How is data maintained?

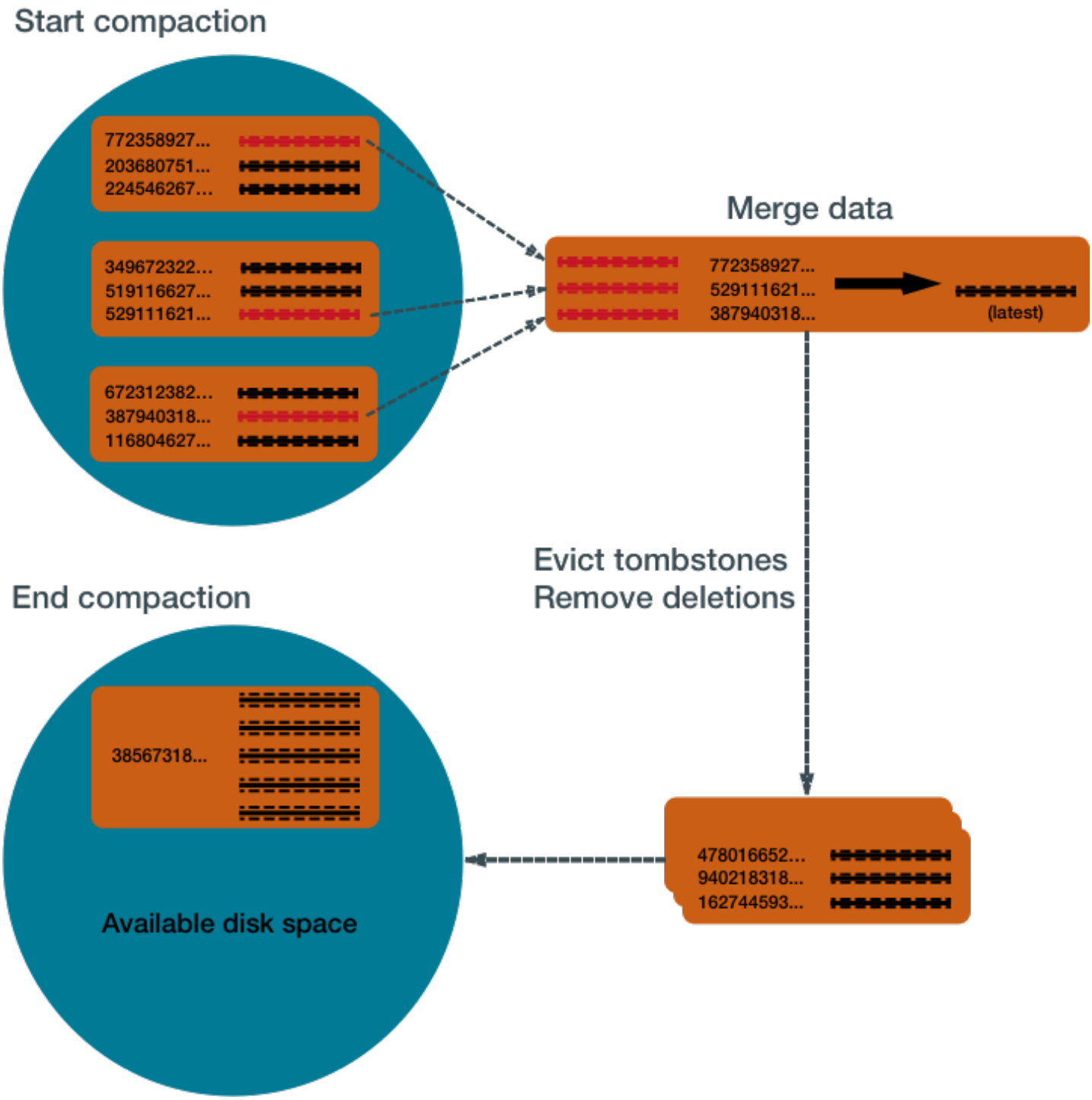
The Cassandra write process stores data in files called SSTables. SSTables are immutable. Instead of overwriting existing rows with inserts or updates, Cassandra writes new timestamped versions of the inserted or updated data in new SSTables. Cassandra does not perform deletes by removing the deleted data: instead, Cassandra marks it with tombstones.

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 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

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.

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.

SizeTieredCompactionStrategy (STCS)

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

LeveledCompactionStrategy (LCS)

Recommended for read-intensive workloads.

- **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

TimeWindowCompactionStrategy (TWCS)

Recommended for time series and expiring TTL workloads.

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)

DateTieredCompactionStrategy (DTCS)

Deprecated in Cassandra 3.0.8/3.8.

- **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.

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.

Configuring and running compaction

Set the compaction strategy for a table in the parameters for the `CREATE TABLE` or `ALTER TABLE` command.

You can start compaction manually using the `nodetool compact` command.

Define a compaction class and properties in simple JSON format:

```
compaction = {
   'class' : 'compaction_strategy_name'
   [, 'subproperty_name' : 'value',...]
}
```

compaction properties

Cassandra provides the following compaction classes, each class has different subproperties:

- SizeTieredCompactionStrategy (STCS)
- DateTieredCompactionStrategy
- TimeWindowCompactionStrategy (TWCS)
- LeveledCompactionStrategy (LCS)

Enabling and disabling background compaction

The following example sets the enable property to disable background compaction:

```
use cycling;
ALTER TABLE cyclist_name WITH COMPACTION = {'class': 'SizeTieredCompacti
onStrategy', 'enabled': 'false'};
```

Disabling background compaction can be harmful: without it, Cassandra does not regain disk space, and may allow zombies to propagate. Although compaction uses I/O, it is better to leave it enabled in most cases.

compaction thresholds

```
nodetool getcompactionthreshold cycling cyclist_name
Current compaction thresholds for cycling/cyclist_name:
 min = 4,  max = 32
```

```
nodetool setcompactionthreshold cycling cyclist_name 2 8

nodetool getcompactionthreshold cycling cyclist_name
Current compaction thresholds for cycling/cyclist_name:
 min = 2,  max = 8
```

nodetool compact

```
nodetool compact
```

```
nodetool compactionstats
- stresscql.blogposts: 1

id                                   compaction type keyspace  table
 completed total       unit  progress
9b5e2f10-6a7c-11e8-a278-d96051c59bfd Compaction      stresscql blogposts
 408586714 13015328594 bytes 3.14%
Active compaction remaining time :   0h12m31s
```