Apache Cassandra : Compaction

Compaction

Compaction

- **Compaction overview**
  - **Tombstones**
  - Compaction strategies:
    - **Unified Compaction Strategy (UCS)**
    - **Size-Tiered Compaction Strategy (STCS)**
    - **Leveled Compaction Strategy (LCS)**
    - **Time Window Compaction Strategy (TWCS)**

Compaction overview

Compaction overview

What is compaction?

Data in **Cassandra** is created in **memtables**. Once a memory threshold is reached, to free up memory again, the data is written to an **SSTable**, an immutable file residing on disk.

Because SSTables are immutable, when data is updated or deleted, the old data is not overwritten with inserts or updates, or removed from the SSTable. Instead, a new SSTable is created with the updated data with a new timestamp, and the old SSTable is marked for deletion. The piece of deleted data is known as a tombstone.

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.

Why must compaction be run?

Since SSTables are consulted during read operations, it is important to keep the number of SSTables small. Write operations will cause the number of SSTables to grow, so compaction is necessary. Besides the issue of tombstones, data is deleted for other reasons, too, such as Time-To-Live (TTL) expiration of some data. Deleting, updating, or expiring data are all valid triggers for compaction.

What does compaction accomplish?

Two important factors accomplished by compaction are performance improvement and disk space reclamation. If SSTables have duplicate data that must be read, read operations are slower. Once tombstones and duplicates are removed, read operations are faster. SSTables use disk space, and reducing the size of SSTables through compaction frees up disk space.

How does compaction work?

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 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.

Types of compaction

The concept of compaction is used for different kinds of operations in **Cassandra**, the common thing about these operations is that it takes one or more SSTables, merges, and outputs new SSTables. The types of compactions are:

Minor compaction

A minor compaction triggered automatically in **Cassandra** for several actions:

- When an SSTable is added to the node through flushing
- When autocompaction is enabled after being disabled (`nodetool enableautocompaction`)
- When compaction adds new SSTables
- A check for new minor compactions every 5 minutes

Major compaction

A major compaction is triggered when a user executes a compaction over all SSTables on the node.

User defined compaction

Similar to a major compaction, a user-defined compaction executes when a user triggers a compaction on a given set of SSTables.

Scrub

A scrub triggers a compaction to try to fix any broken SSTables. This can actually remove valid data if that data is corrupted. If that happens you will need to run a full repair on the node.

UpgradeSSTables

A compaction occurs when you upgrade SSTables to the latest version. Run this after upgrading to a new major version.

Cleanup

Compaction executes to remove any ranges that a node no longer owns. This type of compaction is typically triggered on neighbouring nodes after a node has been bootstrapped, since the bootstrapping node will take ownership of some ranges from those nodes.

Secondary index rebuild

A compaction is triggered if the secondary indexes are rebuilt on a node.

Anticompaction

After repair, the ranges that were actually repaired are split out of the SSTables that existed when repair started. This type of compaction rewrites SSTables to accomplish this task.

Sub range compaction

It is possible to only compact a given sub range - this action is useful if you know a token that has been misbehaving - either gathering many updates or many deletes. The command `nodetool compact`

`-st x -et y` will pick all SSTables containing the range between x and y and issue a compaction for those SSTables. For Size Tiered Compaction Strategy, this will most likely include all SSTables, but with Leveled Compaction Strategy, it can issue the compaction for a subset of the SSTables. With LCS the resulting SSTable will end up in L0.

Strategies

Different compaction strategies are available to optimize for different workloads. Picking the right compaction strategy for your workload will ensure the best performance for both querying and for compaction itself.

`Unified Compaction Strategy (UCS)`

UCS is a good choice for most workloads and is recommended for new workloads. This compaction strategy is designed to handle a wide variety of workloads. It is designed to be able to handle both immutable time-series data and workloads with lots of updates and deletes. It is also designed to be able to handle both spinning disks and SSDs.

`Size Tiered Compaction Strategy (STCS)`

STCS is the default compaction strategy, because it is useful as a fallback when other strategies don’t fit the workload. Most useful for not strictly time-series workloads with spinning disks, or when the I/O from `LCS` is too high.

`Leveled Compaction Strategy (LCS)`

Leveled Compaction Strategy (LCS) is optimized for read heavy workloads, or workloads with lots of updates and deletes. It is not a good choice for immutable time-series data.

`Time Window Compaction Strategy (TWCS)`

Time Window Compaction Strategy is designed for TTL’ed, mostly immutable time-series data.

Tombstones

What are tombstones?

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 deletion as an insertion, and inserts a time-stamped deletion marker called a tombstone. The tombstones go through **Cassandra**'s write path, and are written to SSTables on one or more nodes. The key feature difference of a tombstone is that it has a built-in expiration date/time. At the end of its expiration period, the grace period, the tombstone is deleted as part of **Cassandra**'s normal compaction process.

NOTE You can also mark a **Cassandra** row or column with a time-to-live (TTL) value. After this amount of time has ended, **Cassandra** marks the object with a tombstone, and handles it like other tombstoned objects.

Why tombstones?

The tombstone represents the deletion of an object, either a row or column value. This approach is used instead of removing values because of the distributed nature of **Cassandra**. Once an object is marked as a tombstone, queries will ignore all values that are time-stamped previous to the tombstone insertion.

Zombies

In a multi-node cluster, **Cassandra** may store replicas of the same data on two or more nodes. This helps prevent data loss, but it complicates the deletion process. If a node receives a delete command for data it stores locally, the node tombstones the specified object and tries to pass the tombstone to other nodes containing replicas of that object. 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 object. If the tombstoned object has already been deleted from the rest of the cluster before that node recovers, **Cassandra** treats the object on the recovered node as new data, and propagates it to the rest of the cluster. This kind of deleted but persistent object is called a zombie.

Grace period

To prevent the reappearance of zombies, **Cassandra** gives each tombstone a grace period. The grace period for a tombstone is set with the table property ` WITH gc_grace_seconds`. Its default value is 864000 seconds (ten days), after which a tombstone expires and can be deleted during compaction. Prior to the grace period expiring, **Cassandra** will retain a tombstone through compaction events. Each table can have its own value for this property.

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 object during the grace period, **Cassandra** overwrites the tombstone. If a client sends a read for that object during the grace period, **Cassandra** disregards the tombstone and retrieves the object from other replicas if possible.

When an unresponsive node recovers, **Cassandra** uses hinted handoff to replay the database mutations the node missed while it was down. **Cassandra** does not replay a mutation for a tombstoned object 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.

Deletion

After `gc_grace_seconds` has expired the tombstone may be removed (meaning there will no longer be any object that a certain piece of data was deleted). But one complication for deletion is that a tombstone can live in one SSTable and the data it marks for deletion in another, so a compaction must also remove both SSTables. More precisely, drop an actual tombstone the:

- The tombstone must be older than `gc_grace_seconds`. Note that tombstones will not be removed until a compaction event even if `gc_grace_seconds` has elapsed.
- If partition X contains the tombstone, the SSTable containing the partition plus all SSTables containing data older than the tombstone containing X must be included in the same compaction. If all data in any SSTable containing partition X is newer than the tombstone, it can be ignored.
- If the option `only_purge_repaired_tombstones` is enabled, tombstones are only removed if the data has also been repaired. This process is described in the "Deletes with tombstones" sections.

If a node remains down or disconnected for longer than `gc_grace_seconds`, its deleted data will be repaired back to the other nodes and reappear in the cluster. This is basically the same as in the "Deletes without Tombstones" section.

Deletes without tombstones

Imagine a three node cluster which has the value [A] replicated to every node.:

```
[A], [A], [A]
```

If one of the nodes fails and and our delete operation only removes existing values, we can end up with a cluster that looks like:

```
[], [], [A]
```

Then a repair operation would replace the value of [A] back onto the two nodes which are missing the value.:

```
[A], [A], [A]
```

This would cause our data to be resurrected as a zombie even though it had been deleted.

Deletes with tombstones

Starting again with a three node cluster which has the value [A] replicated to every node.:

```
[A], [A], [A]
```

If instead of removing data we add a tombstone object, so the single node failure situation will look like:

```
[A, Tombstone[A]], [A, Tombstone[A]], [A]
```

Now when we issue a repair the tombstone will be copied to the replica, rather than the deleted data being resurrected:

```
[A, Tombstone[A]], [A, Tombstone[A]], [A, Tombstone[A]]
```

Our repair operation will correctly put the state of the system to what we expect with the object [A] marked as deleted on all nodes. This does mean we will end up accruing tombstones which will permanently accumulate disk space. To avoid keeping tombstones forever, we set `gc_grace_seconds` for every table in **Cassandra**.

Fully expired SSTables

If an SSTable contains only tombstones and it is guaranteed that SSTable is not shadowing data in any other SSTable, then the compaction can drop that SSTable. If you see SSTables with only tombstones (note that TTL'd data is considered tombstones once the time-to-live has expired), but it is not being dropped by compaction, it is likely that other SSTables contain older data. There is a tool called `sstableexpiredblockers` that will list which SSTables are droppable and which are blocking them from being dropped. With `TimeWindowCompactionStrategy` it is possible to remove the guarantee (not check for shadowing data) by enabling `unsafe_aggressive_sstable_expiration`.

TTL

Data in Cassandra can have an additional property called time to live - this is used to automatically drop data that has expired once the time is reached. Once the TTL has expired the data is converted to a tombstone which stays around for at least `gc_grace_seconds`. Note that if you mix data with TTL and data without TTL (or just different length of the TTL) Cassandra will have a hard time dropping the tombstones created since the partition might span many SSTables and not all are compacted at once.

Fully expired SSTables

If an SSTable contains only tombstones and it is guaranteed that SSTable is not shadowing data in any other SSTable, then the compaction can drop that SSTable. If you see SSTables with only tombstones (note that TTL-ed data is considered tombstones once the time-to-live has expired), but it is not being dropped by compaction, it is likely that other SSTables contain older data. There is a tool called `sstableexpiredblockers` that will list which SSTables are droppable and which are blocking them from being dropped. With `TimeWindowCompactionStrategy` it is possible to remove the guarantee (not check for shadowing data) by enabling `unsafe_aggressive_sstable_expiration`.

Repaired/unrepaired data

With incremental repairs Cassandra must keep track of what data is repaired and what data is unrepaired. With anticompaction repaired data is split out into repaired and unrepaired SSTables. To avoid mixing up the data again separate compaction strategy instances are run on the two sets of data, each instance only knowing about either the repaired or the unrepaired SSTables. This means that if you only run incremental repair once and then never again, you might have very old data in the repaired SSTables that block compaction from dropping tombstones in the unrepaired (probably newer) SSTables.

Data directories

Since tombstones and data can live in different SSTables it is important to realize that losing an SSTable might lead to data becoming live again - the most common way of losing SSTables is to have a hard drive break down. To avoid making data live tombstones and actual data are always in the same data directory. This way, if a disk is lost, all versions of a partition are lost and no data can get undeleted. To achieve this a compaction strategy instance per data directory is run in addition to the compaction strategy instances containing repaired/unrepaired data, this means that if you have 4 data directories there will be 8 compaction strategy instances running. This has a few more benefits than just avoiding data getting undeleted:

- It is possible to run more compactions in parallel - leveled compaction will have several totally separate levelings and each one can run compactions independently from the others.
- Users can backup and restore a single data directory.
- Note though that currently all data directories are considered equal, so if you have a tiny disk and a big disk backing two data directories, the big one will be limited the by the small one. One work around to this is to create more data directories backed by the big disk.

Single SSTable tombstone compaction

When an SSTable is written a histogram with the tombstone expiry times is created and this is used to try to find SSTables with very many tombstones and run single SSTable compaction on that SSTable in hope of being able to drop tombstones in that SSTable. Before starting this it is also checked how likely it is that any tombstones will actually will be able to be dropped how much this SSTable overlaps with other SSTables. To avoid most of these checks the compaction option `unchecked_tombstone_compaction` can be enabled.

Common options

There is a number of common options for all the compaction strategies;

`enabled` **(default: true)**

Whether minor compactions should run. Note that you can have 'enabled': true as a compaction option and then do 'nodetool enableautocompaction' to start running compactions.

`tombstone_threshold` **(default: 0.2)**

How much of the SSTable should be tombstones for us to consider doing a single SSTable compaction of that SSTable.

`tombstone_compaction_interval` **(default: 86400s (1 day))**

Since it might not be possible to drop any tombstones when doing a single SSTable compaction we need to make sure that one SSTable is not constantly getting recompacted - this option states how often we should try for a given SSTable.

`log_all` **(default: false)**

New detailed compaction logging, see `below <detailed-compaction-logging>`.

`unchecked_tombstone_compaction` **(default: false)**

The single SSTable compaction has quite strict checks for whether it should be started, this option disables those checks and for some use cases this might be needed. Note that this does not change anything for the actual compaction, tombstones are only dropped if it is safe to do so - it might just rewrite an SSTable without being able to drop any tombstones.

`only_purge_repaired_tombstone` **(default: false)**

Option to enable the extra safety of making sure that tombstones are only dropped if the data has been repaired.

`min_threshold` **(default: 4)**

Lower limit of number of SSTables before a compaction is triggered. Not used for `LeveledCompactionStrategy`.

`max_threshold` **(default: 32)**

Upper limit of number of SSTables before a compaction is triggered. Not used for `LeveledCompactionStrategy`.

Further, see the section on each strategy for specific additional options.

Compaction nodetool commands

The `nodetool <nodetool>` utility provides a number of commands related to compaction:

`enableautocompaction`
: Enable compaction.

`disableautocompaction`
: Disable compaction.

`setcompactionthroughput`
: How fast compaction should run at most - defaults to 64MiB/s.

`compactionstats`
: Statistics about current and pending compactions.

`compactionhistory`
: List details about the last compactions.

`setcompactionthreshold`
: Set the min/max SSTable count for when to trigger compaction, defaults to 4/32.

Switching the compaction strategy and options using JMX

It is possible to switch compaction strategies and its options on just a single node using JMX, this is a great way to experiment with settings without affecting the whole cluster. The mbean is:

```
org.apache.cassandra.db:type=ColumnFamilies,keyspace=<keyspace_name>,columnfamily=<table_
name>
```

and the attribute to change is `CompactionParameters` or `CompactionParametersJson` if you use jconsole or jmc. For example, the syntax for the json version is the same as you would use in an `ALTER TABLE <alter-table-statement>` statement:

```
{ 'class': 'LeveledCompactionStrategy', 'sstable_size_in_mb': 123, 'fanout_size': 10}
```

The setting is kept until someone executes an `ALTER TABLE <alter-table-statement>` that touches the compaction settings or restarts the node.

More detailed compaction logging

Enable with the compaction option `log_all` and a more detailed compaction log file will be produced in your log directory.

Tombstones

Tombstones

What are tombstones?

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 deletion as an insertion, and inserts a time-stamped deletion marker called a tombstone. The tombstones go through **Cassandra**'s write path, and are written to SSTables on one or more nodes. The key feature difference of a tombstone is that it has a built-in expiration date/time. At the end of its expiration period, the grace period, the tombstone is deleted as part of **Cassandra**'s normal compaction process.

NOTE You can also mark a **Cassandra** row or column with a time-to-live (TTL) value. After this amount of time has ended, **Cassandra** marks the object with a tombstone, and handles it like other tombstoned objects.

Why tombstones?

The tombstone represents the deletion of an object, either a row or column value. This approach is used instead of removing values because of the distributed nature of **Cassandra**. Once an object is marked as a tombstone, queries will ignore all values that are time-stamped previous to the tombstone insertion.

Zombies

In a multi-node cluster, **Cassandra** may store replicas of the same data on two or more nodes. This helps prevent data loss, but it complicates the deletion process. If a node receives a delete command for data it stores locally, the node tombstones the specified object and tries to pass the tombstone to other nodes containing replicas of that object. 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 object. If the tombstoned object has already been deleted from the rest of the cluster before that node recovers, **Cassandra** treats the object on the recovered node as new data, and propagates it to the rest of the cluster. This kind of deleted but persistent object is called a zombie.

Grace period

To prevent the reappearance of zombies, **Cassandra** gives each tombstone a grace period. The grace period for a tombstone is set with the table property ` WITH gc_grace_seconds`. Its default value is 864000 seconds (ten days), after which a tombstone expires and can be deleted during compaction. Prior to the grace period expiring, **Cassandra** will retain a tombstone through compaction events.

Each table can have its own value for this property.

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 object during the grace period, **Cassandra** overwrites the tombstone. If a client sends a read for that object during the grace period, **Cassandra** disregards the tombstone and retrieves the object from other replicas if possible.

When an unresponsive node recovers, **Cassandra** uses hinted handoff to replay the database mutations the node missed while it was down. **Cassandra** does not replay a mutation for a tombstoned object 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.

Deletion

After `gc_grace_seconds` has expired the tombstone may be removed (meaning there will no longer be any object that a certain piece of data was deleted). But one complication for deletion is that a tombstone can live in one SSTable and the data it marks for deletion in another, so a compaction must also remove both SSTables. More precisely, drop an actual tombstone the:

- The tombstone must be older than `gc_grace_seconds`. Note that tombstones will not be removed until a compaction event even if `gc_grace_seconds` has elapsed.
- If partition X contains the tombstone, the SSTable containing the partition plus all SSTables containing data older than the tombstone containing X must be included in the same compaction. If all data in any SSTable containing partition X is newer than the tombstone, it can be ignored.
- If the option `only_purge_repaired_tombstones` is enabled, tombstones are only removed if the data has also been repaired. This process is described in the "Deletes with tombstones" sections.

If a node remains down or disconnected for longer than `gc_grace_seconds`, its deleted data will be repaired back to the other nodes and reappear in the cluster. This is basically the same as in the "Deletes without Tombstones" section.

Deletes without tombstones

Imagine a three node cluster which has the value [A] replicated to every node.:

```
[A], [A], [A]
```

If one of the nodes fails and and our delete operation only removes existing values, we can end up with a cluster that looks like:

```
[], [], [A]
```

Then a repair operation would replace the value of [A] back onto the two nodes which are missing the value.:

```
[A], [A], [A]
```

This would cause our data to be resurrected as a zombie even though it had been deleted.

Deletes with tombstones

Starting again with a three node cluster which has the value [A] replicated to every node.:

```
[A], [A], [A]
```

If instead of removing data we add a tombstone object, so the single node failure situation will look like:

```
[A, Tombstone[A]], [A, Tombstone[A]], [A]
```

Now when we issue a repair the tombstone will be copied to the replica, rather than the deleted data being resurrected:

```
[A, Tombstone[A]], [A, Tombstone[A]], [A, Tombstone[A]]
```

Our repair operation will correctly put the state of the system to what we expect with the object [A] marked as deleted on all nodes. This does mean we will end up accruing tombstones which will permanently accumulate disk space. To avoid keeping tombstones forever, we set `gc_grace_seconds` for every table in **Cassandra**.

Fully expired SSTables

If an SSTable contains only tombstones and it is guaranteed that SSTable is not shadowing data in any other SSTable, then the compaction can drop that SSTable. If you see SSTables with only tombstones (note that TTL'd data is considered tombstones once the time-to-live has expired), but it is not being dropped by compaction, it is likely that other SSTables contain older data. There is a tool called `sstableexpiredblockers` that will list which SSTables are droppable and which are blocking them from being dropped. With `TimeWindowCompactionStrategy` it is possible to remove the guarantee (not check for shadowing data) by enabling `unsafe_aggressive_sstable_expiration`.

Unified Compaction Strategy (UCS)

Unified Compaction Strategy (UCS)

The `UnifiedCompactionStrategy (UCS)` is recommended for most workloads, whether read-heavy, write-heavy, mixed read-write, or time-series. There is no need to use legacy compaction strategies, because UCS can be configured to behave like any of them.

UCS is a compaction strategy that combines the best of the other strategies plus new features. UCS has been designed to maximize the speed of compactions, which is crucial for high-density nodes, using an unique sharding mechanism that compacts partitioned data in parallel. And whereas STCS, LCS, or TWCS will require full compaction of the data if the compaction strategy is changed, UCS can change parameters in flight to switch from one strategy to another. In fact, a combination of different compaction strategies can be used at the same time, with different parameters for each level of the hierarchy. Finally, UCS is stateless, so it does not rely on any metadata to make compaction decisions.

Two key concepts refine the definition of the grouping:

- Tiered and levelled compaction can be generalized as equivalent, because both create exponentially-growing levels based on the **size** of SSTables (or non-overlapping SSTable runs). Thus, a compaction is triggered when more than a given number of SSTables are present on one level.
- **size** can be replaced by **density**, allowing SSTables to be split at arbitrary points when the output of a compaction is written, while still producing a leveled hierarchy. Density is defined as the size of an SSTable divided by the width of the token range it covers.

Let's look at the first concept in more detail.

Read and write amplification

UCS can adjust the balance between the number of SSTables consulted to serve a read (read amplification, or RA) versus the number of times a piece of data must be rewritten during its lifetime (write amplification, or WA). A single configurable scaling parameter determines how the compaction behaves, from a read-heavy mode to a write-heavy mode. The scaling parameter can be changed at any time, and the compaction strategy will adjust accordingly. For example, an operator may decide to:

- decrease the scaling parameter, lowering the read amplification at the expense of more complex writes when a certain table is read-heavy and could benefit from reduced latencies
- increase the scaling parameter, reducing the write amplification, when it has been identified that compaction cannot keep up with the number of writes to a table

Any such change only initiates compactions that are necessary to bring the hierarchy in a state compatible with the new configuration. Any additional work already done (for example, when switching from negative parameter to positive), is advantageous and incorporated.

In addition, by setting the scaling parameters to simulated a high tiered fanout factor, UCS can accomplish the same compaction as TWCS.

[ucs]

UCS uses a combination of tiered and leveled compaction, plus sharding, to achieve the desired read and write amplification. SSTables are sorted by token range, and then grouped into levels:

[sharding]

UCS groups SSTables in levels based on the logarithm of the SSTables density, with the fanout factor `f` as the base of the logarithm, and with each level triggering a compaction as soon as it has `t` overlapping SSTables.

The choice of the scaling parameter `w`, determines the value of the fanout factor `f`. In turn, `w` defines the mode as either leveled or tiered, and `t` is set as either `t=2` for leveled compaction, or `t=f` for tiered compaction. One last parameter, the minimum SSTable size, is required to determine the complete behaviour of UCS.

Three compaction types are possible, based on the value of `w`, as shown in the diagram of RA and WA above:

- Leveled compaction, with high WA, low RA: `w < 0`, `f = 2 - w` and `t=2` `L_{f}` is the shorthand for specifying this range of `w` values (e.g., L10 for `w=-8`).
- Tiered compaction, with low WA, high RA: `w > 0`, `f = 2 + w` and `t=f`. `T_{f}` is the shorthand (e.g., T4 for `w = 2`).
- Leveled and tiered compaction behave identically in the center: `w = 0` and `f = t = 2`. In shorthand, N for `w = 0`.

NOTE Leveled compaction improves reads at the expense of writes and approaches a sorted array as `f` increases, whereas tiered compaction favors writes at the expense of reads and approaches an unsorted log as `f` increases.

UCS permits the value of `w` to be defined separately for each level; thus, levels can have different behaviours. For example, level zero can use tiered compaction (STCS-like), while higher levels can be leveled (LCS-like), defined with increasing levels of read optimization.

Size-based leveling

The strategy splits SSTables at specific shard boundaries whose number grows with the density of an SSTable. The non-overlap between SSTables created by the splitting makes concurrent compactions possible. However, let's ignore density and splitting for a moment and explore how SSTables are grouped into levels if they are never split.

Memtables are flushed to level zero (L0), and the memtable flush size `s_{f}` is calculated as the average size of all the SSTables written when a memtable is flushed. This parameter, `s_{f}`, is intended to form the basis of the hierarchy where all newly-flushed SSTables end up. Using a fixed fanout factor `f` and `s_{f}`, the level `L` for an SSTable of size `s` is calculated as follows:

```
L =
\begin{cases}
\left \lfloor \log_f {\frac{s}{s_{f}}} \right \rfloor &amp; \text{if } s \ge s_{f} \\
0 &amp; \text{otherwise}
\end{cases}
```

SSTables are assigned to levels based on their size:

Level	Min SSTable size	Max SSTable size
0	0	`s_{f} \cdot f`
1	`s_{f} \cdot f`	`s_{f} \cdot f^2`
2	`s_{f} \cdot f^2`	`s_{f} \cdot f^3`
3	`s_{f} \cdot f^3`	`s_{f} \cdot f^4`
…	…	…
n	`s_{f} \cdot f^n`	`s_{f} \cdot f^{n+1}`

Once SSTables start accumulating in levels, compaction is triggered when the number of SSTables in a level exceeds a threshold `t` discussed earlier:

- `t = 2`, leveled compaction:
    a. An SSTable is promoted to level `n` with size `\ge s_{f} \cdot f^n`.
    b. When a second SSTable is promoted to that level (also with size `\ge s_{f} \cdot f^n`) they compact and form a new SSTable of size `\sim 2s_{f} \cdot f^n` in the same level for `f > 2`.
    c. After this repeats at least `f-2` more times (i.e., `f` total SSTables enter the level), the compaction result grows to `\ge s_{f} \cdot f^{n+1}` and enters the next level.
- `t = f`, tiered compaction:
    - After `f` SSTables enter level `n`, each of size `\ge s{f} \cdot f^n`, they are compacted and form a new SSTable of size `\ge s_{f} \cdot f^{n+1}` in the next level.

Overwrites and deletions are ignored in these schemes, but if an expected proportion of overwrites/deletions are known, the algorithm can be adjusted. The current UCS implementation does this adjustment, but doesn't expose the adjustment at this time.

Estimate of RA and WA

The read and write amplification for each compaction strategy are proportional to the number of levels in the hierarchy:

Strategy	Write amplification	Read amplification
Leveled	`\varpropto` `f-1` times the number of levels	`\varpropto` the number of levels
Tiered	`\varpropto` the number of levels	`\varpropto` `f-1` times the number of levels

Number of levels

Using the maximal dataset size `D`, the number of levels can be calculated as follows:

```
L =
\begin{cases}
\left \lfloor \log_f {\frac {D}{s_{f}}} \right \rfloor &amp; \text{if } D \ge s_{f} \\
0 &amp; \text{otherwise}
\end{cases}
```

This calculation is based on the assumption that the maximal dataset size `D` is reached when all levels are full, and the maximal number of levels is inversely proportional to the logarithm of `f`.

Thus, when we try to control the overheads of compaction on the database, we have a space of choices for the strategy that range from:

- leveled compaction (`t=2`) with high `f`:
  - low number of levels
  - high read efficiency
  - high write cost
  - moving closer to the behaviour of a sorted array as `f` increases
- compaction with `t = f = 2` where leveled is the same as tiered and we have a middle ground with logarithmically increasing read and write costs;
- tiered compaction (`t=f`) with high `f`:
  - very high number of SSTables
  - low read efficiency
  - low write cost

    - moving closer to an unsorted log as f increases

This can be easily generalized to varying fan factors, by replacing the exponentiation with the product of the fan factors for all lower levels:

Level	**Min SSTable size**	**Max SSTable size**
0	0	$s_{f} \cdot f_0$
1	$s_{f} \cdot f_0$	$s_{f} \cdot f_0 \cdot f_1$
2	$s_{f} \cdot f_0 \cdot f_1$	$s_{f} \cdot f_0 \cdot f_1 \cdot f_2$
…	…	…
n	$s_{f} \cdot \prod_{i < n} f_i$	$s_{f} \cdot \prod_{i\le n} f_i$

Density-based leveling

If we replace the size s in the previous discussion with the density measure

$$d = \frac s v$$

where v is the fraction of the token space that the SSTable covers, all formulae and conclusions remain valid. However, using density, the output can now be split at arbitrary points. If several SSTables are compacted and split, the new resulting SSTables formed will be denser than the original SSTables. For example, using a scaling parameter of T4, four input SSTables spanning 1/10 of the token space each, when compacted and split, will form four new SSTables spanning 1/40 of the token space each.

These new SSTables will be the same size but denser, and consequently will be moved to the next higher level, due to the higher density value exceeding the maximum density for the original compacted level. If we can ensure that the split points are fixed (see below), this process will repeat for each shard (token range), executing independent compactions concurrently.

IMPORTANT

It is important to account for locally-owned token share when calculating v. Because vnodes mean that the local token ownership of a node is not contiguous, the difference between the first and last token is not sufficient to calculate token share; thus, any non-locally-owned ranges must be excluded.

Using the density measure allows us to control the size of SSTables through sharding, as well as to execute compactions in parallel. With size-leveled compaction, we could achieve parallelization by pre-splitting the data in a fixed number of compaction shards, based on the data directories. However, that method requires the number of shards to be predetermined and equal for all levels of the hierarchy, and SSTables can become too small or too large. Large SSTables complicate streaming and repair and increase the duration of compaction operations, pinning resources to long-running operations and making it more likely that too many SSTables will accumulate on lower levels of the hierarchy.

Density-leveled compaction permits a much wider variety of splitting options. For instance, the size of SSTables can be kept close to a selected target, allowing UCS to deal with the leveling of both STCS (SSTable size grows with each level) and LCS (token share shrinks with each level).

Sharding

Basic sharding scheme

This sharding mechanism is independent of the compaction specification. There are a range of choices for splitting SSTables:

- Split when a certain output size is reached (like LCS), forming non-overlapping SSTable runs instead of individual SSTables
- Split the token space into shards at predefined boundary points
- Split at predefined boundaries, but only if a certain minimum size has been reached

Splitting only by size results in individual SSTables with start positions that vary. To compact SSTables split in this way, you must choose to either compact the entire token range of a level sequentially or compact and copy some of the data more times than necessary, due to overlapping SSTables. If predefined boundary points are used, some of the token range can be sparser with fewer inputs and skew the density of the resulting SSTables. If that occurs, further splitting may be required. In the hybrid option, the density skew can occur less frequently, but can still occur.

To avoid these problems and permit concurrent compactions of all levels of the compaction hierarchy, UCS predefines boundary points for every compaction and always splits SSTables at these points. The number of boundaries is determined from the density of the input SSTables and the estimated density of the resulting SSTables. As the density grows larger, the number of boundaries is increased, keeping the size of individual SSTables close to a predefined target. Using power-of-two multiples of a specified base count, i.e., splitting shards in the middle, ensures that any boundary that applies to a given output density also applies to all higher densities.

Two sharding parameters can be configured:

- base shard count `b`
- target SSTable size `s_{t}`

At the start of every compaction, recall that the density of the output `d` is estimated, based on the input size `s` of the SSTables and the token range `v`:

```
d = \frac s v
```

where `v` is the fraction of the token range covered by the input SSTables, in a value of 0 to 1. `v = 1` means that the entire token range is covered by the input SSTables, and `v = 0` means that the input

SSTables cover no token range.

When the initial flush of the memtable to L0 occurs, `v = 1` since the entire token range is included in the memtable. In subsequent compactions, the token range `v` is the fraction of the token range covered by the SSTables being compacted.

With the calculated density of the output, plus the values of `b` and `s_{t}`, the number of shards `S` into which to split the token space can be calculated:

```
S =
\begin{cases}
b
  &amp; \text{if } {\frac d s_{t} \cdot \frac 1 b} &lt; 1 \\
2^{\left\lfloor \log_2 \left( {\frac d s_{t} \cdot \frac 1 b}\right)\right\rceil} \cdot b
  &amp; \text{if } {\frac d s_{t} \cdot \frac 1 b} \ge 1 \\
\end{cases}
```

where `\lfloor x \rceil` stands for `x` rounded to the nearest integer, i.e. `\lfloor x + 0.5 \rfloor`. Thus, in the second case,the density is divided by the target size and rounded to a power-of-two multiple of `b`. If the result is less than 1, the number of shards will be the base shard count, because the memtable is split into `{2 \cdot b}`, or `b` L0 shards.

However, the token range is not the only factor that influences if we switch between `b` shards or more (where the condition is greater than or equal to 1). If the memtable is very large and able to flush several gigabytes at once, `d` may be a magnitude larger than `s_{t}`, and cause SSTables to be split into multiple shards even on L0. Conversely, if the memtable is small, `d` may still be smaller than `s_{t}` on levels above L0, where the condition is less than 1, and thus, there will be stem[b] shards.

`S - 1` boundaries are generated, splitting the local token space equally into `S` shards. Splitting the local token space will split the result of the compaction on these boundaries to form a separate SSTable for each shard. SSTables produced will have sizes that fall between `s_{t}/\sqrt 2` and `s_{t} \cdot \sqrt 2`.

For example, let's use a target SSTable size of `s_{t} = 100MiB` and `b = 4` base shards. If a `s_{f} = 200 MiB` input memtable is flushed, the condition for calculating the number of shards is:

```
\frac{200}{100} \cdot \frac{1}{4} = 0.5 &lt; 1
```

This calculation results in `0.5 < 1`, because the value of `v = 1` on the initial flush. Because the result is less than 1, the base shard count is used, and the memtable is split into four L0 shards of approximately 50MiB each. Each shard spans 1/4 of the token space.

To continue the example, on the next level of compaction, for just one of the four shards, let's compact six of these 50 MiB SSTables. The estimated density of the output will be:

```
\left( \frac{6 \cdot 50\, \mathrm{MiB}}{\frac{1}{4}} \right) = 1200 \mathrm{MiB}
```

using 1/4 as the `v` value for the token range covered by the input SSTables.

The condition for splitting will be:

```
(\frac{1200}{100} \cdot \frac{1}{4}) = 3 &gt; 1
```

Thus, the number of shards will be calculated as:

```
2^{\left\lfloor \log_2 \left( {\frac{1200}{100} \cdot \frac{1}{4}}\right)\right\rceil}
\cdot b
```

or `2^{\log_2 3}`, rounded to `2^2 \cdot 4` shards for the whole local token space, and that compaction covering 1/4 of the token space. Assuming no overwrites or deletions, the resulting SSTables will be of size 75 MiB, token share 1/16 and density 1200 MiB.

Full sharding scheme

This sharding scheme can be easily extended. There are two extensions currently implemented, SSTable growth and a minimum SSTable size.

First, let's examine the case when the size of the data set is expected to grow very large. To avoid pre-specifying a sufficiently large target size to avoid problems with per-SSTable overhead, an `SSTtable growth` parameter has been implemented. This parameter determines what part of the density growth should be assigned to increased SSTable size, reducing the growth of the number of shards, and hence, non-overlapping SSTables.

The second extension is a mode of operation with a fixed number of shards that splits conditionally on reaching a minimum size. Defining a `minimum SSTable size`, the base shard count can be reduced whenever a split would result in SSTables smaller than the provided minimum.

There are four user-defined sharding parameters:

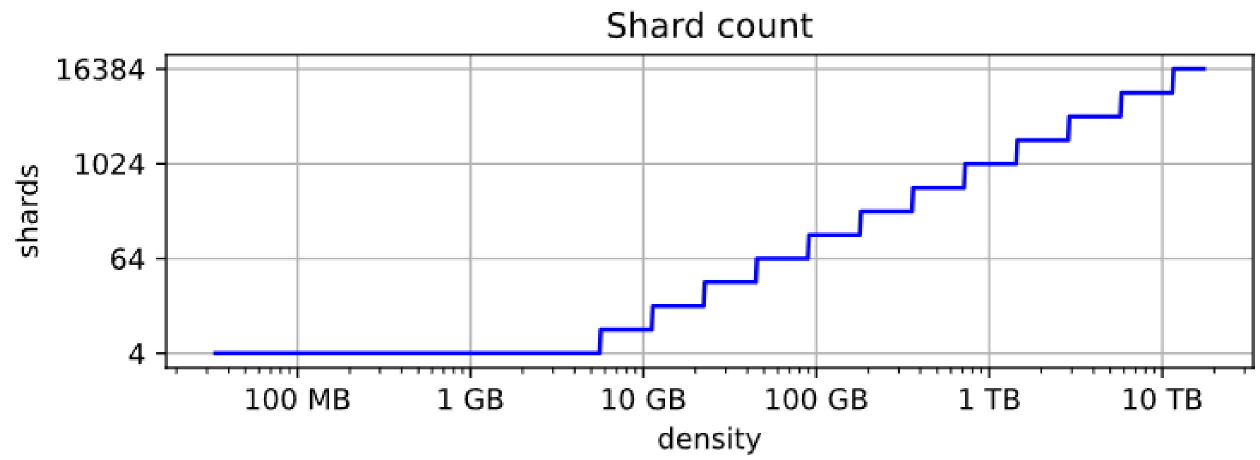
- base shard count `b`
- target SSTable size `s_{t}`
- minimum SSTable size `s_{m}`
- SSTable growth component `\lambda`

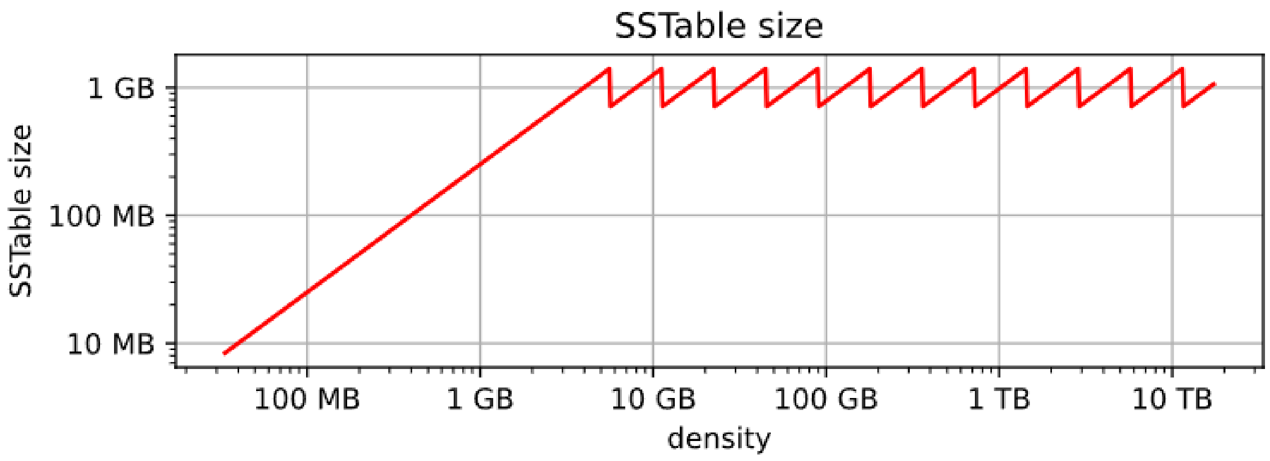
The number of shards `S` for a given density `d` is then calculated as

```
S =
\begin{cases}
1
    &amp; \text{if } d &lt; s_{m} \\
\min(2^{\left\lfloor \log_2 \frac {d}{s_{m}}  \right\rfloor}, x)
    &amp; \text{if } d &lt; s_{m} \cdot b \text{, where } x \text{ is the largest power
of 2 divisor of } b \\
b
    &amp; \text{if } d &lt; s_{t} \cdot b \\
2^{\left\lfloor (1-\lambda) \cdot \log_2 \left( {\frac {d}{s_{t}} \cdot \frac 1
b}\right)\right\rceil} \cdot b
    &amp; \text{otherwise}
\end{cases}
```

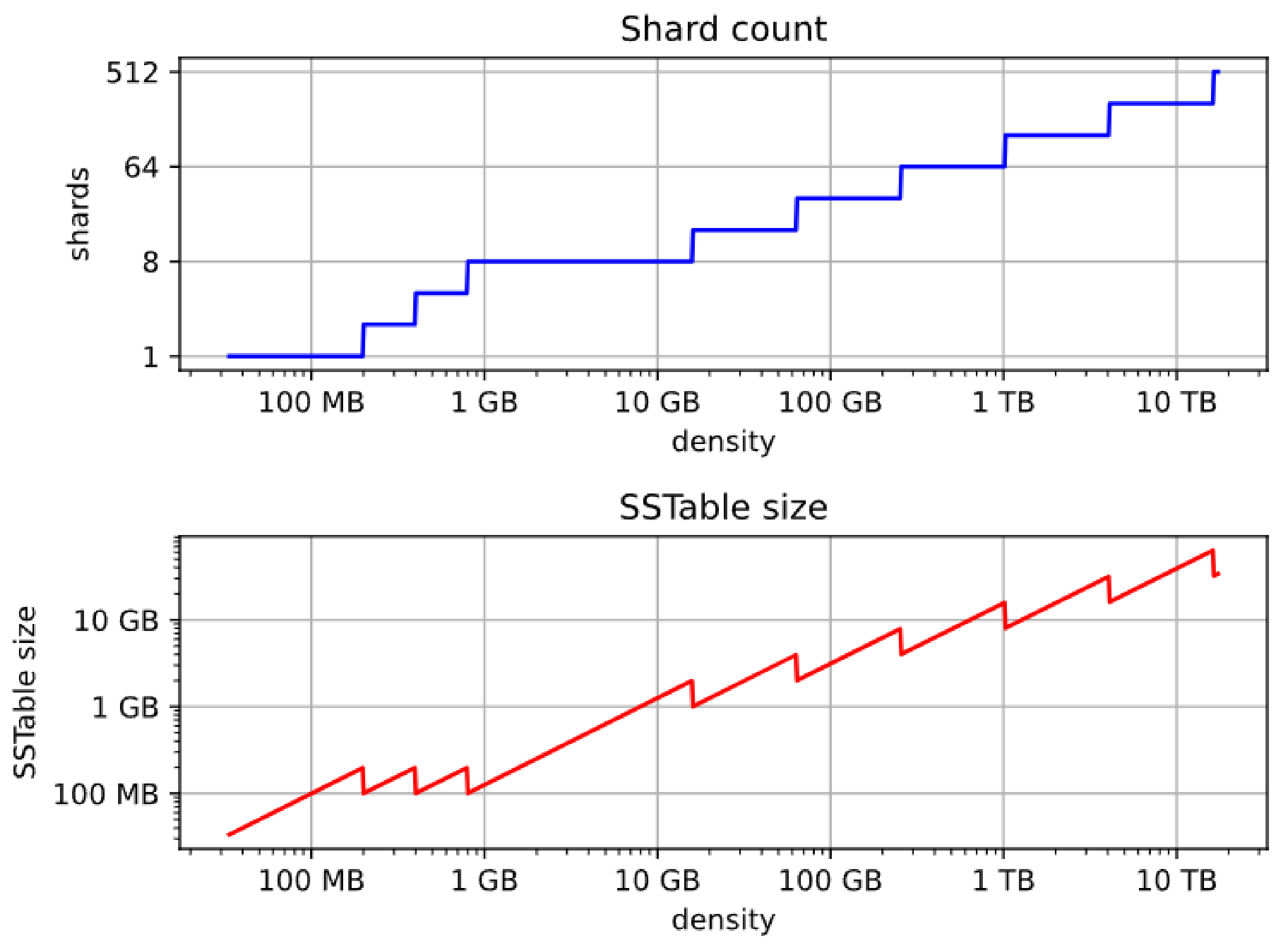
Some useful combinations of these parameters:

- The basic scheme above uses a SSTable growth `\lambda=0`, and a minimum SSTable size `s_{m}=0`. The graph below illustrates the behaviour for base shard count `b=4` and target SSTable size `s_{t} = 1\, \mathrm{GB}`:

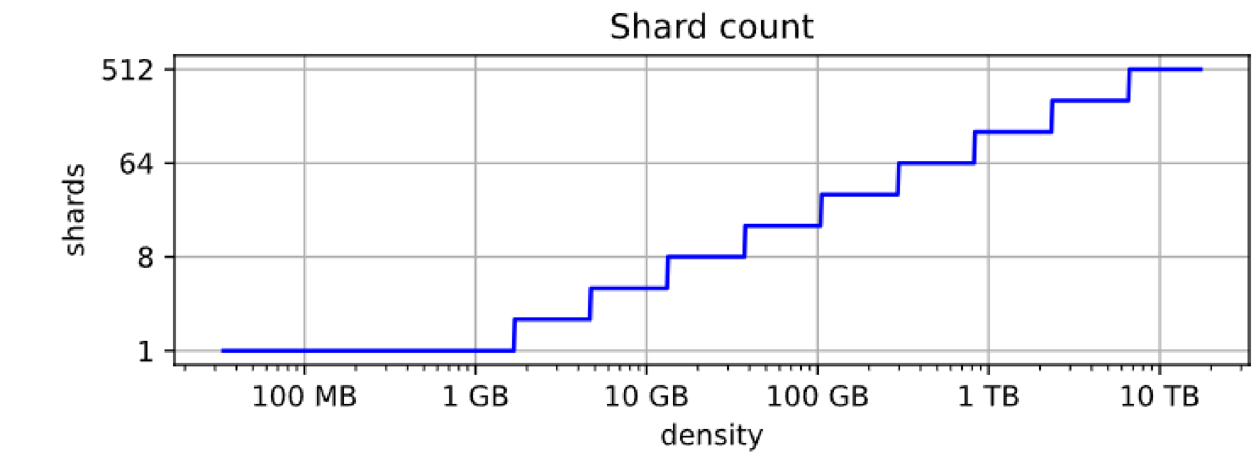


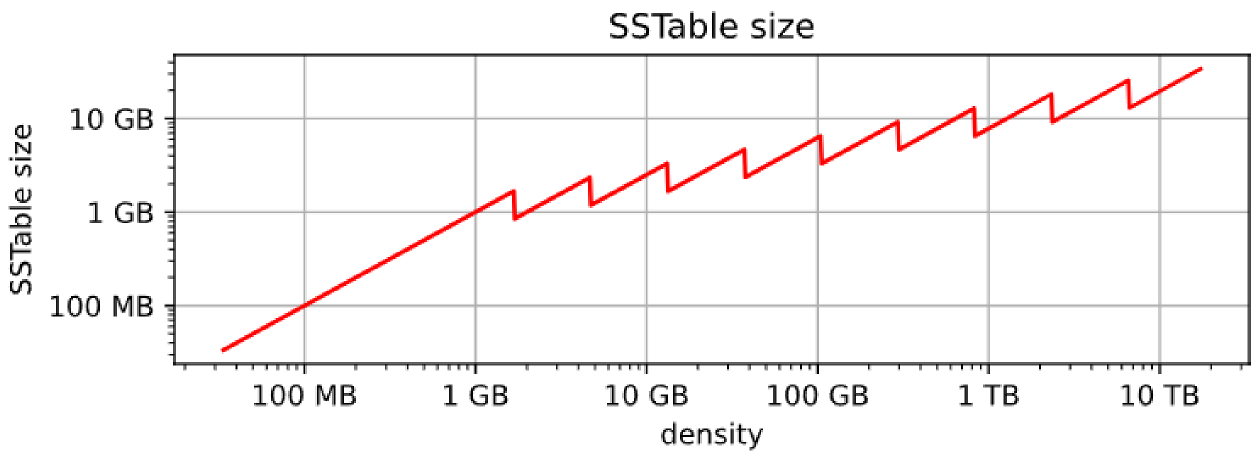


- Using `\lambda = 0.5` grows the shard count and SSTable size evenly. When the density quadruples, both the shard count and the expected SSTable size for that density band will double. The example below uses `b=8`, `s_{t} = 1\, \mathrm{GB}` and also applies a minimal size `m = 100\, \mathrm{MB}`:

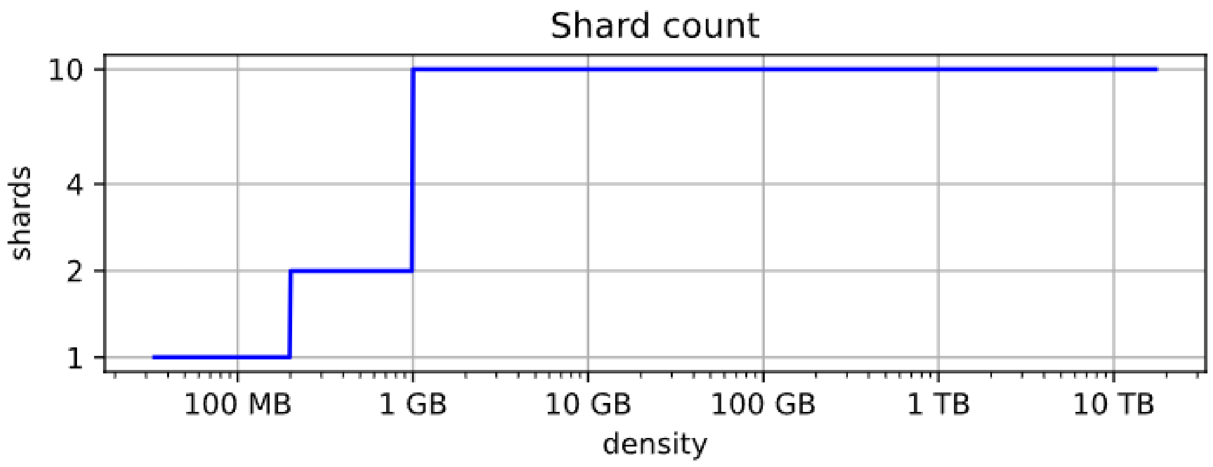


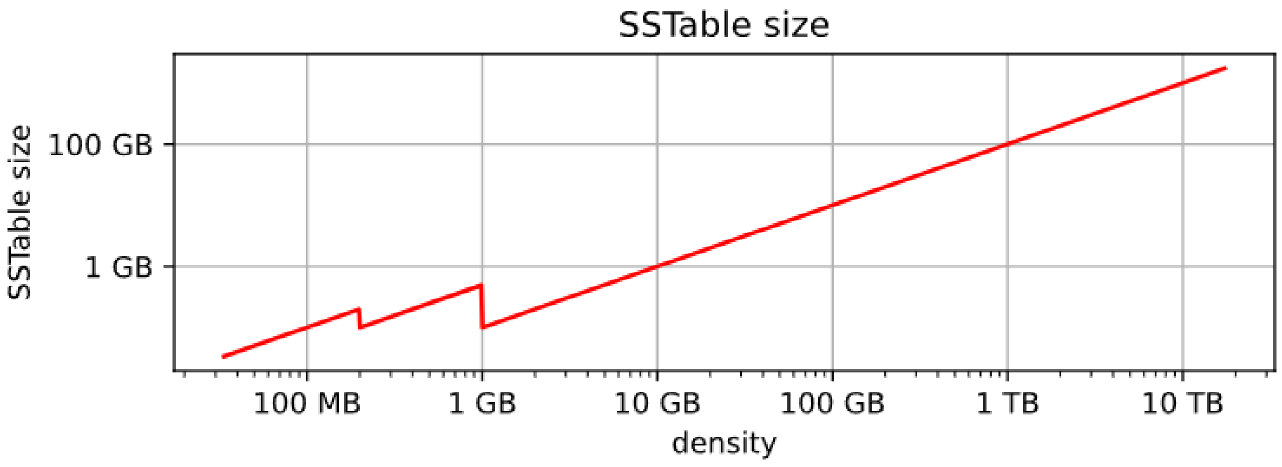
- Similarly, `\lambda = 1/3` makes the the SSTable growth the cubic root of the density growth, i.e. the SSTable size grows with the square root of the growth of the shard count. The graph below uses `b=1` and `s_{t} = 1\, \mathrm{GB}` (note: when `b=1`, the minimal size has no effect):





- A growth component of 1 constructs a hierarchy with exactly `b` shards at every level. Combined with a minimum SSTable size, the mode of operation uses a pre-specified number of shards, but splits only after reaching a minimum size. Illustrated below for `b=10` and `s_{m} = 100\, \mathrm{MB}` (note: the target SSTable size is irrelevant when `\lambda=1`):





Choosing SSTables to compact

The density leveling separates SSTables in levels defined by the compaction configuration's fan factors. However, unlike in size leveling, where SSTables are expected to cover the full token space, the number of SSTables on a level cannot be used as a trigger due to possible non-overlapping SSTables. Read queries are less efficient in this situation. To deal with this, execute sharding that performs multiple compactions on a level concurrently, and reduces the size of individual compaction operations. The non-overlapping section must be separated into different buckets, and the number of overlapping SSTables in a bucket determines what to do. A bucket is the selected set of SSTables that will be compacted together.

First form a minimal list of overlap sets that satisfy the following requirements:

- two SSTables that do not overlap are never put in the same set
- if two SSTables overlap, there is a set in the list that contains both
- SSTables are placed in consecutive positions in the list

The second condition can also be rephrased to say that for any point in the token range, there is a set in the list that contains all SSTables whose range covers that point. In other words, the overlap sets give us the maximum number of SSTables that need to be consulted to read any key, i.e., the read amplification that our trigger `t` aims to control. We don't calculate or store the exact spans the overlapping sets cover, only the participating SSTables. The sets can be obtained in `O(n\log n)` time.

For example, if SSTables A, B, C and D cover, respectively, tokens 0-3, 2-7, 6-9 and 1-8, we compute a list of overlap sets that are ABD and BCD. A and C don't overlap, so they must be in separate sets. A, B and D overlap at token 2 and must thus be present in at least one set, and similarly for B, C and D at 7. Only A and D overlap at 1, but the set ABD already includes this combination.

These overlap sets are sufficient to decide whether or not a compaction should be carried out, if and only if the number of elements in a set is at least as large as `s_{t}`. However, we may need to include more SSTables in the compaction than this set alone.

It is possible for our sharding scheme to end up constructing SSTables spanning differently-sized shards for the same level. One clear example is the case of leveled compaction. In this case, SSTables enter at some density, and after the first compaction the resulting SSTable is 2x bigger than the initial density, causing the SSTable to split in half at the middle of the token range. When another SSTable enters the same level, we will have separate overlap sets between the two older SSTables and the new one. For efficiency, the compaction that is triggered next needs to select both of the overlap sets.

To deal with cases of partial overlap, the overlap sets will transitively extend with all neighboring ones that share some SSTable. Thus, the set of all SSTables that is constructed has some chain of overlapping SSTables that connects it to the initial set. This extended set forms the compaction bucket.

NOTE In addition to `TRANSITIVE`, "overlap inclusion methods" of `NONE` and `SINGLE` are also implemented for experimentation, but they are not recommended for the UCS sharding scheme.

In normal operation, we compact all SSTables in the compaction bucket. If compaction is very late, we may apply a limit on the number of overlapping sources we compact. In that case, we use the collection of oldest SSTables that would select at most limit-many in any included overlap set, making sure that if an SSTable is included in this compaction, all older ones are also included to maintain time order.

Selecting the compaction to run

Compaction strategies aim to minimize the read amplification of queries, which is defined by the number of SSTables that overlap on any given key. For highest efficiency in situations where compaction is late, a compaction bucket is selected with the highest overlap among the possible choices. If there are multiple choices, choose one uniformly and randomly within each level. Between the levels, prefer the lowest level, as this is expected to cover a larger fraction of the token space for the same amount of work.

Under sustained load, this mechanism prevents the accumulation of SSTables on some level that could sometimes happen with legacy strategies. With older strategies, all resources could be consumed by L0 and SSTables accumulating on L1. With UCS, a steady state where compactions always use more SSTables than the assigned threshold and fan factor is accomplished, and a tiered hierarchy is maintained based on the lowest overlap they are able to maintain for the load.

Major compaction

Under the working principles of UCS, a major compaction is an operation which compacts together all SSTables that have (transitive) overlap, and where the output is split on shard boundaries appropriate for the expected resulting density.

In other words, a major compaction will result in `b` concurrent compactions, each containing all SSTables covered in each of the base shards. The result will be split on shard boundaries whose number depends on the total size of data contained in the shard.

Differences with STCS and LCS

Note that there are some differences between tiered UCS and legacy STCS, and between leveled UCS and legacy LCS.

Tiered UCS vs STCS

STCS is very similar to UCS. However, STCS defines buckets/levels by looking for similarly sized SSTables, rather than using a predefined banding of sizes. Thus, STCS can end up with some odd selections of buckets, spanning SSTables of wildly different sizes; UCS's selection is more stable and predictable.

STCS triggers a compaction when it finds at least `min_threshold` SSTables on some bucket, and it compacts between `min_threshold` and `max_threshold` SSTables from that bucket at a time. `min_threshold` is equivalent to UCS's `t = f = w + 2`. UCS drops the upper limit because its compaction is still efficient with very large numbers of SSTables.

UCS uses a density measure to split results, in order to keep the size of SSTables and the time for compactions low. Within a level, UCS will only consider overlapping SSTables when deciding whether the threshold is hit, and will independently compact sets of SSTables that do not overlap.

If there are multiple choices to pick SSTables within a bucket, STCS groups them by size, while UCS groups them by timestamp. Because of that, STCS easily loses time order which makes whole table expiration less efficient. UCS efficiently tracks time order and whole table expiration. Because UCS can apply whole-table expiration, this features also proves useful for time-series data with time-to-live constraints.

UCS-leveled vs LCS

LCS seems very different in behaviour compared to UCS. However, the two strategies are, in fact, very similar.

LCS uses multiple SSTables per level to form a sorted run of non-overlapping SSTables of small fixed size. So physical SSTables on increasing levels increase in number (by a factor of `fanout_size`) instead of size. In this way, LCS reduces space amplification and ensures shorter compaction times. When the combined size of a run on a level is higher than expected, it selects some SSTables to compact with overlapping ones from the next level of the hierarchy. Eventually, the size of the next level gets pushed over its size limit and triggers higher-level operations.

In UCS, SSTables on increasing levels increase in density by a fanout factor `f`. A compaction is triggered when a second overlapping SSTable is located on a sharded level. UCS compacts the overlapping bucket on that level, and the result most often ends up on that level, too. But eventually, the data reaches sufficient size for the next level. Given an even data spread, UCS and LCS behave similarly, with compactions triggered in the same timeframe.

The two approaches end up with a very similar effect. UCS has the added benefit that compactions cannot affect other levels. In LCS, L0-to-L1 compactions can prevent any concurrent L1-to-L2 compactions, an unfortunate situation. In UCS, SSTables are structured such that they can be easily switched to tiered UCS or changed with different parameter settings.

Because LCS SSTables are based on size only and thus vary on split position, when LCS selects SSTables to compact on the next level, some SSTables that only partially overlap are included. Consequently, SSTables can be compacted more often than strictly necessary.

UCS handles the problem of space amplification by sharding on specific token boundaries. LCS splits SSTables based on a fixed size with boundaries usually falling inside SSTables on the next level, kicking off compaction more frequently than necessary. Therefore UCS aids with tight write amplification control. Those boundaries guarantee that we can efficiently select higher-density SSTables that exactly match the span of the lower-density ones.

UCS Options

Subproperty	**Description**
enabled	Enables background compaction. Default value: true
only_purge_repaired_tombstone	Enabling this property prevents data from resurrecting when repair is not run within the `gc_grace_seconds`. Default value: false

Subproperty	Description
scaling_parameters	A list of per-level scaling parameters, specified as `L_{f}`, `T_{f}`, `N`, or an integer value specifying `w` directly. If more levels are present than the length of this list, the last value is used for all higher levels. Often this will be a single parameter, specifying the behaviour for all levels of the hierarchy. Leveled compaction, specified as `L_{f}`, is preferable for read-heavy workloads, especially if bloom filters are not effective (e.g. with wide partitions); higher levelled fan factors improve read amplification (and hence latency, as well as throughput for read-dominated workloads) at the expense of increased write costs. The equivalent of legacy LCS is L10. Tiered compaction, specified as `T_{f}`, is preferable for write-heavy workloads, or ones where bloom filters or time order can be exploited; higher tiered fan factors improve the cost of writes (and hence throughput) at the expense of making reads more difficult. `N` is the middle ground that has the features of leveled (one SSTable run per level), as well as tiered (one compaction to be promoted to the next level) and a fan factor of 2. This value can also be specified as T2 or L2. Default value: T4 (STCS with threshold 4)
target_sstable_size	The target sstable size `s_{t}`, specified as a human-friendly size in bytes, such as MiB. The strategy will split data in shards that aim to produce sstables of size between `s_{t}/\sqrt{2}` and `s_{t} \cdot \sqrt{2}`. Smaller sstables improve streaming and repair, and make compactions shorter. On the other hand, each sstable on disk has a non-trivial in-memory footprint that also affects garbage collection times. Increase this if the memory pressure from the number of sstables in the system becomes too high. Default: 1 GiB
min_sstable_size	The minimum sstable size, applicable when the base shard count will result is SSTables that are considered too small. If set, the strategy will split the space into fewer than the base count shards, to make the estimated SSTables size at least as large as this value. A value of 0 disables this feature. Default: 100MiB

Subproperty	Description
base_shard_count	The minimum number of shards `b`, used for levels with the smallest density. This gives the minimum compaction concurrency for the lowest levels. A low number would result in larger L0 sstables but may limit the overall maximum write throughput (as every piece of data has to go through L0). Default: is 4 (1 for system tables, or when multiple data locations are defined)
sstable_growth	The sstable growth component `\lambda`, applied as a factor in the shard exponent calculation. This is a number between 0 and 1 that controls what part of the density growth should apply to individual sstable size and what part should increase the number of shards. Using a value of 1 has the effect of fixing the shard count to the base value. Using 0.5 makes the shard count and SSTable size grow with the square root of the density growth. This is useful to decrease the sheer number of SSTables created for very large data sets. For example, without growth correction, a data set of 10TiB with 1GiB target size would result in over 10K SSTables. That many SSTables can result in too much overhead, both for on-heap memory used by per-SSTables structures, as well as lookup-time for intersecting SSTables and tracking overlapping sets during compaction. For example, with a base shard count of 4, the growth factor can reduce the potential number of SSTables to ~160 of size ~64GiB, manageable in terms of memory overhead, individual compaction duration, and space overhead. The parameter can be adjusted, increasing the value to get fewer but bigger SSTables on the top level, and decreasing the value to favour a higher count of smaller SSTables. Default: 0.333 (SSTable size grows with the square root of the growth of the shard count)
expired_sstable_check_frequency_seconds	Determines how often to check for expired SSTables. Default: 10 minutes

Subproperty	Description
max_sstables_to_compact	The maximum number of sstables to compact in one operation. Larger value may reduce write amplification but can cause very long compactions, and thus a very high read amplification overhead while such compactions are processing. The default aims to keep the length of operations under control and prevent accumulation of SSTables while compactions are taking place. If the fanout factor is larger than the maximum number of SSTables, the strategy will ignore the latter. Default: none (although 32 is a good choice)
overlap_inclusion_method	Specifies how to extend overlapping sections into buckets. TRANSITIVE makes sure that if we choose an SSTable to compact, we also compact the ones that overlap with it. SINGLE only does this extension once (i.e. it selects only SSTables that overlap with the original overlapping SSTables section. NONE does not add any overlapping sstables. NONE is not recommended, SINGLE may offer a little more parallelism at the expense of recompacting some data when upgrading from LCS or during range movements. Default: TRANSITIVE
unsafe_aggressive_sstable_expiration	Expired SSTables are dropped without checking if their data is shadowing other SSTables. This flag can only be enabled if `cassandra.allow_unsafe_aggressive_sstable_expiration` is true. Turning this flag on can cause correctness issues, such as the reappearance of deleted data. See discussions in CASSANDRA-13418 for valid use cases and potential problems. Default: false

In `cassandra.yaml`, there is also one parameter that affects compaction:

concurrent_compactors

Number of simultaneous compactions to allow, NOT including validation "compactions" for anti-entropy repair. Higher values increase compaction performance but may increase read and write latencies.

Size Tiered Compaction Strategy (STCS)

Size Tiered Compaction Strategy (STCS)

IMPORTANT The Unified Compaction Strategy (UCS) is the recommended compaction strategy for most workloads starting with **Cassandra 5.0**. If you are creating new tables, use this strategy.

The `SizeTieredCompactionStrategy (STCS)` is recommended for write-intensive workloads, and is the legacy recommended compaction strategy. It is the default compaction strategy if no other strategy is specified.

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 them into even larger SSTables. At any given time, several SSTables of varying sizes are present.

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 fact 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. However, SSTables may not grow quickly enough to merge and evict old data regularly.

Most STCS compactions are minor compactions, which merge a few SSTables into one. In contrast, when executing a major compaction with STCS, two SSTables per data directory, one for repaired data and one for unrepaired data, will exist during the compaction. 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. This phenomenon is known as space amplification, the problem of growing SSTable size, and the issue of outgrowing a cluster’s ability to do compaction. Major compactions are not recommended for STCS.

STCS depends on the calculation of the average size of SSTables to determine which SSTables to merge. This process is called bucketing. The following options are used to calculate the bucket into which an SSTable will be grouped, based on that average size. The bucketing process groups SSTables based on their size differing from the average size by either 50% or 150% more than the average size. Another way to state this calculation is that the bucketing process groups SSTables with a size within [average-size × bucket_low] and [average-size × bucket_high].

IMPORTANT The `SizeTieredCompactionStrategy` is the default compaction strategy. Any other compaction strategy must be defined in the `cassandra.yaml` file.

STCS options

SizeTieredCompactionStrategy (STCS) options are set per table using table options. The `min_threshold` option of a table is the main value that triggers a minor compaction. Minor compactions do not involve all the tables in a keyspace.

Subproperty	**Description**
enabled	Enables background compaction. Default value: true
tombstone_compaction_interval	The minimum number of seconds after which an SSTable is created before **Cassandra** considers the SSTable for tombstone compaction. An SSTable is eligible for tombstone compaction if the table exceeds the `tombstone_threshold` ratio. Default value: 86400
tombstone_threshold	The ratio of garbage-collectable tombstones to all contained columns. If the ratio exceeds this limit, **Cassandra** starts compaction on that table alone, to purge the tombstones. Default value: 0.2
unchecked_tombstone_compaction	If set to `true`, allows **Cassandra** to run tombstone compaction without pre-checking which tables are eligible for this operation. Even without this pre-check, **Cassandra** checks an SSTable to make sure it is safe to drop tombstones. Default value: false
log_all	Activates advanced logging for the entire cluster. Default value: false
max_threshold	The maximum number of SSTables to allow in a minor compaction. Default value: 32
min_threshold	The minimum number of SSTables to trigger a minor compaction. Default value: 4
bucket_high	An SSTable is added to a bucket if its size is less than 150% of the average size of that bucket. For example, if the SSTable size is 13 MB, and the bucket average size is 10 MB, then the SSTable will be added to that bucket and the new average size will be computed for that bucket. Default value: 1.5
bucket_low	An SSTable is added to a bucket if the SSTable size is greater than 50% of the average size of that bucket. For example, if the SSTable size is 6 MB, and the bucket average size is 10 MB, then the SSTable will be added to that bucket and the new average size will be computed for that bucket. Default value: 0.5
min_sstable_size	SSTables smaller than this value will be grouped into one bucket where the average size is less than this setting. Default value: 50MB

Subproperty	Description
only_purge_repaired_tombstones	If set to `true`, allows purging tombstones only from repaired SSTables. The purpose is to prevent data from resurrecting if repair is not run within `gc_grace_seconds`. If you do not run repair for a long time, **Cassandra** keeps all tombstones — this may cause problems. Default value: false

Leveled Compaction Strategy (LCS)

Leveled Compaction Strategy (LCS)

IMPORTANT The Unified Compaction Strategy (UCS) is the recommended compaction strategy for most workloads starting with **Cassandra 5.0**. If you are creating new tables, use this strategy.

The `LeveledCompactionStrategy (LCS)` is recommended for read-heavy workloads, although UCS is the best recommendation for all workloads today. It alleviates some of the read operation issues with STCS, while providing reasonable write operations. This strategy works with a series of levels, where each level contains a set of SSTables. When data in memtables is flushed, SSTables are written in the first level (L0), where SSTables are not guaranteed to be non-overlapping. LCS compaction merges these first SSTables with larger SSTables in level L1. Each level is by default 10x the size of the previous one. Once an SSTable is written to L1 or higher, the SSTable is guaranteed to be non-overlapping with other SSTables in the same level. If a read operation needs to access a row, it will only need to look at one SSTable per level.

To accomplish compaction, all overlapping SSTables are merged into a new SSTable in the next level. For L0 → L1 compactions, we almost always need to include all L1 SSTables since most L0 SSTables cover the full range of partitions. LCS compacts SSTables from one level to the next, writing partitions to fit a defined SSTable size. In addition, each level has a prescribed size, so that compaction will be triggered when a level reaches its size limit. Creating new SSTables in one level can trigger compaction in the next level, and so on, until all levels have been compacted based on the settings.

There is a failsafe if too many SSTables reads are being done in the L0 level. An STCS compaction will be triggered in L0 if there are more than 32 SSTables in L0. This compaction quickly merges SSTables out of L0, and into L1, where they will be compacted to non-overlapping SSTables.

LCS is not as disk hungry as STCS, needing only approximately 10% of disk to execute, but it is more IO and CPU intensive. For ongoing minor compactions in a read-heavy workload, the amount of compaction is reasonable. It is not a good choice for write-heavy workloads, though, because it will cause a lot of disk IO and CPU usage. Major compactions are not recommended for LCS.

Bootstrapping

During bootstrapping, SSTables are streamed from other nodes. Because many SSTables will be both flushed from the new writes to memtables, as well as streaming from a remote note, the new node will have many SSTables in L0. To avoid a collision of the flushing and streaming SSTables, only STCS in L0 is executed until the bootstrapping is complete.

Starved sstables

If the leveling is not optimal, LCS can end up with starved sstables. High level SSTables can be stranded and not compacted, because SSTables in lower levels are not getting merged and compacted. For example, this situation can make it impossible for lower levels to drop tombstones. If these starved SSTables are not resolved within a defined number of compaction rounds, they will be included in other compactions. This situation generally occurs if a user lowers the `sstable_size` setting.

IMPORTANT	The `SizeTieredCompactionStrategy` is the default compaction strategy. Any other compaction strategy must be defined in the `cassandra.yaml` file.

LCS options

Subproperty	Description
enabled	Enables background compaction. Default value: true
tombstone_compaction_interval	The minimum number of seconds after which an SSTable is created before **Cassandra** considers the SSTable for tombstone compaction. An SSTable is eligible for tombstone compaction if the table exceeds the `tombstone_threshold` ratio. Default value: 86400
tombstone_threshold	The ratio of garbage-collectable tombstones to all contained columns. If the ratio exceeds this limit, **Cassandra** starts compaction on that table alone, to purge the tombstones. Default value: 0.2
unchecked_tombstone_compaction	If set to `true`, allows **Cassandra** to run tombstone compaction without pre-checking which tables are eligible for this operation. Even without this pre-check, **Cassandra** checks an SSTable to make sure it is safe to drop tombstones. Default value: false
log_all	Activates advanced logging for the entire cluster. Default value: false
sstable_size_in_mb	The target size for SSTables. Although SSTable sizes should be less or equal to sstable_size_in_mb, it is possible that compaction could produce a larger SSTable during compaction. This occurs when data for a given partition key is exceptionally large. The **Cassandra** database does not split the data into two SSTables. Default: 160
fanout_size	The target size of levels increases by this `fanout_size` multiplier. You can reduce the space amplification by tuning this option. Default: 10
single_sstable_uplevel	??? Default: true

LCS also supports a startup option, `-Dcassandra.disable_stcs_in_l0=true` which disables STCS in L0.

Time Window Compaction Strategy (TWCS)

Time Window Compaction Strategy (TWCS)

IMPORTANT The Unified Compaction Strategy (UCS) is the recommended compaction strategy for most workloads starting with **Cassandra 5.0**. If you are creating new tables, use this strategy.

The `TimeWindowCompactionStrategy` (TWCS) is the recommended compaction strategy for time-series and expiring Time-To-Live (TTL) workloads. If the workload has data grouped by timestamp, TWCS can be used to group SSTables by time window, and then drop entire SSTables when they expire. Thus, disk space can be reclaimed much more reliably than when using STCS or LCS.

TWCS compacts SSTables using a series of time windows buckets. During the active time window, TWCS compacts all SSTables flushed from memory into larger SSTables using STCS. At the end of the time period, all of these SSTables are compacted 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. Notice that each TWCS time window contains varying amounts of data. Then the next time window starts and the process repeats.

TimeWindowCompactionStrategy Operational Concerns

The primary motivation for TWCS is to separate data on disk by timestamp and to allow fully expired SSTables to drop more efficiently. One potential way this optimal behavior can be subverted is if data is written to SSTables out of order, with new data and old data in the same SSTable.

Out of order data can appear in two ways:

- If the user mixes old data and new data in the traditional write path, the data will be comingled in the memtables and flushed into the same SSTable, where it will remain comingled.
- If the user's read requests for old data cause read repairs that pull old data into the current memtable, that data will be comingled and flushed into the same SSTable.

While TWCS tries to minimize the impact of comingled data, users should attempt to avoid this behavior. Specifically, users should avoid queries that explicitly set the timestamp via CQL `USING TIMESTAMP`. Additionally, users should run frequent repairs (which streams data in such a way that it does not become comingled).

IMPORTANT The `SizeTieredCompactionStrategy` is the default compaction strategy. Any other compaction strategy must be defined in the `cassandra.yaml` file.

TWCS Options

Subproperty	Description
enabled	Enables background compaction. Default value: true
tombstone_compaction_interval	The minimum number of seconds after which an SSTable is created before **Cassandra** considers the SSTable for tombstone compaction. An SSTable is eligible for tombstone compaction if the table exceeds the `tombstone_threshold` ratio. Default value: 86400
tombstone_threshold	The ratio of garbage-collectable tombstones to all contained columns. If the ratio exceeds this limit, **Cassandra** starts compaction on that table alone, to purge the tombstones. Default value: 0.2
unchecked_tombstone_compaction	If set to `true`, allows **Cassandra** to run tombstone compaction without pre-checking which tables are eligible for this operation. Even without this pre-check, **Cassandra** checks an SSTable to make sure it is safe to drop tombstones. Default value: false
log_all	Activates advanced logging for the entire cluster. Default value: false
compaction_window_unit	Time unit used to define the bucket size. The value is based on the Java TimeUnit. For the list of valid values, see the Java API TimeUnit page located at https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TimeUnit.html. Default: days
compaction_window_size	Units per bucket. Default value: 1
timestamp_resolution	The timestamp resolution used when inserting data, can be MILLISECONDS, MICROSECONDS etc (should be understandable by Java TimeUnit). Do not change this value unless you do mutations with USING TIMESTAMP (or equivalent directly in the client). Default: microseconds
expired_sstable_check_frequency_seconds	Determines how often to check for expired SSTables. Default: 10 minutes
unsafe_aggressive_sstable_expiration	Expired SSTables will be dropped without checking its data is shadowing other SSTables. This is a potentially risky option that can lead to data loss or deleted data re-appearing, going beyond what `unchecked_tombstone_compaction` does for single SSTable compaction. Due to the risk, the JVM must also be started with the option: `-Dcassandra unsafe_aggressive_sstable_expiration=true`. Default: false

Taken together, the operator can specify windows of virtually any size, and TWCS will work to create a single SSTable for writes within that window. For efficiency during writing, the newest window will be compacted using STCS.

Ideally, operators should select a `compaction_window_unit` and `compaction_window_size` pair that produces approximately 20-30 windows. If writing with a 90 day TTL, for example, a 3 Day window would be a reasonable choice, setting the options to `'compaction_window_unit':'DAYS' and 'compaction_window_size':3`.

Changing TimeWindowCompactionStrategy Options

Operators wishing to enable TWCS on existing data should consider running a major compaction first, placing all existing data into a single (old) window. Subsequent newer writes will then create typical SSTables as expected.

Operators wishing to change `compaction_window_unit` or `compaction_window_size` can do so, but may trigger additional compactions as adjacent windows are joined together. If the window size is decrease d (for example, from 24 hours to 12 hours), then the existing SSTables will not be modified. TWCS can not split existing SSTables into multiple windows.