Introducing Cassandra

Distributed and Decentralized

- Cassandra is distributed, which means that it is capable of running on multiple machines while appearing to users as a unified whole.
- The fact that Cassandra is decentralized means that there is no single point of failure. All of the nodes in a Cassandra cluster function exactly the same. This is sometimes referred to as “server symmetry.” Cassandra features a peer-to-peer architecture and uses a gossip protocol to maintain and keep in sync a list of nodes that are alive or dead.

Elastic Scalability

Elastic scalability refers to a special property of horizontal scalability. It means that your cluster can seamlessly scale up and scale back down. To do this, the cluster must be able to accept new nodes that can begin participating by getting a copy of some or all of the data and start serving new user requests without major disruption or reconfiguration of the entire cluster.

High Availability and Fault Tolerance

Cassandra is highly available. You can replace failed nodes in the cluster with no downtime, and you can replicate data to multiple data centers to offer improved local performance and prevent downtime.

Tuneable Consistency

Out of the box, Cassandra trades some consistency in order to achieve total availability. But Cassandra is more accurately termed “tuneably consistent,” which means it allows you to easily decide the level of consistency you require, in balance with the level of availability.
Eventual consistency is one of several consistency models available:

- Strict consistency
- Causal consistency
- Weak(eventual) consistency

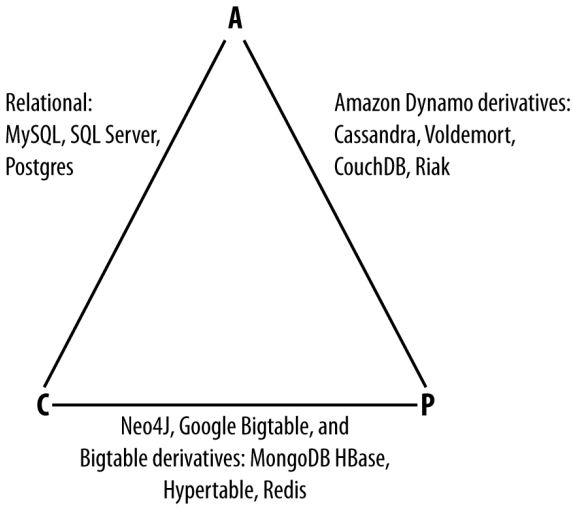
Dynamo and Cassandra choose to be always writable, opting to defer the complexity of reconciliation to read operations, and realize tremendous performance gains. The alternative is to reject updates amidst network and server failures.

Brewer’s CAP Theorem

The theorem states that within a large-scale distributed data system, there are three requirements that have a relationship of sliding dependency:

- Consistency All database clients will read the same value for the same query, even given concurrent updates.
- Availability All database clients will always be able to read and write data.
- Partition tolerance The database can be split into multiple machines; it can continue functioning in the face of network segmentation breaks.

Brewer’s theorem is that in any given system, you can strongly support only two of the three.



- CA To primarily support consistency and availability means that you’re likely using two-phase commit for distributed transactions. It means that the system will block when a network partition occurs, so it may be that your system is limited to a single data center cluster in an attempt to mitigate this. If your application needs only this level of scale, this is easy to manage and allows you to rely on familiar, simple structures.
- CP To primarily support consistency and partition tolerance, you may try to advance your architecture by setting up data shards in order to scale. Your data will be consistent, but you still run the risk of some data becoming unavailable if nodes fail.
- AP To primarily support availability and partition tolerance, your system may return inaccurate data, but the system will always be available, even in the face of network partitioning. DNS is perhaps the most popular example of a system that is massively scalable, highly available, and partition tolerant.

Row-Oriented

Cassandra’s data model can be described as a partitioned row store, in which data is stored in sparse multidimensional hash tables. “Sparse” means that for any given row you can have one or more columns, but each row doesn’t need to have all the same columns as other rows like it. “Partitioned” means that each row has a unique partition key used to distribute the rows across multiple data stores. Somewhat confusingly, this type of data model is also frequently referred to as a wide column store.
Cassandra has frequently been referred to as a column-oriented or columnar database, but this is not technically correct.

High Performance

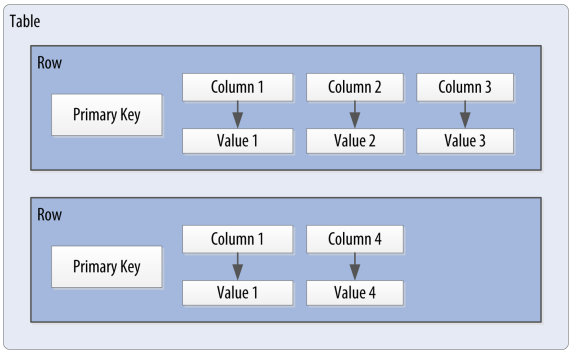
Cassandra was designed specifically from the ground up to take full advantage of multiprocessor/multicore machines, and to run across many dozens of these machines housed in multiple data centers.

Is Cassandra a Good Fit for My Project?

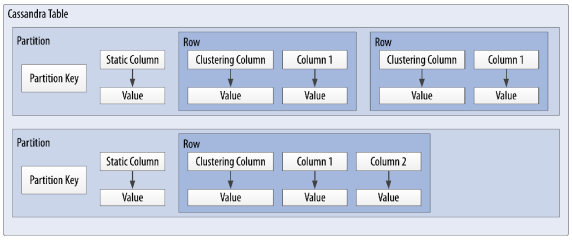
Good for

- Large deployments
- Lots of Writes, Statistics, and Analysis
- Geographical Distribution
- Hybrid Cloud and Multi-cloud Deployment

Cassandra’s Data Model



Cassandra uses a special type of primary key called a composite key (or compound key) to represent groups of related rows, also called partitions. The composite key consists of a partition key, plus an optional set of clustering columns.
The partition key is used to determine the nodes on which rows are stored and can itself consist of multiple columns.
The clustering columns are used to control how data is sorted for storage within a partition.
Cassandra also supports an additional construct called a static column, which is for storing data that is not part of the primary key but is shared by every row in a partition.



We have the basic Cassandra data structures:

- The column, which is a name/value pair
- The row, which is a container for columns referenced by a primary key
- The partition, which is a group of related rows that are stored together on the same nodes

- The table, which is a container for rows organized by partitions
- The keyspace, which is a container for tables
- The cluster, which is a container for keyspaces that spans one or more nodes

Keyspaces A keyspace is the outermost container for data in Cassandra, corresponding closely to a database in the relational model.
Tables A table is a container for an ordered collection of rows. Rows are organized in partitions and assigned to nodes in a Cassandra cluster according to the column(s) designated as the partition key. The ordering of data within a partition is determined by the clustering columns.
Because Cassandra uses an append model, there is no fundamental difference between the insert and update operations. If you insert a row that has the same primary key as an existing row, the row is replaced. If you update a row and the primary key does not exist, Cassandra creates it.For this reason, it is often said that Cassandra supports **upsert**, meaning that inserts and updates are treated the same.

Timestamps

Each time you write data into Cassandra, a timestamp, in microseconds, is generated for each column value that is inserted or updated. Internally, Cassandra uses these timestamps for resolving any conflicting changes that are made to the same value, in what is frequently referred to as a last write wins approach. But, you're not allowed to ask for the timestamp on primary key columns:

```
cqlsh:my_keyspace> SELECT WRITETIME(first_name) FROM user;
InvalidRequest: code=2200 [Invalid query] message="Cannot use
  selection function writeTime on PRIMARY KEY part first_name"
```

Cassandra also allows you to specify a timestamp you want to use when performing writes (note that the timestamp must be later than the one from your SELECT command, or the UPDATE will be ignored):

```
cqlsh:my_keyspace> UPDATE user USING TIMESTAMP 1567886623298243
  SET middle_initial = 'Q' WHERE first_name = 'Mary' AND last_name = 'Rodriguez';
```

Time to live (TTL)

The time to live (or TTL) is a value that Cassandra stores for each column value to indicate how long to keep the value.The TTL value defaults to null, meaning that data that is written will not expire.

```
cqlsh:my_keyspace> UPDATE user USING TTL 3600 SET middle_initial =
  'Z' WHERE first_name = 'Mary' AND last_name = 'Rodriguez';
```

Remember that TTL is stored on a per-column level for non-primary key columns.

After you create a table, there is no way to modify the primary key, because this controls how data is distributed within the cluster, and even more importantly, how it is stored on disk.

Data Modeling

- **No joins** You cannot perform joins in Cassandra. If you have designed a data model and find that you need something like a join, you'll have to either do the work on the client side, or create a denormalized second table that represents the join results for you.
- **No referential integrity** Although Cassandra supports features such as lightweight transactions and batches, Cassandra itself has no concept of referential integrity across tables.
- **Denormalization** Cassandra performs best when the data model is denormalized.
- **Query-first design** Instead of modeling the data first and then writing queries, with Cassandra you model the queries and let the data be organized around them. Think of the most common query paths your application will use, and then create the tables that you need to support them.
- **Designing for optimal storage** Because Cassandra tables are each stored in separate files on disk, it's important to keep related columns defined together in the same table.A key goal as you begin creating data models in Cassandra is to minimize the number of partitions that must be searched in order to satisfy a given query. Because the partition is a unit of storage that does not get divided across nodes, a query that searches a single partition will typically yield the best performance.
- **Sorting is a design decision** The sort order available on queries is fixed, and is determined entirely by the selection of clustering columns you supply in the CREATE TABLE command. The CQL SELECT statement does support ORDER BY semantics.

The Importance of Primary Keys The design of the primary key is extremely important, as it will determine how much data will be stored in each partition and how that data is organized on disk, which in turn will affect how quickly Cassandra processes read queries.
Searching Over a Range Use clustering columns to store attributes that you need to access in a range query. Remember that the order of the clustering columns is important.

The Wide Partition Pattern

The essence of the pattern is to group multiple related rows in a partition in order to support fast access to multiple rows within the partition in a single query.

Evaluating and Refining

Calculating Partition Size

Partition size is measured by the number of cells (values) that are stored in the partition. Cassandra's hard limit is two billion cells per partition. The recommended size of a partition is not more than 100,000 cells.In order to calculate the size of your partitions, you use the following formula:
Nv=Nr(Nc−Npk−Ns)+Ns

The number of values (or cells) in the partition (Nv) is equal to the number of static columns (Ns) plus the product of the number of rows (Nr) and the number of values per row. The number of values per row is defined as the number of columns (Nc) minus the number of primary key columns (Npk) and static columns (Ns).

Calculating Size on Disk

$$S_t = \sum_i sizeOf(c_{k_i}) + \sum_j sizeOf(c_{s_j}) + N_r \times \left(\sum_k sizeOf(c_{r_k}) + \sum_l sizeOf(c_{c_l}) \right) +$$

$$N_v \times sizeOf(t_{avg})$$

- In this formula, ck refers to partition key columns, cs to static columns, cr to regular columns, and cc to clustering columns.
- The term avg refers to the average number of bytes of metadata stored per cell, such as timestamps. It is typical to use an estimate of 8 bytes for this value.
- You recognize the number of rows Nr and number of values Nv from previous calculations.
- The sizeOf() function refers to the size, in bytes, of the CQL data type of each referenced column.

Keep in mind also that this estimate only counts a single replica of your data. You will need to multiply the value obtained here by the number of partitions and the number of replicas specified by the keyspace's replication strategy in order to determine the total required capacity for each table.

Breaking Up Large Partitions

It is quite possible to design wide partition–style tables that approach Cassandra's built-in limits. Performing sizing analysis on tables may reveal partitions that are potentially too large, either in number of values, size on disk, or both.
The technique for splitting a large partition is straightforward: add an additional column to the partition key. In most cases, moving one of the existing columns into the partition key will be sufficient. Another option is to introduce an additional column to the table to act as a sharding key, but this requires additional application logic.

The Cassandra Architecture

Gossip and Failure Detection

To support decentralization and partition tolerance, Cassandra uses a gossip protocol that allows each node to keep track of state information about the other nodes in the cluster. The gossiper runs every second on a timer.
Cassandra has robust support for failure detection, as specified by a popular algorithm for distributed computing called Phi Accrual Failure Detector.
Traditional failure detectors, which are implemented by simple "heartbeats" and decide whether a node is dead or not dead based on whether a heartbeat is received or not. But accrual failure detection decides that this approach is naive, and finds a place in between the extremes of dead and alive—a suspicion level.
Therefore, the failure monitoring system outputs a continuous level of "suspicion" regarding how confident it is that a node has failed. This is desirable because it can take into account fluctuations in the network environment. For example, just because one connection gets caught up doesn't necessarily mean that the whole node is dead.

Snitches

The job of a snitch is to provide information about your network topology so that Cassandra can efficiently route requests. The snitch will figure out where nodes are in relation to other nodes. The snitch will determine relative host proximity for each node in a cluster, which is used to determine which nodes to read and write from.
As an example, let's examine how the snitch participates in a read operation. When Cassandra performs a read, it must contact a number of replicas determined by the consistency level. In order to support the maximum speed for reads, Cassandra selects a single replica to query for the full object, and asks additional replicas for hash values in order to ensure the latest version of the requested data is returned. The snitch helps to help identify the replica that will return the fastest, and this is the replica that is queried for the full data.

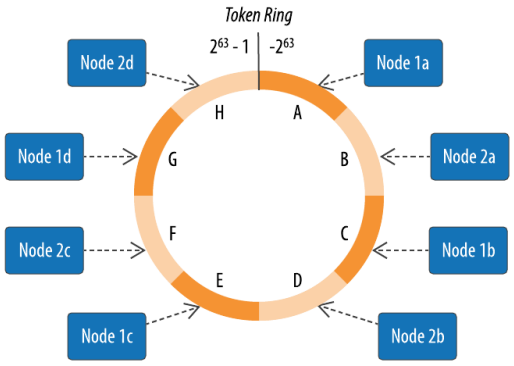
Dynamic Snitch

Cassandra provides a feature called dynamic snitching that helps optimize the routing of reads and writes over time. The dynamic snitch gets its basic understanding of the topology from the selected snitch. It then monitors the performance of requests to the other nodes, even keeping track of things like which nodes are performing compaction. The performance data is used to select the best replica for each query. This enables Cassandra to avoid routing requests to replicas that are busy or performing poorly.
The dynamic snitching implementation uses a modified version of the Phi failure detection mechanism used by gossip. The badness threshold is a configurable parameter that determines how much worse a preferred node must perform than the best-performing node in order to lose its preferential status. The scores of each node are reset periodically in order to allow a poorly performing node to demonstrate that it has recovered and reclaim its preferred status.

Rings and Tokens

Cassandra represents the data managed by a cluster as a ring. Each node in the ring is assigned one or more ranges of data described by a token, which determines its position in the ring.
A node claims ownership of the range of values less than or equal to each token and greater than the last token of the previous node, known as a token range. The node with the lowest token owns the range less than or equal to its token and the range greater than the highest token, which is also known as the wrapping range.

Data is assigned to nodes by using a hash function to calculate a token for the partition key. This partition key token is compared to the token values for the various nodes to identify the range, and therefore the node, that owns the data.

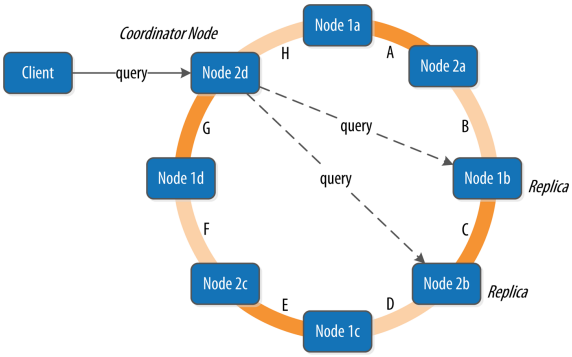


Virtual Nodes

Instead of assigning a single token to a node, the token range is broken up into multiple smaller ranges. Each physical node is then assigned multiple tokens. Historically, each node has been assigned 256 of these tokens, meaning that it represents 256 virtual nodes.
Vnodes make it easier to maintain a cluster containing heterogeneous machines. For nodes in your cluster that have more computing resources available to them, you can increase the number of vnodes by setting the num_tokens property in the cassandra.yaml file.

Partitioners

A partitioner determines how data is distributed across the nodes in the cluster. A partitioner, then, is a hash function for computing the token of a partition key. The role of the partitioner is to compute the token based on the partition key columns.

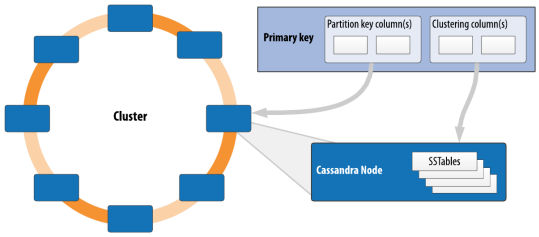


Hinted Handoff

Consider the following scenario: a write request is sent to Cassandra, but a replica node where the write properly belongs is not available. In order to ensure general availability of the ring in such a situation, Cassandra implements a feature called hinted handoff. If the replica node where the write belongs has failed, the coordinator will create a hint, which is a small reminder that says, “I have the write information that is intended for node B. I’m going to hang on to this write, and I’ll notice when node B comes back online; when it does, I’ll send it the write request.” That is, once it detects via gossip that node B is back online, node A will “hand off” to node B the “hint” regarding the write. Cassandra holds a separate hint for each partition that is to be written.
In general, hints do not count as writes for the purposes of consistency level. The exception is the consistency level ANY. That is, even if only a hint was able to be recorded, the write still counts as successful.
There is a practical problem with hinted handoffs (and guaranteed delivery approaches, for that matter): if a node is offline for some time, the hints can build up considerably on other nodes. Then, when the other nodes notice that the failed node has come back online, they tend to flood that node with requests. To address this problem, Cassandra limits the storage of hints to a configurable time window. It is also possible to disable hinted handoff entirely.

Anti-Entropy, Repair, and Merkle Trees

Replica synchronization is supported via two different modes known as read repair and anti-entropy repair. Read repair refers to the synchronization of replicas as data is read. Cassandra reads data from multiple replicas in order to achieve the requested consistency level, and detects if any replicas have out-of-date values. If an insufficient number of nodes



Replication Strategies

A node serves as a replica for different ranges of data. Cassandra replicates data across nodes in a manner transparent to the user, and the replication factor is the number of nodes in your cluster that will receive copies (replicas) of the same data.
The first replica will always be the node that claims the range in which the token falls, but the remainder of the replicas are placed according to the replication strategy (sometimes also referred to as the replica placement strategy).

Consistency Levels

You specify a consistency level on each read or write query that indicates how much consistency you require. A higher consistency level means that more nodes need to respond to a read or write query, giving you more assurance that the values present on each replica are the same. The replication factor is set per keyspace. The consistency level is specified per query, by the client.
Q=floor(RF/2+1)

Queries and Coordinator Nodes

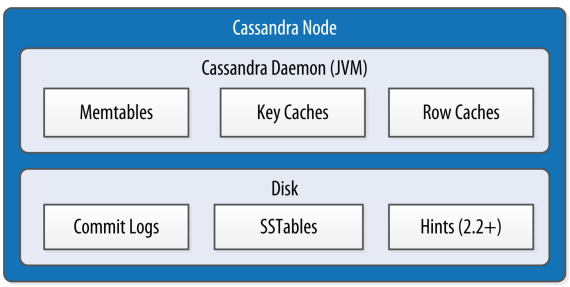
A client may connect to any node in the cluster to initiate a read or write query. This node is known as the coordinator node. The coordinator identifies which nodes are replicas for the data that is being written or read and forwards the queries to them.

have the latest value, a read repair is performed immediately to update the out-of-date replicas. Otherwise, the repairs can be performed in the background after the read returns.
Anti-entropy repair (sometimes called manual repair) is a manually initiated operation performed on nodes as part of a regular maintenance process.
During a validation compaction, the server initiates a TreeRequest/TreeReponse conversation to exchange Merkle trees with neighboring replicas. The Merkle tree is a hash representing the data in that table. If the trees from the different nodes don’t match, they have to be reconciled (or “repaired”) to determine the latest data values they should all be set to.

Lightweight Transactions and Paxos

Cassandra supports a lightweight transaction (LWT) mechanism that provides linearizable consistency.Cassandra’s LWT implementation is based on Paxos. Paxos is a consensus algorithm that allows distributed peer nodes to agree on a proposal, without requiring a leader to coordinate a transaction.
The basic Paxos algorithm consists of two stages: prepare/promise and propose/accept. To modify data, a coordinator node can propose a new value to the replica nodes, taking on the role of leader. Other nodes may act as leaders simultaneously for other modifications. Each replica node checks the proposal, and if the proposal is the latest it has seen, it promises to not accept proposals associated with any prior proposals. Each replica node also returns the last proposal it received that is still in progress. If the proposal is approved by a majority of replicas, the leader commits the proposal, but with the caveat that it must first commit any in-progress proposals that preceded its own proposal.
Thus, a successful transaction requires four round-trips between the coordinator node and replicas. This is more expensive than a regular write, which is why you should think carefully about your use case before using LWTs.
Cassandra’s lightweight transactions are limited to a single partition.

Memtables, SSTables, and Commit Logs



When a node receives a write operation, it immediately writes the data to a commit log. The commit log is a crash-recovery mechanism that supports Cassandra's durability goals. A write will not count as successful on the node until it's written to the commit log, to ensure 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.
After it's written to the commit log, the value is written to a memory-resident data structure called the memtable. Each memtable contains data for a specific table.
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 an SSTable. A new memtable is then created. This flushing is a non-blocking operation; multiple memtables may exist for a single table, one current and the rest waiting to be flushed.
Once a memtable is flushed to disk as an SSTable, it is immutable and cannot be changed by the application.
Each commit log maintains an internal bit flag to indicate whether it needs flushing. When a write operation is first received, it is written to the commit log and its bit flag is set to 1. There is only one bit flag per table, because only one commit log is ever being written to across the entire server. Once the memtable has been properly flushed to disk, the corresponding commit log's bit flag is set to 0, indicating that the commit log no longer has to maintain that data for durability purposes.
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 disk.

Durable Writes

```
cqlsh> DESCRIBE KEYSPACE my_keyspace ;

CREATE KEYSPACE my_keyspace WITH replication =
  {'class': 'SimpleStrategy',
   'replication_factor': '1'}  AND durable_writes = true;
```

The durable_writes property controls whether Cassandra will use the commit log for writes to the tables in the keyspace. This value defaults to true, meaning that the commit log will be updated on modifications. Setting the value to false increases the speed of writes, but also risks losing data if the node goes down before the data is flushed from memtables into SSTables.

Bloom Filters

Bloom filters are used to boost the performance of reads. Bloom filters are very fast, nondeterministic 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 not a false-negative.
Cassandra maintains a Bloom filter for each SSTable. When a query is performed, the Bloom filter is checked first before accessing disk. Because false-negatives are not possible, if the filter indicates that the element does not exist in the set, it certainly doesn't; but if the filter thinks that the element is in the set, the disk is accessed to make sure.

Caching

- **The key cache** stores a map of partition keys to row index entries, facilitating faster read access into SSTables stored on disk. The key cache is stored on the JVM heap.
- **The row cache** caches entire rows and can greatly speed up read access for frequently accessed rows, at the cost of more memory usage. The row cache is stored in off-heap memory.
- **The chunk cache** was added in the 3.6 release to store uncompressed chunks of data read from SSTable files that are accessed frequently. The chunk cache is stored in off-heap memory.
- **The counter cache** was added in the 2.1 release to improve counter performance by reducing lock contention for the most frequently accessed counters.

By default, key and counter caching are enabled, while row caching is disabled, as it requires more memory.

Compaction

Periodic compaction of these SSTables is important in order to support fast read performance and clean out stale data values. A compaction operation in Cassandra is performed in order to merge SSTables. 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.

Deletion and Tombstones

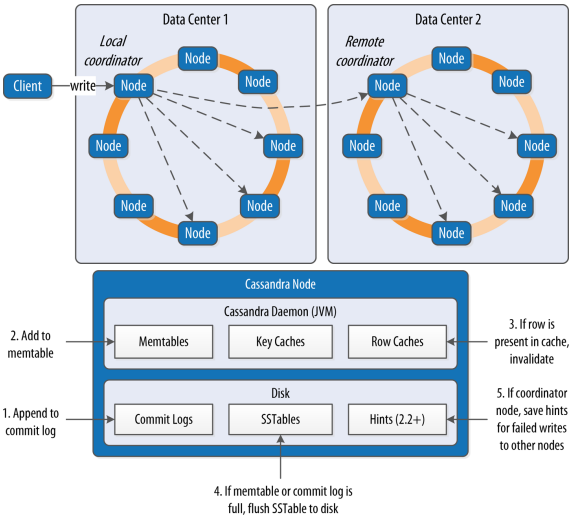
To prevent deleted data from being reintroduced, Cassandra uses a concept called a tombstone. A tombstone is a marker that is kept to indicate data that has been deleted. When you execute a delete operation, the data is not immediately deleted. Instead, it's treated as an update operation that places a tombstone on the value.
Tombstones are not kept forever; instead, they are removed as part of compaction. There is a setting per table called gc_grace_seconds (Garbage Collection Grace Seconds), which represents the amount of time that nodes will wait to garbage collect (or compact) tombstones. By default, it's set to 864,000 seconds, the equivalent of 10 days.

Writing and Reading Data

Write Consistency Levels

Consistency level	Implication
`ANY`	Ensure that the value is written to a minimum of one replica node before returning to the client, allowing hints to count as a write.
`ONE`, `TWO`, `THREE`	Ensure that the value is written to the commit log and memtable of at least one, two, or three nodes before returning to the client.
`LOCAL_ONE`	Similar to `ONE`, with the additional requirement that the responding node is in the local data center.
`QUORUM`	Ensure that the write was received by at least a majority of replicas ((`replication factor` / 2) + 1).
`LOCAL_QUORUM`	Similar to `QUORUM`, where the responding nodes are in the local data center.
`EACH_QUORUM`	Ensure that a `QUORUM` of nodes respond in each data center.
`ALL`	Ensure that the number of nodes specified by `replication factor` received the write before returning to the client. If even one replica is unresponsive to the write operation, fail the operation.

Write Path



Lightweight Transactions

Cassandra does provide two mechanisms that offer some transactional behavior: lightweight transactions and batches.

- On an INSERT, adding the IF NOT EXISTS clause will ensure that you do not overwrite an existing row with the same primary key. Alternatively, the IF EXISTS clause will only update the row with the provided primary key if it is already present in the database.
- On an UPDATE, adding an IF <conditions> clause will perform a check of one or more provided conditions, where multiple conditions are separated by an AND.

Conditional write statements use a serial consistency level in addition to the regular consistency level. The serial consistency level determines the number of nodes that must reply in the Paxos phase of the write, when the participating nodes are negotiating about the proposed write.

Consistency level	Implication
SERIAL	This is the default serial consistency level, indicating that a quorum of nodes must respond.
LOCAL_SERIAL	Similar to SERIAL, but indicates that the transaction will only involve nodes in the local data center.

Batches

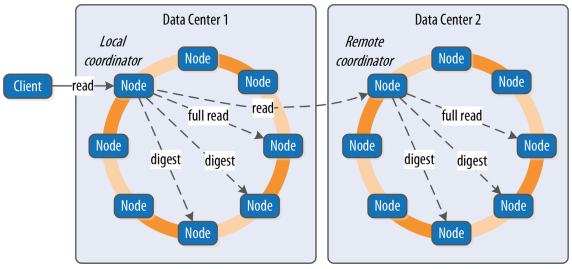
While lightweight transactions are limited to a single partition, Cassandra provides a batch mechanism that allows you to group multiple modifications into a single statement, whether they address the same partition or different partitions.

- Only modification statements (INSERT, UPDATE, or DELETE) may be included in a batch.
- Batches may be logged or unlogged, where logged batches have more safeguards.
- Batches are not a transaction mechanism, but you can include lightweight transaction statements in a batch. Multiple lightweight transactions in a batch must apply to the same partition.
- Counter modifications are only allowed within a special form of batch known as a counter batch. A counter batch can only contain counter modifications.

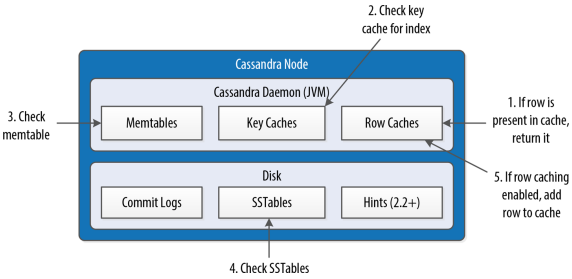
Reading

In Cassandra, reads are generally slower than writes due to file I/O from reading SSTables. To fulfill read operations, Cassandra typically has to perform seeks, but you may be able to keep more data in memory by adding nodes, using compute instances with more memory, and using Cassandra's caches. Cassandra also has to wait for responses synchronously on reads (based on consistency level and replication factor), and then perform read repairs as necessary.
The consistency levels for read operations are similar to the write consistency levels, but the way they are handled behind the scenes is slightly different. A higher consistency level means that more nodes need to respond to the query, giving you more assurance that the values present on each replica are the same. If two nodes respond with different timestamps, the newest value wins, and that's what will be returned to the client. Cassandra will then perform what's called a read repair: it takes notice of the fact that one or more replicas responded to a query with an outdated value, and updates those replicas with the most current value so that they are all consistent.

The Cassandra Read Path



If the coordinator is not itself a replica, the coordinator then sends a read request to the fastest replica, as determined by the dynamic snitch. The coordinator node also sends a digest request to the other replicas. A digest request is similar to a standard read request, except the replicas return a digest, or hash, of the requested data.The coordinator calculates the digest hash for data returned from the fastest replica and compares it to the digests returned from the other replicas. If the digests are consistent, and the desired consistency level has been met, then the data from the fastest replica can be returned. If the digests are not consistent, then the coordinator must perform a read repair, as discussed in the following section.



There is only a single memtable for a given table, so that part of the search is straightforward. However, there are potentially many physical SSTables for a single Cassandra table, each of which may contain a portion of the requested data.
The first step in searching SSTables on disk is to use a Bloom filter to determine whether the requested partition does not exist in a given SSTable, which would make it unnecessary to search that SSTable.If the SSTable passes the Bloom filter, Cassandra checks the key cache to see if it contains the offset of the partition key in the SSTable. The key cache is implemented as a map structure in which the keys are a combination of the SSTable file descriptor and partition key, and the values are offset locations into SSTable files. The key cache helps to eliminate seeks within SSTable files for frequently accessed data, because the data can be read directly.If the offset is not obtained from the key cache, Cassandra uses a two-level index stored on disk in order to locate the offset. The first-level index is the partition summary, which is used to obtain an offset for searching for the partition key within the second-level index, the partition index. The partition index is where the offset into the SSTable for the partition key is stored.If the offset for the partition key is found, Cassandra accesses the SSTable at the specified offset and starts reading data.
Once data has been obtained from all of the SSTables, Cassandra merges the SSTable data and memtable data by selecting the value with the latest timestamp for each requested column.Finally, the merged data can be added to the row cache (if enabled) and returned to the client or coordinator node.

Read Repair

The read repair may be performed either before or after the return to the client. If you are using one of the two stronger consistency levels (QUORUM or ALL), then the read repair happens before data is returned to the client. If the client specifies a weak consistency level (such as ONE), then the read repair is optionally performed in the background after returning to the client.

Range Queries, Ordering and Filtering

The syntax of the WHERE clause involves two rules. First, all elements of the partition key must be identified. Second, a given clustering key may only be restricted if all previous clustering keys are restricted by equality.
https://www.datastax.com/blog/deep-look-cql-where-clause

Deleting

In Cassandra, a delete does not actually remove the data immediately. There's a simple reason for this: Cassandra's durable, eventually consistent, distributed design. If Cassandra had a traditional design for deletes, any nodes that were down at the time of a delete would not receive the delete. Once one of these nodes came back online, it would mistakenly think that all of the nodes that had received the delete had actually missed a write (the data that it still has because it missed the delete), and it would start repairing all of the other nodes. So Cassandra needs a more sophisticated mechanism to support deletes. That mechanism is called a tombstone.A tombstone is a special marker issued in a delete, acting as a placeholder. If any replica did not receive the delete operation, the tombstone can later be propagated to those replicas when they are available again. The net effect of this design is that your data store will not immediately shrink in size following a delete. Each node keeps track of the age of all its tombstones. Once they reach the age configured in gc_grace_seconds (which is 10 days by default), then a compaction is run, the tombstones are garbage collected, and the corresponding disk space is recovered.Because SSTables are immutable, the data is not deleted from the SSTable. On compaction, tombstones are accounted for, merged data is sorted, a new index is created over the sorted data, and the freshly merged, sorted, and indexed data is written to a single new file.
If your application generates a large number of tombstones, Cassandra's read performance can begin to be impacted by having to traverse over these tombstones as it reads SSTable files.
Here are a few techniques to help minimize the impact of tombstones on your cluster:

- Avoid writing NULL values into your tables, as these are interpreted as deletes.
- Delete data at the largest granularity you can, ideally entire partitions at once. This will minimize the number of tombstones you create.
  https://thelastpickle.com/blog/2016/07/27/about-deletes-and-tombstones.html
- Exercise care when updating collections. If possible, avoid replacing the entire contents of a list, set, or map, as this will generate tombstones for all of the previous content. Instead, update only the elements you need to modify.
- Use Cassandra's time-to-live (TTL) feature when inserting data, which allows Cassandra to expire data automatically on your behalf.

Configuring and Deploying Cassandra