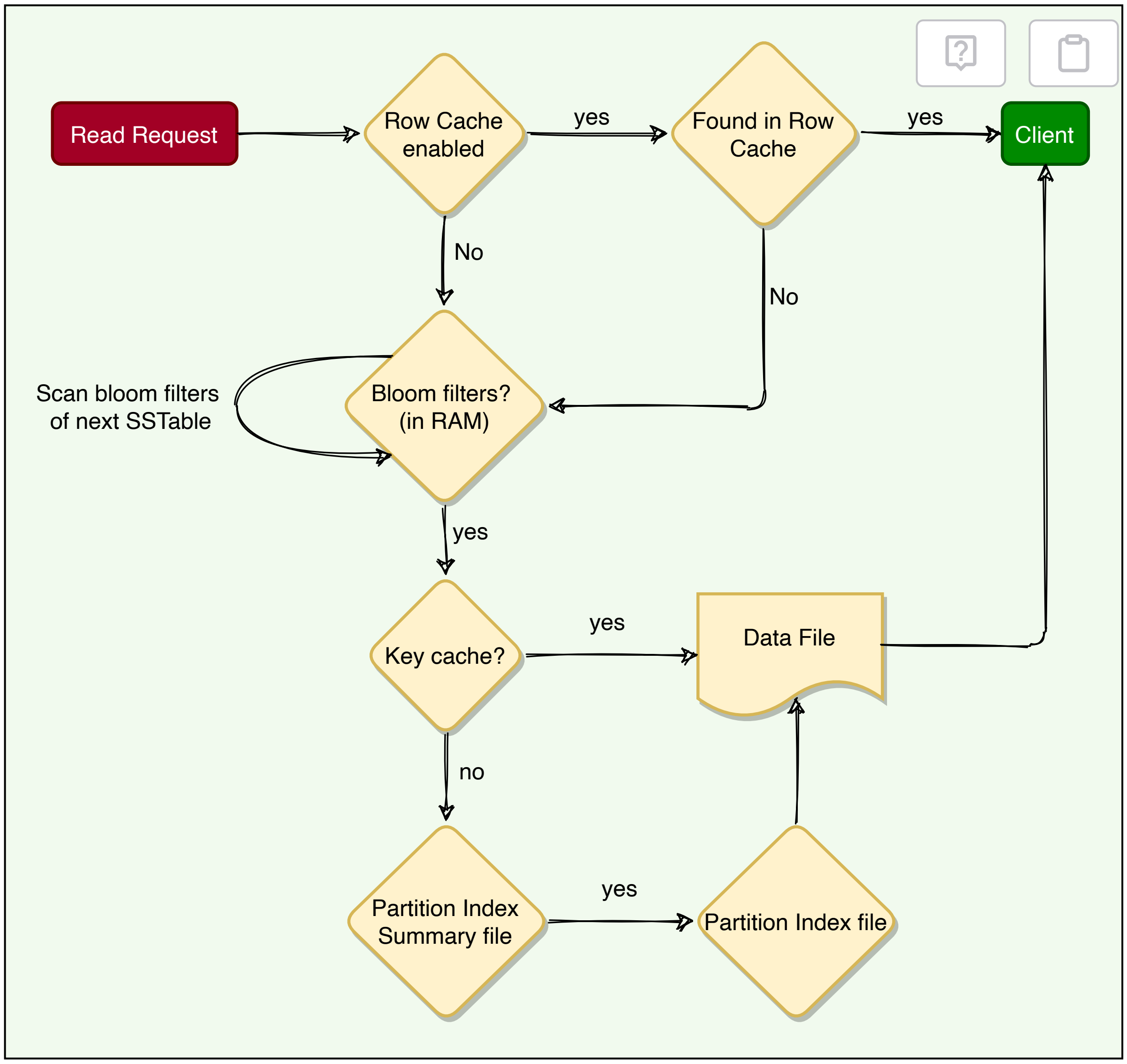
As the Key Cache stores a map of recently read partition keys to their SSTable offsets, it is the fastest way to find the required row in the SSTable.



Reading SSTable through key cache

If data is not present in MemTable, we have to look it up in SSTables or other data structures like partition index, etc. Here is the summary of Cassandra's read operation:

1. First, Cassandra checks if the row is present in the Row Cache. If present, the data is returned, and the request ends.
2. If the row is not present in the Row Cache, bloom filters are checked. If a bloom filter indicates that the data is present in an SSTable, Cassandra looks for the required partition in that SSTable.
3. The key cache is checked for the partition key presence. A cache hit provides an offset for the partition in SSTable. This offset is then used to retrieve the partition, and the request completes.
4. Cassandra continues to seek the partition in the partition summary and partition index. These structures also provide the partition offset in an SSTable which is then used to retrieve the partition and return. The caches are updated if present with the latest data read.



Cassandra's read operation workflow

Back

Anatomy of Cassandra's Write Operat...

Next

Compaction

Completed

Report an Issue

Compaction

Let's explore how Cassandra handles compaction.

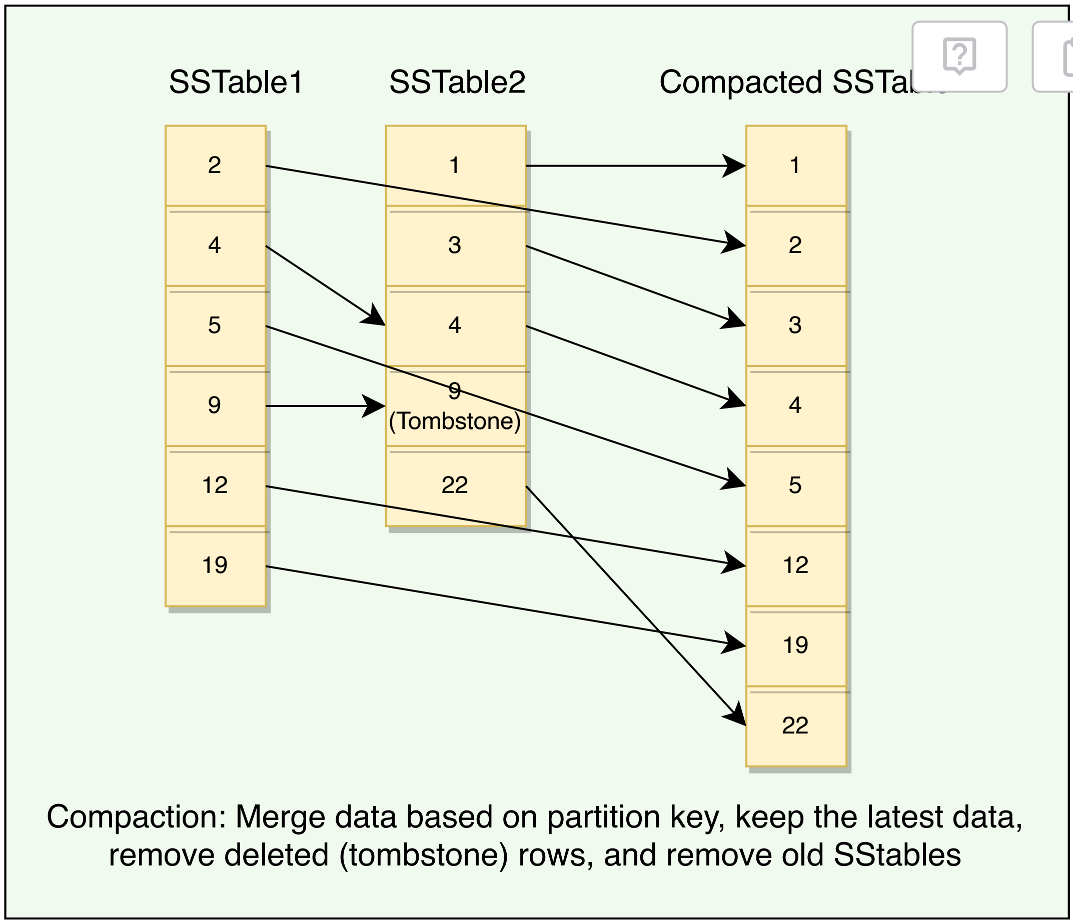
We'll cover the following

- How does compaction work in Cassandra?
- Compaction strategies
- Sequential writes

How does compaction work in Cassandra?#

As we have already discussed, SSTables are immutable, which helps Cassandra achieve such high write speeds. Flushing of MemTable to SStable is a continuous process. This means we can have a large number of SStables lying on the disk. While reading, it is tedious to scan all these SStables. So, to improve the read performance, we need compaction. Compaction in Cassandra refers to the operation of merging multiple related SSTables into a single new one. During compaction, the data in SSTables is merged: the keys are merged, columns are combined, obsolete values are discarded, and a new index is created.

On compaction, the merged data is sorted, a new index is created over the sorted data, and this freshly merged, sorted, and indexed data is written to a single new SSTable.


SSTable1
SSTable2
Compacted SSTa
2
4
5
9
12
19
1
3
4
9
(Tombstone)
22
1
2
3
4
5
12
19
22
Compaction: Merge data based on partition key, keep the latest data, remove deleted (tombstone) rows, and remove old SStables

Compacting two SSTables into one

- Compaction will reduce the number of SSTables to consult and therefore improve read performance.
- Compaction will also reclaim space taken by obsolete data in SSTable.

Compaction strategies#

SizeTiered Compaction Strategy: This compaction strategy is suitable for insert-heavy and general workloads. This is the default compaction strategy and is triggered when multiple SSTables of a similar size are present.

Leveled Compaction Strategy: This strategy is used to optimize read performance. This strategy groups SSTables into levels, each of which has a fixed size limit which is ten times larger than the previous level.

Time Window Compaction Strategy: The Time Window Co ti Strategy is designed to work on time series data. It compacts SSTables within a configured time window. This strategy is ideal for time series data which is immutable after a fixed time interval.

Sequential writes#

Sequential writes are the primary reason that writes perform so well in Cassandra. No reads or seeks of any kind are required for writing a value to Cassandra because all writes are ‘append’ operations. This makes the speed of the disk one key limitation on performance. Compaction is intended to amortize the reorganization of data, but it uses sequential I/O to do so, which makes it efficient. If Cassandra naively inserted values where they ultimately belonged, writing clients would pay for seeks upfront.

Back
Anatomy of Cassandra's Read Operati...

Next
Tombstones

✓ Completed

Report an Issue

Tombstones

Let's explore how Tombstones work in Cassandra.

We'll cover the following

- What are Tombstones?
- Common problems associated with Tombstones

What are Tombstones?#

An interesting case with Cassandra can be when we delete some data for a node that is down or unreachable, that node could miss a delete. When that node comes back online later and a repair occurs, the node could "resurrect" the data that had been previously deleted by re-sharing it with other nodes. To prevent deleted data from being reintroduced, Cassandra uses a concept called a tombstone. A tombstone is similar to the idea of a "**soft delete**" from the relational database world. When we delete data, Cassandra does not delete it right away, instead associates a tombstone with it, with a time to expiry. In other words, a tombstone is a marker that is kept to indicate data that has been deleted. When we execute a delete operation, the data is not immediately deleted. Instead, it's treated as an update operation that places a tombstone on the value.

Each tombstone has an expiry time associated with it, representing the amount of time that nodes will wait before removing the data permanently. By default, each tombstone has an expiry of ten days. The purpose of this delay is to give a node that is unavailable time to recover. If a node is down longer than this value, then it should be treated as failed and

Tombstones are removed as part of compaction. During compaction, any row with an expired tombstone will not be propagated further.

Common problems associated with Tombstones#

Tombstones make Cassandra writes actions efficient because the data is not removed right away when deleted. Instead, it is removed later during compaction. Having said that, Tombstones cause the following problems:

- As a tombstone itself is a record, **it takes storage space**. Hence, it should be kept in mind that upon deletion, the application will end up increasing the data size instead of shrinking it. Furthermore, if there are a lot of tombstones, the available storage for the application could be substantially reduced.
- When a table accumulates many tombstones, read queries on that table could become slow and can cause serious performance problems like timeouts. This is because we have to read much more data until the actual compaction happens and removes the tombstones.

Back
Compaction

Next
Summary: Cassandra

Completed

Report an Issue

Summary: Cassandra

Here is a quick summary of Cassandra for you!

We'll cover the following

- Summary
- System design patterns
- Cassandra characteristics
- References and further reading

Summary#

1. Cassandra is a **distributed**, **decentralized**, **scalable**, and **highly available** NoSQL database.
2. Cassandra was designed with the understanding that software/hardware **failures can and do occur**.
3. Cassandra is a **peer-to-peer** distributed system, i.e., it does not have any leader or follower nodes. All nodes are equal, and there is no single point of failure.
4. Data, in Cassandra, is automatically distributed across nodes.
5. Data is replicated across the nodes for fault tolerance and redundancy.
6. Cassandra uses the **Consistent Hashing** algorithm to distribute the data among nodes in the cluster. Cassandra cluster has a ring-type architecture, where its nodes are logically distributed like a ring.
7. Cassandra utilizes the data model of Google's Bigtable, i.e., **SSTables** and **MemTables**.

MemTables.

8. Cassandra utilizes distributed features of Amazon’s Dynamo, i.e., consistent hashing, partitioning, and replication.
9. Cassandra offers **Tunable consistency** for both read and write operations to adjust the tradeoff between availability and consistency of data.
10. Cassandra uses the **gossip protocol** for inter-node communication.

System design patterns#

Here is a summary of system design patterns used in Cassandra:

- **Consistent Hashing:** Cassandra uses Consistent Hashing to distribute its data across nodes.

- **Quorum:** To ensure data consistency, each Cassandra write operation can be configured to be successful only if the data has been written to at least a quorum of replica nodes.

- **Write-Ahead Log:** To ensure durability, whenever a node receives a write request, it immediately writes the data to a commit log which is a write-ahead log.

- **Segmented Log:** Cassandra uses the segmented log strategy to split its commit log into multiple smaller files instead of a single large file for easier operations. As we know, when a node receives a write operation, it immediately writes the data to a commit log. As the commit log grows and reaches its threshold in size, a new commit log is created. Hence, over time several commit logs could be present, each of which is called a segment. Commit log segments reduce the number of seeks needed to write to disk. Commit log segments are truncated when Cassandra has flushed corresponding data to SSTables. Commit log segments can be archived, deleted, or recycled once all its data has been flushed to

SSTables.

- **Gossip protocol:** Cassandra uses gossip protocol that allows each node to keep track of state information about the other nodes in the cluster.
- **Generation number:** In Cassandra, each node keeps a generation number which is incremented whenever a node restarts. This generation number is included in gossip messages exchanged between nodes and is used to distinguish the node's current state from its state before a restart.
- **Phi Accrual Failure Detector:** Cassandra uses an adaptive failure detection mechanism as described by the Phi Accrual Failure Detector algorithm. This algorithm, instead of providing a binary output telling if the system is up or down, uses historical heartbeat information to output the suspicion level about a node. A higher suspicion level means there are high chances that the node is down.
- **Bloom filters:** In Cassandra, each SStable has a Bloom filter associated with it, which tells if a particular key is present in it or not.
- **Hinted Handoff:** Cassandra nodes use Hinted Handoff to remember the write operation for failing nodes.
- **Read Repair:** Cassandra uses 'Read Repair' to push the latest version of the data to nodes with the older versions.

Cassandra characteristics#

Here are a few reasons behind Cassandra's performance and popularity:

- **Distributed** means it can run on a large number of machines.
- **Decentralized** means there's no leader-follower paradigm. All nodes are identical and can perform all functions of Cassandra.

- **Scalable** means that Cassandra can be easily scaled horizontally by adding more nodes to the cluster without any performa[illegible] manual intervention or rebalancing is required. Cassandra achieves linear scalability and proven fault-tolerance on commodity hardware.
- **Highly Available** means Cassandra is fault-tolerant, and the data remains available even if one or several nodes or data centers go down.
- **Fault-Tolerant and reliable,** as data is replicated to multiple nodes, fault-tolerance is pretty high.
- **Tunable consistency** means that it is possible to adjust the tradeoff between availability and consistency of data on Cassandra nodes, typically by configuring replication factor and consistency level settings.
- **Durable** means Cassandra stores data permanently.
- **Eventually Consistent** as Cassandra favors high availability at the cost of strong consistency.
- **Geographic distribution** means Cassandra supports geographical distribution and efficient data replication across multiple clouds and data centers.

References and further reading#

- Bigtable
- Dynamo
- Datastax docs
- Tombstones common problems
- The Phi Accrual Failure Detector
- Cassandra introduction video

Back
Tombstones

Next
Quiz: Cassandra

Report an Issue