How is data written?(Cassandra Write PATH)

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

* Logging data in the commit log
* Writing data to the memtable
* Flushing data from the memtable
* Storing data on disk in SSTables

Logging writes and memtable storage

When a write occurs, Cassandra stores the data in a memory structure called memtable, and to provide [configurable durability](http://docs.datastax.com/en/cassandra/3.0/cassandra/configuration/configCassandra_yaml.html), it also appends writes to the commit log on disk. The commit log receives every write made to a Cassandra node, and these durable writes survive permanently even if power fails on a node. The memtable is a write-back cache of data partitions that Cassandra looks up by key. The memtable stores writes in sorted order until reaching a configurable limit, and then is flushed.

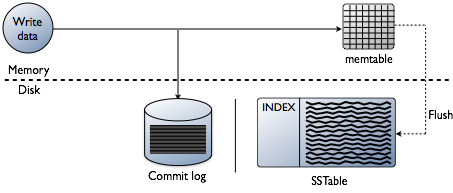
Flushing data from the memtable

To flush the data, Cassandra writes the data to disk, in the memtable-sorted order.. A partition index is also created on the disk that maps the tokens to a location on disk. When the memtable content exceeds the [configurable threshold](http://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsMemtableThruput.html) or the commitlog space exceeds the **commitlog\_total\_space\_in\_mb**, the memtable is put in a queue that is flushed to disk. The queue can be configured with the**memtable\_heap\_space\_in\_mb** or **memtable\_offheap\_space\_in\_mb** setting in the [cassandra.yaml](http://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlHowDataWritten.html) file. If the data to be flushed exceeds the **memtable\_cleanup\_threshold**, Cassandra blocks writes until the next flush succeeds. You can manually flush a table using [nodetool flush](http://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsFlush.html" \o "Flushes one or more tables from the memtable.)or [nodetool drain](http://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsDrain.html" \o "Drains the node.) (flushes memtables without listening for connections to other nodes). To reduce the commit log replay time, the recommended best practice is to flush the memtable before you restart the nodes. If a node stops working, replaying the commit log restores to the memtable the writes that were there before it stopped.

Data in the commit log is purged after its corresponding data in the memtable is flushed to an SSTable on disk.

Storing data on disk in SSTables

Memtables and SSTables are maintained per table. The commit log is shared among tables. SSTables are immutable, not written to again after the memtable is flushed. Consequently, a partition is typically stored across multiple SSTable files. A number of other SSTable structures exist to assist read operations:



For each SSTable, Cassandra creates these structures:

1. **Data (Data.db)**

The SSTable data

1. **Primary Index (Index.db)**

Index of the row keys with pointers to their positions in the data file

1. [**Bloom filter**](http://docs.datastax.com/en/glossary/doc/glossary/gloss_bloom_filter.html)**(Filter.db)**

A structure stored in memory that checks if row data exists in the memtable before accessing SSTables on disk

1. **Compression Information (CompressionInfo.db)**

A file holding information about uncompressed data length, chunk offsets and other compression information

1. **Statistics (Statistics.db)**

Statistical metadata about the content of the SSTable

1. **Digest (Digest.crc32, Digest.adler32, Digest.sha1)**

A file holding adler32 checksum of the data file

1. **CRC (CRC.db)**

A file holding the CRC32 for chunks in an a uncompressed file.

1. **SSTable Index Summary (SUMMARY.db)**

A sample of the partition index stored in memory

1. **SSTable Table of Contents (TOC.txt)**

A file that stores the list of all components for the SSTable TOC

1. **Secondary Index (SI\_.\*.db)**

Built-in secondary index. Multiple SIs may exist per SSTable

The SSTables are files stored on disk. The naming convention for SSTable files has changed with Cassandra 2.2 and later to shorten the file path. The data files are stored in a data directory that varies with installation. For each keyspace, a directory within the data directory stores each table. For example,/data/data/ks1/cf1-5be396077b811e3a3ab9dc4b9ac088d/la-1-big-Data.db represents a data file. **ks1** represents the keyspace name to distinguish the keyspace for streaming or bulk loading data. A hexadecimal string, 5be396077b811e3a3ab9dc4b9ac088d in this example, is appended to table names to represent unique table IDs.

Cassandra creates a subdirectory for each table, which allows you to symlink a table to a chosen physical drive or data volume. This provides the capability to move very active tables to faster media, such as SSDs for better performance, and also divides tables across all attached storage devices for better I/O balance at the storage layer.

How is data maintained?

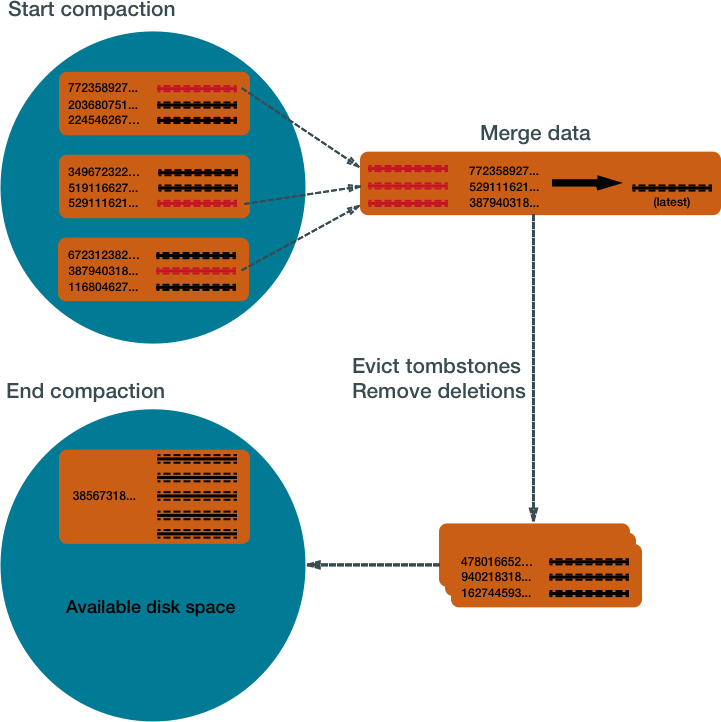
The Cassandra write process stores data in files called SSTables. SSTables are immutable. Instead of overwriting existing rows with inserts or updates, Cassandra writes new timestamped versions of the inserted or updated data in new SSTables. Cassandra does not perform deletes by removing the deleted data: instead, Cassandra marks it with [tombstones](http://docs.datastax.com/en/glossary/doc/glossary/gloss_tombstone.html).

Over time, Cassandra may write many versions of a row in different SSTables. Each version may have a unique set of columns stored with a different timestamp. As SSTables accumulate, the distribution of data can require accessing more and more SSTables to retrieve a complete row.

To keep the database healthy, Cassandra periodically merges SSTables and discards old data. This process is called compaction.

Compaction

Compaction works on a collection of SSTables. From these SSTables, compaction collects all versions of each unique row and assembles one complete row, using the most up-to-date version (by timestamp) of each of the row's columns. The merge process is performant, because rows are sorted by [partition key](http://docs.datastax.com/en/glossary/doc/glossary/gloss_partition_key.html)within each SSTable, and the merge process does not use random I/O. The new versions of each row is written to a new SSTable. The old versions, along with any rows that are ready for deletion, are left in the old SSTables, and are deleted as soon as pending reads are completed.



Compaction causes a temporary spike in disk space usage and disk I/O while old and new SSTables co-exist. As it completes, compaction frees up disk space occupied by old SSTables. It improves read performance by incrementally replacing old SSTables with compacted SSTables. Cassandra can read data directly from the new SSTable even before it finishes writing, instead of waiting for the entire compaction process to finish.

As Cassandra processes writes and reads, it replaces the old SSTables with new SSTables in the page cache. The process of caching the new SSTable, while directing reads away from the old one, is incremental — it does not cause a the dramatic cache miss. Cassandra provides predictable high performance even under heavy load.

Compaction strategies

Cassandra supports different compaction strategies, which control how which SSTables are chosen for compaction, and how the compacted rows are sorted into new SSTables. Each strategy has its own strengths. The sections that follow explain each of Cassandra's compaction strategies.

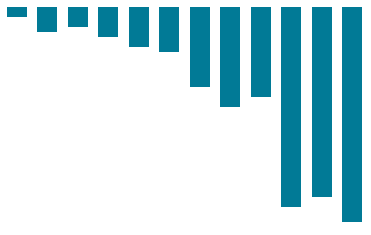
Although each of the following sections starts with a generalized recommendation, many factors complicate the choice of a compaction strategy. See [Which compaction strategy is best?](http://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlHowDataMaintain.html#dmlHowDataMaintain__dml_which_compaction_strategy_is_best).

**1.SizeTieredCompactionStrategy (STCS)**

Recommended for write-intensive workloads.

The SizeTieredCompactionStrategy (STCS) initiates compaction when Cassandra has accumulated a set number (default: 4) of similar-sized SSTables. STCS merges these SSTables into one larger SSTable. As these larger SSTables accumulate, STCS merges these into even larger SSTables. At any given time, several SSTables of varying sizes are present.

*Size tiered compaction after many inserts*



While STCS works well to compact a write-intensive workload, it makes reads slower because the merge-by-size process does not group data by rows. This makes it more likely that versions of a particular row may be spread over many SSTables. Also, STCS does not evict deleted data predictably because its trigger for compaction is SSTable size, and SSTables might not grow quickly enough to merge and evict old data. As the largest SSTables grow in size, the amount of disk space needed for both the new and old SSTables simultaneously during STCS compaction can outstrip a typical amount of disk space on a node.

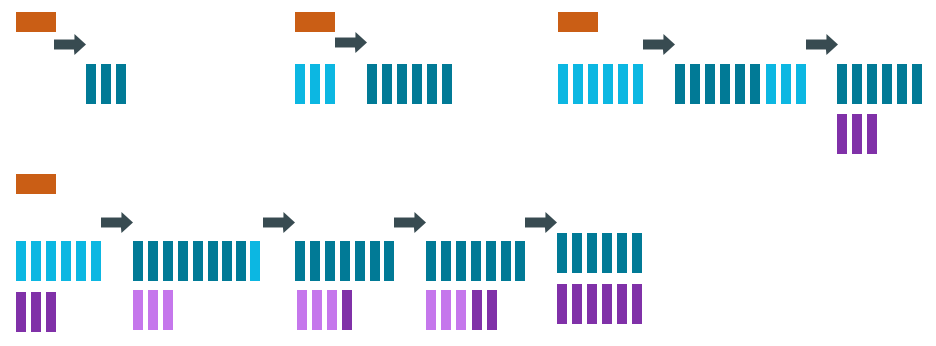
* **Pros:** Compacts write-intensive workload very well.
* **Cons:** Can hold onto stale data too long. Amount of memory needed increases over time.

**2. LeveledCompactionStrategy (LCS)**

Recommended for read-intensive workloads.

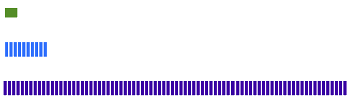
The LeveledCompactionStrategy (LCS) alleviates some of the read operation issues with STCS. This strategy works with a series of levels. First, data in memtables is flushed to SSTables in the first level (L0). LCS compaction merges these first SSTables with larger SSTables in level L1.

*Leveled compaction — adding SSTables*



The SSTables in levels greater than L1 are merged into SSTables with a size greater than or equal to sstable\_size\_in\_mb (default: 160 MB). If a L1 SSTable stores data of a partition that is larger than L2, LCS moves the SSTable past L2 to the next level up.

*Leveled compaction after many inserts*



In each of the levels above L0, LCS creates SSTables that are about the same size. Each level is 10X the size of the last level, so level L1 has 10X as many SSTables as L0, and level L2 has 100X as many. If the result of the compaction is more than 10 SSTables in level L1, the excess SSTables are moved to level L2.

The LCS compaction process guarantees that the SSTables within each level starting with L1 have non-overlapping data. For many reads, this process enables Cassandra to retrieve all the required data from only one or two SSTables. In fact, 90% of all reads can be satisfied from one SSTable. Since LCS does not compact L0 tables, however, resource-intensive reads involving many L0 SSTables may still occur.

At levels beyond L0, LCS requires less disk space for compacting — generally, 10X the fixed size of the SSTable. Obsolete data is evicted more often, so deleted data uses smaller portions of the SSTables on disk. However, LCS compaction operations take place more often and place more I/O burden on the node. For write-intensive workloads, the payoff of using this strategy is generally not worth the performance loss to I/O operations. In many cases, tests of LCS-configured tables reveal I/O saturation on writes and compactions.

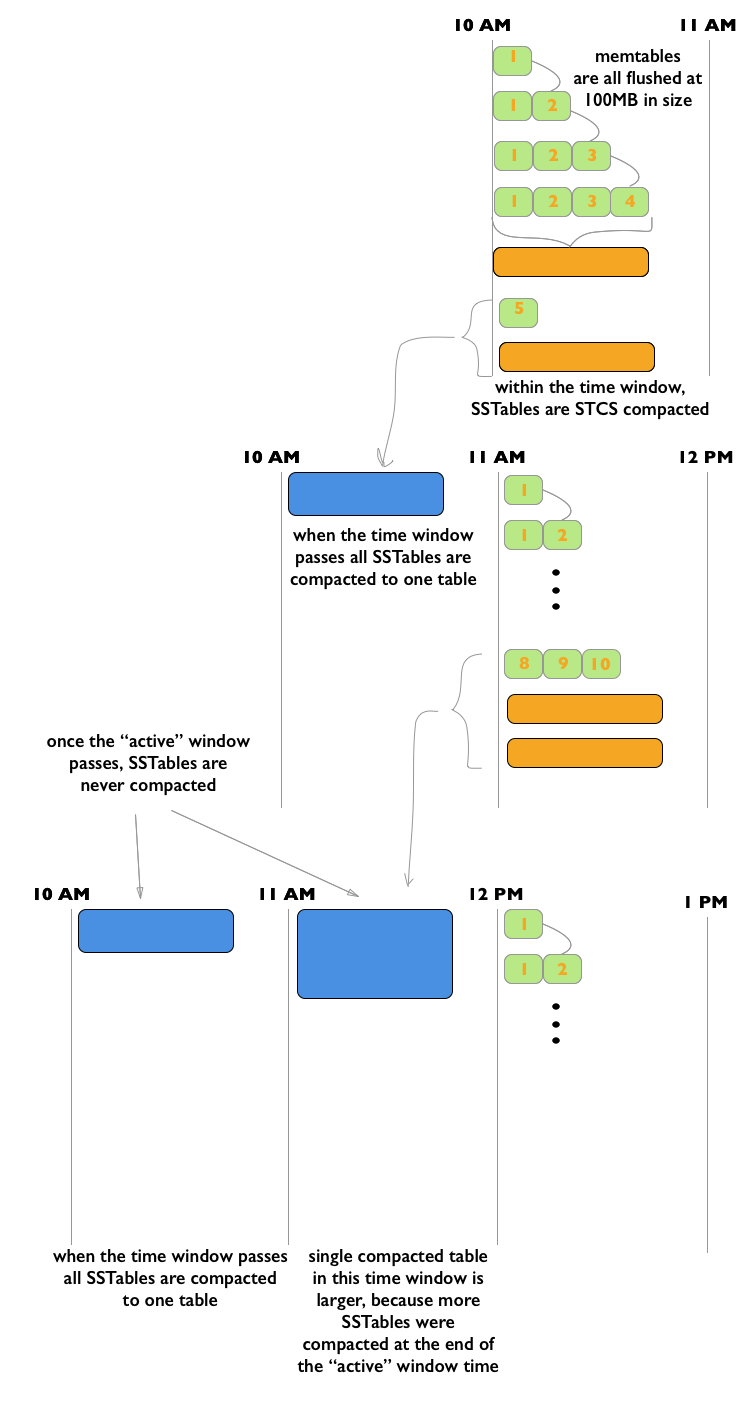
**Note:** Cassandra bypasses compaction operations when [bootstrapping](http://docs.datastax.com/en/glossary/doc/glossary/gloss_bootstrap.html) a new node using LCS into a cluster. The original data is moved directly to the correct level because there is no existing data, so no partition overlap per level is present. For more information, see [Apache Cassandra 2.2 - Bootstrapping Performance Improvements for Leveled Compaction](https://www.datastax.com/dev/blog/bootstrapping-performance-improvements-for-leveled-compaction).

* **Pros:** Disk requirements are easier to predict. Read operation latency is more predictable. Stale data is evicted more frequently.
* **Cons:** Much higher I/O utilization impacting operation latency

**3. TimeWindowCompactionStrategy (TWCS)**

Recommended for time series and expiring TTL workloads.

The TimeWindowCompactionStrategy (TWCS) is similar to DTCS with simpler settings. TWCS groups SSTables using a series of time windows. During compaction, TWCS applies STCS to uncompacted SSTables in the most recent time window. At the end of a time window, TWCS compacts all SSTables that fall into that time window into a single SSTable based on the SSTable maximum timestamp. Once the major compaction for a time window is completed, no further compaction of the data will ever occur. The process starts over with the SSTables written in the next time window.



As the figure shows, from 10 AM to 11 AM, the memtables are flushed from memory into 100MB SSTables. These SSTables are compacted into larger SSTables using STCS. At 11 AM, all these SSTables are compacted into a single SSTable, and never compacted again by TWCS. At 12 NOON, the new SSTables created between 11 AM and 12 NOON are compacted using STCS, and at the end of the time window the TWCS compaction repeats. Notice that each TWCS time window contains varying amounts of data.

**Note:** For an animated explanation, see the [Datastax Academy Time Window Compaction Strategy](https://academy.datastax.com/courses/ds210-datastax-enterprise-operations-apache-cassandra/time-windowed-compaction" \t "_blank) video.

The TWCS configuration has two main property settings:

* **compaction\_window\_unit**: time unit used to define the window size (milliseconds, seconds, hours, and so on)
* **compaction\_window\_size**: how many units per window (1,2,3, and so on)

The configuration for the above example: compaction\_window\_unit = ‘minutes’,compaction\_window\_size = 60

**Pros**: Used for time series data, stored in tables that use the default TTL for all data. Simpler configuration than that of DTCS.

**Cons**: Not appropriate if out-of-sequence time data is required, since SSTables will not compact as well. Also, not appropriate for data without a TTL, as storage will grow without bound. Less fine-tuned configuration is possible than with DTCS.

**4. DateTieredCompactionStrategy (DTCS)**

Deprecated in Cassandra 3.0.8/3.8.

The DateTieredCompactionStrategy (DTCS) is similar to STCS. But instead of compacting based on SSTable size, DTCS compacts based on SSTable age. (Each column in an SSTable is marked with the timestamp at write time. As the *age* of an SSTable, DTCS uses the oldest (minimum) timestamp of any column the SSTable contains.)

Configuring the DTCS time window ensures that new and old data are not mixed in merged SSTables. In fact, using Time-To-Live (TTL) timestamps, DTCS can eject whole SSTables containing expired data. This strategy often generates similar-sized SSTables if time series data is ingested at a steady rate.

DTCS compacts SSTables into larger tables, as with STCS, when the system accumulates a configurable number of SSTables within a configurable time interval. However, DTCS skips compacting SSTables that reach a configurable age. This logic reduces the number of times data is rewritten. Queries that ask for data in a particular last time interval, such as an hour, can be executed very efficiently on DTCS-compacted SSTables (particularly if the requested time interval is coordinated with the configured interval for compaction).

One use case that can cause difficulty with this strategy is out-of-sequence writing. For example, an operation that writes a timestamped record with a timestamp outside the current time window. Read repairs can inject out-of-sequence timestamps, so be sure to turn off read repairs when using DTCS. For more information, see [DateTieredCompactionStrategy: Notes from the Field](https://www.datastax.com/dev/blog/dtcs-notes-from-the-field" \t "_blank).

* **Pros**: Specifically designed for time series data, stored in tables that use the default TTL. DTCS is a better choice when fine-tuning is required to meet space-related SLAs.
* **Cons**: Insertion of records out of time sequence (by repairs or hint replaying) can increase latency or cause errors. In some cases, it may be necessary to turn off read repair and carefully test and control the use of TIMESTAMP options in BATCH, DELETE, INSERT and UPDATE CQL commands.

Which compaction strategy is best?

To implement the best compaction strategy:

1. Review your application's requirements.
2. Configure the table to use the most appropriate strategy.
3. Test the compaction strategies against your data.

The following questions are based on the experiences of Cassandra developers and users with the strategies described above.

**Does your table process time series data?**

If so, your best choices are TWCS or DTCS. For details, read the descriptions on this page.

If your table is not focused on time series data, the choice becomes more complicated. The following questions introduce other considerations that may guide your choice.

**Does your table handle more reads than writes, or more writes than reads?**

LCS is a good choice if your table processes twice as many reads as writes or more – especially randomized reads. If the proportion of reads to writes is closer, the performance hit exacted by LCS may not be worth the benefit. Be aware that LCS can be quickly overwhelmed by a high volume of writes.

**Does the data in your table change often?**

One advantage of LCS is that it keeps related data in a small set of SSTables. If your data is *immutable* or not subject to frequent [upserts](http://docs.datastax.com/en/glossary/doc/glossary/gloss_upsert.html" \t "_blank), STCS accomplishes the same type of grouping without the LCS performance hit.

**Do you require predictable levels of read and write activity?**

LCS keeps the SSTables within predictable sizes and numbers. For example, if your table's read/write ratio is small, and it is expected to conform to a Service Level Agreements (SLAs) for reads, it may be worth taking the write performance penalty of LCS in order to keep read rates and latency at predictable levels. And you may be able to overcome this write penalty through horizontal scaling (adding more nodes).

**Will your table be populated by a batch process?**

On both batch reads and batch writes, STCS performs better than LCS. The batch process causes little or no fragmentation, so the benefits of LCS are not realized; batch processes can overwhelm LCS-configured tables.

**Does your system have limited disk space?**

LCS handles disk space more efficiently than STCS: it requires about 10% *headroom* in addition to the space occupied by the data is handles. STCS and DTCS generally require, in some cases, as much as 50% more than the data space.

**Is your system reaching its limits for I/O?**

LCS is significantly more I/O intensive than DTCS or STCS. Switching to LCS may introduce extra I/O load that offsets the advantages.

Testing compaction strategies

Suggestions for determining which compaction strategy is best for your system:

* Create a three-node cluster using one of the compaction strategies, stress test the cluster using cassandra-stress, and measure the results.
* Set up a node on your existing cluster and use Cassandra's write survey mode to sample live data. See [What’s new in Cassandra 1.1: live traffic sampling](https://www.datastax.com/dev/blog/whats-new-in-cassandra-1-1-live-traffic-sampling).

Configuring and running compaction

Set the compaction strategy for a table in the parameters for the CREATE TABLE or ALTER TABLEcommand. For details, see [Table properties](http://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlCreateTable.html#tabProp__moreCompaction).

You can start compaction manually using the [nodetool compact](http://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsCompact.html" \o "Forces a major compaction on one or more tables.) command.

More information about compaction

The following blog posts and videos provide more information from developers that have tested compaction strategies:

* [When to Use Leveled Compaction](https://www.datastax.com/dev/blog/when-to-use-leveled-compaction)
* [Leveled compaction in Apache Cassandra](https://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra)
* [Using TimeWindowCompactionStrategy for Time Series Workloads](https://www.youtube.com/watch?v=PWtekUWCIaw)
* [DateTieredCompactionStrategy: Notes from the Field](https://www.datastax.com/dev/blog/dtcs-notes-from-the-field)
* [Date-Tiered Compaction in Cassandra](https://labs.spotify.com/2014/12/18/date-tiered-compaction/)
* [DateTieredCompactionStrategy: Compaction for Time Series Data](https://www.datastax.com/dev/blog/datetieredcompactionstrategy).
* [What delays a tombstone purge when using LCS in Cassandra](http://stackoverflow.com/questions/27862808/what-delays-a-tombstone-purge-when-using-lcs-in-cassandra)

# How is data updated?

Cassandra treats each new row as an [upsert](http://docs.datastax.com/en/glossary/doc/glossary/gloss_upsert.html" \t "_blank): if the new row has the same primary key as that of an existing row, Cassandra processes it as an update to the existing row.

During a [write](http://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlHowDataWritten.html), Cassandra adds each new row to the database without checking on whether a duplicate record exists. This policy makes it possible that many versions of the same row may exist in the database. For more details about writes, see [How is data written?](http://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlHowDataWritten.html)

Periodically, the rows stored in memory are streamed to disk into structures called SSTables. At certain intervals, Cassandra [compacts](http://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlHowDataMaintain.html#dmlHowDataMaintain__dml-compaction) smaller SSTables into larger SSTables. If Cassandra encounters two or more versions of the same row during this process, Cassandra only writes the most recent version to the new SSTable. After compaction, Cassandra drops the original SSTables, deleting the outdated rows.

Most Cassandra installations store replicas of each row on two or more nodes. Each node performs compaction independently. This means that even though out-of-date versions of a row have been dropped from one node, they may still exist on another node.

This is why Cassandra performs another round of comparisons during a [read process](http://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlAboutReads.html). When a client requests data with a particular primary key, Cassandra retrieves many versions of the row from one or more replicas. The version with the most recent timestamp is the only one returned to the client ("last-write-wins").

**Note:** Some database operations may only write partial updates of a row, so some versions of a row may include some columns, but not all. During a compaction or write, Cassandra assembles a complete version of each row from the partial updates, using the most recent version of each column.

How is data deleted?

Cassandra's processes for deleting data are designed to improve performance, and to work with Cassandra's built-in properties for data distribution and fault-tolerance.

Cassandra treats a delete as an insert or [upsert](http://docs.datastax.com/en/glossary/doc/glossary/gloss_upsert.html" \t "_blank). The data being added to the partition in the [DELETE](http://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlDelete.html) command is a deletion marker called a [tombstone](http://docs.datastax.com/en/glossary/doc/glossary/gloss_tombstone.html). The tombstones go through Cassandra's write path, and are written to SSTables on one or more nodes. The key difference feature of a tombstone: it has a built-in expiration date/time. At the end of its expiration period (for details see below) the tombstone is deleted as part of Cassandra's normal [compaction](http://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlHowDataMaintain.html#dmlHowDataMaintain__dml-compaction) process.

You can also mark a Cassandra record (row or column) with a [time-to-live](http://docs.datastax.com/en/cql/3.3/cql/cql_using/useTTL.html) value. After this amount of time has ended, Cassandra marks the record with a tombstone, and handles it like other tombstoned records.

Deletion in a distributed system

In a multi-node cluster, Cassandra can store replicas of the same data on two or more nodes. This helps prevent data loss, but it complicates the delete process. If a node receives a delete for data it stores locally, the node tombstones the specified record and tries to pass the tombstone to other nodes containing replicas of that record. But if one replica node is unresponsive at that time, it does not receive the tombstone immediately, so it still contains the pre-delete version of the record. If the tombstoned record has already been deleted from the rest of the cluster befor that node recovers, Cassandra treats the record on the recovered node as new data, and propagates it to the rest of the cluster. This kind of deleted but persistent record is called a [zombie](http://docs.datastax.com/en/glossary/doc/glossary/gloss_zombie.html).

To prevent the reappearance of zombies, Cassandra gives each tombstone a *grace period*. The purpose of the grace period is to give unresponsive nodes time to recover and process tombstones normally. If a client writes a new update to the tombstoned record during the grace period, Cassandra overwrites the tombstone. If a client sends a read for that record during the grace period, Cassandra disregards the tombstone and retrieves the record from other replicas if possible.

When an unresponsive node recovers, Cassandra uses [hinted handoff](http://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsRepairNodesHintedHandoff.html) to replay the database [mutations](http://docs.datastax.com/en/glossary/doc/glossary/gloss_mutation.html" \t "_blank)the node missed while it was down. Cassandra does not replay a mutation for a tombstoned record during its grace period. But if the node does not recover until after the grace period ends, Cassandra may miss the deletion.

After the tombstone's grace period ends, Cassandra deletes the tombstone during compaction.

The grace period for a tombstone is set by the property **gc\_grace\_seconds**. Its default value is 864000 seconds (ten days). Each table can have its own value for this property.

More about Cassandra deletes

Details:

* The expiration date/time for a tombstone is the date/time of its creation plus the value of the [table property](http://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlCreateTable.html#tabProp) *gc\_grace\_seconds*.
* Cassandra also supports [Batch data insertion and updates](http://docs.datastax.com/en/cql/3.3/cql/cql_using/useBatchTOC.html). This procedure also introduces the danger of replaying a record insertion after that record has been removed from the rest of the cluster. Cassandra does not replay a batched mutation for a tombstoned record that is still within its grace period.
* On a single-node cluster, you can set *gc\_grace\_seconds* to 0 (zero).
* To completely prevent the reappearance of zombie records, run [nodetool repair](http://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsRepair.html" \o "Repairs one or more tables.) on a node after it recovers, and on each table every *gc\_grace\_seconds*.
* If all records in a table are given a TTL at creation, and all are allowed to expire and not deleted manually, it is not necessary to run [nodetool repair](http://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsRepair.html" \o "Repairs one or more tables.) for that table on a regular basis.
* If you use the [SizeTieredCompactionStrategy or DateTieredCompactionStrategy](http://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlHowDataMaintain.html" \o "Cassandra processes data at several stages on the write path. Compaction to maintain healthy SSTables is the last step in the write path process.), you can delete tombstones immediately by [manually starting the compaction process](http://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsCompact.html).

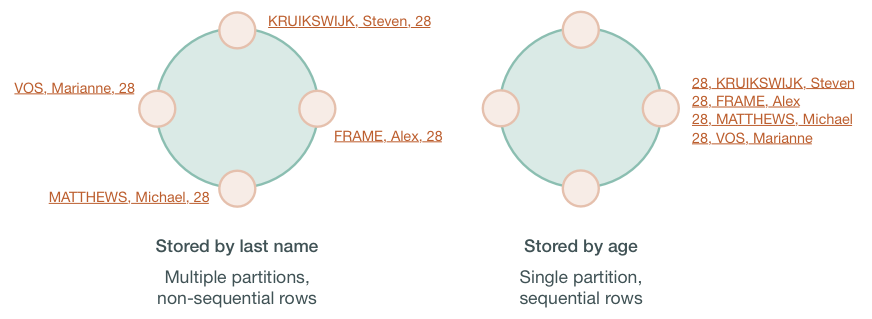
**CAUTION:**

If you force compaction, Cassandra may create one very large SSTable from all the data. Cassandra will not trigger another compaction for a long time. The data in the SSTable created during the forced compaction can grow very stale during this long period of non-compaction.

* Cassandra allows you to set a *default\_time\_to\_live* [property](http://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlCreateTable.html#tabProp) for an entire table. Columns and rows marked with regular TTLs are processed as described above; but when a record exceeds the table-level TTL, Cassandra deletes it immediately, without tombstoning or compaction.
* Cassandra supports immediate deletion through the [DROP KEYSPACE](http://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlDropKeyspace.html) and [DROP TABLE](http://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlDropTable.html) statements.

# How are indexes stored and updated?

Secondary indexes are used to filter a table for data stored in non-primary key columns. For example, a table storing cyclist names and ages using the last name of the cyclist as the primary key might have a secondary index on the age to allow queries by age. Querying to match a non-primary key column is an anti-pattern, as querying should always result in a continuous slice of data retrieved from the table.



If the table rows are stored based on last names, the table may be spread across several partitions stored on different nodes. Queries based on a particular range of last names, such as all cyclists with the last name Matthews will retrieve sequential rows from the table, but a query based on the age, such as all cyclists who are 28, will require all nodes to be queried for a value. Non-primary keys play no role in ordering the data in storage, thus querying for a particular value of a non-primary key column results in scanning all partitions. Scanning all partitions generally results in a prohibitive read latency, and is not allowed.

Secondary indexes can be built for a column in a table. These indexes are stored locally on each node in a hidden table and built in a background process. If a secondary index is used in a query that is not restricted to a particular partition key, the query will have prohibitive read latency because all nodes will be queried. A query with these parameters is only allowed if the query option ALLOW FILTERING is used. This option is not appropriate for production environments. If a query includes both a partition key condition and a secondary index column condition, the query will be successful because the query can be directed to a single node partition.

This technique, however, does not guarantee trouble-free indexing, so know [when and when not to use an index](http://docs.datastax.com/en/cql/3.3/cql/cql_using/useWhenIndex.html). In the example shown above, an index on the age could be used, but a better solution is to create a materialized view or additional table that is ordered by age.

As with relational databases, keeping indexes up to date uses processing time and resources, so unnecessary indexes should be avoided. When a column is updated, the index is updated as well. If the old column value still exists in the memtable, which typically occurs when updating a small set of rows repeatedly, Cassandra removes the corresponding obsolete index entry; otherwise, the old entry remains to be purged by compaction. If a read sees a stale index entry before compaction purges it, the reader thread invalidates it.

How is data read?

To satisfy a read, Cassandra must combine results from the active memtable and potentially multiple SSTables.

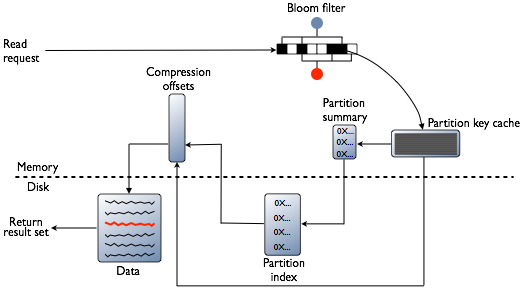
Cassandra processes data at several stages on the read path to discover where the data is stored, starting with the data in the memtable and finishing with SSTables:

* Check the memtable
* Check row cache, if enabled
* Checks Bloom filter
* Checks partition key cache, if enabled
* Goes directly to the compression offset map if a partition key is found in the partition key cache, or checks the partition summary if not

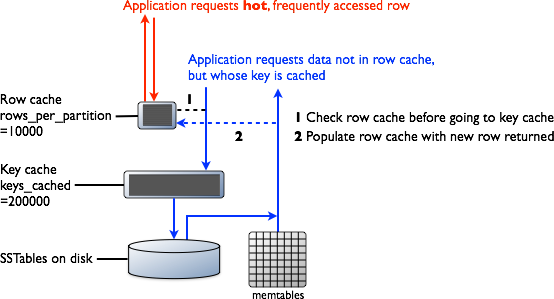
If the partition summary is checked, then the partition index is accessed

* Locates the data on disk using the compression offset map
* Fetches the data from the SSTable on disk

*Read request flow*



*Row cache and Key cache request flow*



Memtable

If the memtable has the desired partition data, then the data is read and then merged with the data from the SSTables. The SSTable data is accessed as shown in the following steps.

Row Cache

Typical of any database, reads are fastest when the most in-demand data fits into memory. The operating system page cache is best at improving performance, although the row cache can provide some improvement for very read-intensive operations, where read operations are 95% of the load. Row cache is contra-indicated for write-intensive operations. The row cache, if enabled, stores a subset of the partition data stored on disk in the SSTables in memory. In Cassandra 2.2 and later, it is stored in fully off-heap memory using a new implementation that relieves garbage collection pressure in the JVM. The subset stored in the row cache use a configurable amount of memory for a specified period of time. The row cache uses LRU (least-recently-used) eviction to reclaim memory when the cache has filled up.

The row cache size is configurable, as is the number of rows to store. Configuring the number of rows to be stored is a useful feature, making a "Last 10 Items" query very fast to read. If row cache is enabled, desired partition data is read from the row cache, potentially saving two seeks to disk for the data. The rows stored in row cache are frequently accessed rows that are merged and saved to the row cache from the SSTables as they are accessed. After storage, the data is available to later queries. The row cache is not write-through. If a write comes in for the row, the cache for that row is invalidated and is not cached again until the row is read. Similarly, if a partition is updated, the entire partition is evicted from the cache. When the desired partition data is not found in the row cache, then the [Bloom filter](http://docs.datastax.com/en/glossary/doc/glossary/gloss_bloom_filter.html) is checked.

Bloom Filter

First, Cassandra checks the Bloom filter to discover which SSTables are likely to have the request partition data. The Bloom filter is stored in off-heap memory. Each SSTable has a Bloom filter associated with it. A Bloom filter can establish that a SSTable does not contain certain partition data. A Bloom filter can also find the likelihood that partition data is stored in a SSTable. It speeds up the process of partition key lookup by narrowing the pool of keys. However, because the Bloom filter is a probabilistic function, it can result in false positives. Not all SSTables identified by the Bloom filter will have data. If the Bloom filter does not rule out an SSTable, Cassandra checks the [partition key cache](http://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsConfiguringCaches.html)

The Bloom filter grows to approximately 1-2 GB per billion partitions. In the extreme case, you can have one partition per row, so you can easily have billions of these entries on a single machine. The Bloom filter is tunable if you want to trade memory for performance.

Partition Key Cache

The partition key cache, if enabled, stores a cache of the partition index in off-heap memory. The key cache uses a small, configurable amount of memory, and each "hit" saves one seek during the read operation. If a partition key is found in the key cache can go directly to the compression offset map to find the compressed block on disk that has the data. The partition key cache functions better once warmed, and can greatly improve over the performance of cold-start reads, where the key cache doesn't yet have or has purged the keys stored in the key cache. It is possible to limit the number of partition keys saved in the key cache, if memory is very limited on a node. If a partition key is not found in the key cache, then the [partition summary](http://docs.datastax.com/en/glossary/doc/glossary/gloss_index_summary.html) is searched.

The partition key cache size is configurable, as are the number of partition keys to store in the key cache.

Partition Summary

The partition summary is an off-heap in-memory structure that stores a sampling of the partition index. A partition index contains all partition keys, whereas a partition summary samples every X keys, and maps the location of every Xth key's location in the index file. For example, if the partition summary is set to sample every 20 keys, it will store the location of the first key as the beginning of the SSTable file, the 20th key and its location in the file, and so on. While not as exact as knowing the location of the partition key, the partition summary can shorten the scan to find the partition data location. After finding the range of possible partition key values, the [partition index](http://docs.datastax.com/en/glossary/doc/glossary/gloss_primary_index.html) is searched.

By configuring the sample frequency, you can trade memory for performance, as the more granularity the partition summary has, the more memory it will use. The sample frequency is changed using the [index interval](http://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlCreateTable.html#tabProp) property in the table definition. A fixed amount of memory is configurable using the [index\_summary\_capacity\_in\_mb](http://docs.datastax.com/en/cassandra/3.0/cassandra/configuration/configCassandra_yaml.html" \l "configCassandra_yaml__index_summary_capacity_in_mb) property, and defaults to 5% of the heap size.

Partition Index

The partition index resides on disk and stores an index of all partition keys mapped to their offset. If the partition summary has been checked for a range of partition keys, now the search passes to the partition index to seek the location of the desired partition key. A single seek and sequential read of the columns over the passed-in range is performed. Using the information found, the partition index now goes to the compression offset map to find the compressed block on disk that has the data. If the partition index must be searched, two seeks to disk will be required to find the desired data.

Compression offset map

The compression offset map stores pointers to the exact location on disk that the desired partition data will be found. It is stored in off-heap memory and is accessed by either the partition key cache or the partition index. The desired compressed partition data is fetched from the correct SSTable(s) once the compression offset map identifies the disk location. The query receives the result set.

**Note:** Within a partition, all rows are not equally expensive to query. The very beginning of the partition (the first rows, clustered by your key definition) is slightly less expensive to query because there is no need to consult the partition-level index.

The compression offset map grows to 1-3 GB per terabyte compressed. The more you compress data, the greater number of compressed blocks you have and the larger the compression offset table. Compression is enabled by default even though going through the compression offset map consumes CPU resources. Having compression enabled makes the page cache more effective, and typically, almost always pays off.

# How do write patterns affect reads?

It is important to consider how the write operations will affect the read operations in the cluster. The type of [compaction strategy](http://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlHowDataMaintain.html) Cassandra performs on your data is configurable and can significantly affect read performance. Using the SizeTieredCompactionStrategy or DateTieredCompactionStrategy tends to cause data fragmentation when rows are frequently updated. The LeveledCompactionStrategy (LCS) was designed to prevent fragmentation under this condition.