**Commit Log:**

Writes in Cassandra are durable. All writes to a replica node are recorded both in memory and in a commit log on disk before they are acknowledged as a success. If a crash or server failure occurs before the memtables are flushed to disk, the commit log is replayed on restart to recover any lost writes.

Commit Log is present in disk and data is written to it. It helps in case of failure or crash.

All data is written first to the commit log for durability. After all its data has been flushed to SSTables, it can be archived, deleted, or recycled

Commit log is append only datastructure. Commitlog is immutable

Every write includes a timestamp.

We just append at the end of commit log and is just append append.

Suggestion: Store commit log into the separate hard disk as of SSTable.

The commit log is a crash-recovery mechanism that supports Cassandra’s durability goals. A write will not count as successful 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), and it will still be possible to recover the data. If you shut down the database or it crashes unexpectedly, the commit log can ensure that data is not lost. That’s because the next time you start the node, the commit log gets replayed. In fact, that’s the only time the commit log is read; clients never read from it.

**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.

**Difference between Memtable and Commit Log:**

Memtable is always ordered by partitioning key and clustering column whereas Commit log stores sequentially, every record just append to another commit log.

Memtable will be used for serve the read request while commitlog will be used in case of failure to replay.

Cassandra send ACK say it’s done.

Commitlog just append append.

Internally it is just changing the reference.

**SSTable:**

A sorted string table (SSTable) is an immutable data file to which Cassandra writes memtables periodically. SSTables are append only and stored on disk sequentially and maintained for each Cassandra table.

SST Table means data is first sorted by partitioning key and then by clustering column values. That data is immutable.

* Immutable data file for row storage.
* Every write includes a timestamp of when it was written.
* Partition is spread across multiple SSTables.
* Same column can be in multiple SSTables.
* Merged through compaction, only latest timestamp is kept.
* Delete are written as tombstones.
* Easy backups!

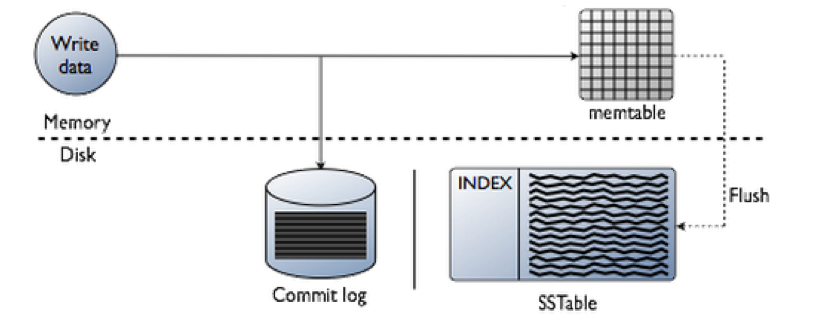
**Tombstones:**

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. A tombstone is a deletion marker that is required to suppress older data in SSTables until compaction can run.

**Garbage Collection Grace Seconds**

This is the amount of time that the server will wait to garbage-collect a tombstone. By default, it’s set to 864,000 seconds, the equivalent of 10 days. Cassandra keeps track of tombstone age, and once a tombstone is older than GCGraceSeconds, it will be garbage-collected. 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 is treated as failed and replaced.

**How is data written?**



**All writes in Cassandra are append-only.**

Data written to a Cassandra node is first recorded in an on-disk commit log and then written to a memory-based structure called a memtable. When a memtable's size exceeds a configurable threshold, the data is written to an immutable file on disk called an SSTable. Buffering writes in memory in this way allows writes always to be a fully sequential operation, with many megabytes of disk I/O happening at the same time, rather than one at a time over a long period. This architecture gives Cassandra its legendary write performance.

**Compaction:**

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, tombstones are discarded, and a new index is created.

Compaction is the process of freeing up space by merging large accumulated datafiles.

On compaction, the 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 SSTable (each SSTable consists of multiple files including: Data, Index, and Filter).

The available strategies include:

• SizeTieredCompactionStrategy (STCS) is the default compaction strategy and is recommended for write-intensive tables

• LeveledCompactionStrategy (LCS) is recommended for read-intensive tables

• DateTieredCompactionStrategy (DTCS), which is intended for time series or

Otherwise date-based data.

nodetool <options> compact <keyspace> <table> ...

Other options are:

* keyspace is the name of a keyspace.
* table is one or more table names, separated by a space.

Update a table to set the compaction strategy using the ALTER TABLE statement.

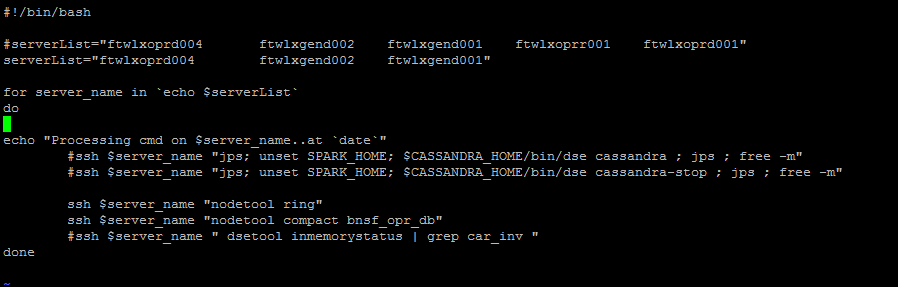
**ALTER TABLE users WITH**

**compaction = { 'class' : 'LeveledCompactionStrategy' }**

**ALTER** **TABLE** **users**

**WITH** compaction =

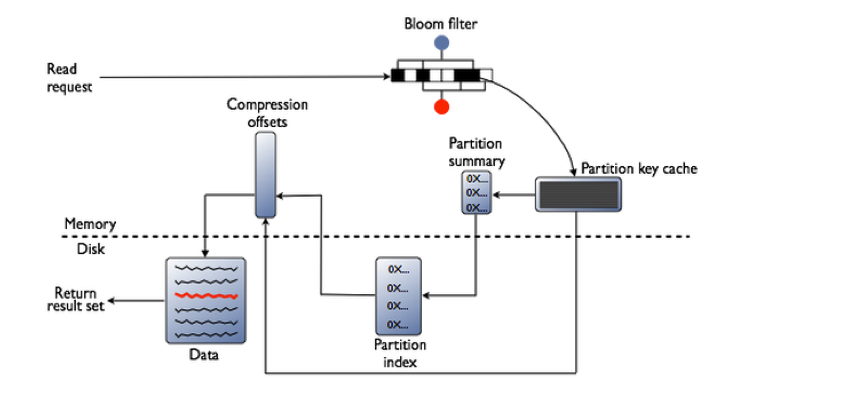
{'class' : 'SizeTieredCompactionStrategy', 'min\_threshold' : 6 }



**Bloom Filters**

Bloom filters are used to boost the performance of reads. Bloom filters are very fast, non-deterministic algorithms for testing whether an element is a member of a set.

**How is Data Read?**



# Repairing nodes

Nodetool repair:

Node tool repair is used to sync data across nodes in case data is not synced.

Node repair makes data on a replica consistent with data on other nodes. The [nodetool repair](https://docs.datastax.com/en/archived/cassandra/2.0/cassandra/tools/toolsRepair.html" \o "Repairs one or more tables.) command repairs inconsistencies across all of the replicas for a given range of data.

* As a best practice, you should schedule repairs weekly.

**Note:** If deletions never occur, you should still schedule regular repairs. Be aware that setting a column to null is a delete.

* During node recovery. For example, when bringing a node back into the cluster after a failure.
* On nodes containing data that is not read frequently.
* To update data on a node that has been down.

Guidelines for running routine node repair include:

* The hard requirement for routine repair frequency is the value of [gc\_grace\_seconds](https://docs.datastax.com/en/cql/3.1/cql/cql_reference/tabProp.html" \t "_blank). Run a repair operation at least once on each node within this time period. Following this important guideline ensures that deletes are properly handled in the cluster.
* Schedule regular repair operations for low-usage hours.
* To minimize impact, use nodetool repair and do not invoke more than one repair at a time.
* In systems that seldom delete or overwrite data, you can raise the value of gc\_grace with minimal impact to disk space. This allows wider intervals for scheduling repair operations with the [nodetool utility](https://docs.datastax.com/en/archived/cassandra/2.0/cassandra/tools/toolsNodetool_r.html" \o "A command line interface for Cassandra for managing a cluster.).

**COMMAND:**

./nodetool repair –full -seq

# [Data consistency](https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlDataConsistencyTOC.html)

Consistency refers to how up-to-date and synchronized all replicas of a row of Cassandra data are at any given moment. Ongoing [repair operations](https://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsRepairNodesTOC.html) in Cassandra ensure that all replicas of a row will eventually be consistent. Repairs work to decrease the variability in replica data, but constant data traffic through a widely distributed system can lead to inconsistency (stale data) at any given time.

The consistency level determines the number of replicas that need to acknowledge the read or write operation success to the client application. For read operations, the read consistency level specifies how many replicas must respond to a read request before returning data to the client application. If a read operation reveals inconsistency among replicas.

For write operations, the write consistency level specified how many replicas must respond to a write request before the write is considered successful. Even at low consistency levels, Cassandra writes to all replicas of the partition key, including replicas in other datacenters.