**How is data read from cassandra?**

In Apache Cassandra, reading data from SSTables (Sorted String Tables) involves a process called "SSTable scanning." When a read request is received, Cassandra follows a specific process to retrieve the requested data from the SSTables. Here's a simplified explanation of how Cassandra reads data from SSTables:

1. Key Lookup: Cassandra uses a distributed hash table called the "Partitioner" to determine which node in the cluster contains the data for the requested key. It identifies the node responsible for that key and routes the read request to that node.

2. Memtable Lookup: Upon receiving the read request, the node first checks if the requested data is available in the Memtable. The Memtable is an in-memory data structure where recently written or updated data is stored. If the data is found in the Memtable, Cassandra retrieves it directly and returns the result.

3. SSTable Lookup: If the data is not found in the Memtable, Cassandra moves on to the SSTable lookup. Cassandra maintains a list of SSTables on disk that contain the data for each node. The SSTables are stored in a directory structure on the node's local file system.

4. Bloom Filter: Before accessing the SSTables, Cassandra utilizes a data structure called a Bloom filter. The Bloom filter is a probabilistic filter that quickly determines whether an SSTable might contain the requested data or not. If the Bloom filter indicates that the SSTable may contain the data, Cassandra proceeds to scan the SSTable.

5. SSTable Scanning: During SSTable scanning, Cassandra sequentially reads the SSTables and compares the requested key with the keys stored in the SSTables. Since the SSTables are sorted by key, Cassandra can use efficient binary search algorithms to locate the data quickly. Once the key is found, Cassandra retrieves the corresponding value and returns it as the result of the read request.

6. Compaction Considerations: It's important to note that if compaction is currently in progress, Cassandra may need to consult multiple SSTables to retrieve the requested data. Compaction merges and organizes SSTables, so the data for a single key may be spread across multiple SSTables. In such cases, Cassandra performs a process called "read repair" to ensure consistency and retrieve the most up-to-date value for the requested key.

This process allows Cassandra to efficiently read data from SSTables by leveraging the sorted nature of the SSTables and utilizing Bloom filters for faster lookup. It's worth noting that other components, such as the row cache and key cache, can further optimize read performance by caching frequently accessed data in memory.

**What is compaction?**  
In Apache Cassandra, compaction is a process that merges multiple SSTables (Sorted String Tables) into a single SSTable to improve read performance and manage disk space more efficiently.

SSTables are the on-disk data storage format in Cassandra, where each SSTable represents a sorted sequence of key-value pairs. As data is continuously written and updated in Cassandra, new SSTables are created. Over time, the accumulation of SSTables can lead to inefficient read operations and excessive disk space usage.Compaction helps address these issues by combining multiple SSTables into a smaller number of SSTables. During the compaction process, Cassandra identifies overlapping data ranges in the SSTables, merges them, discards tombstoned (deleted) data, and creates a new compacted SSTable. This process reduces the number of SSTables and improves read efficiency by reducing the number of disk seeks required to retrieve data.

**SizeTieredCompactionStrategy:**

The SizeTieredCompactionStrategy is a compaction strategy used in Apache Cassandra to manage data storage and improve read performance. It operates by dividing SSTables (Sorted String Tables) into different tiers based on their size and triggers compaction when certain thresholds are reached.

Here's a simple explanation of how the SizeTieredCompactionStrategy works with an example:

1. Initial Data: Let's say you have a Cassandra cluster with three nodes, and you start storing data. As data is written to each node, Cassandra creates SSTables to store the data. Each SSTable represents a sorted sequence of key-value pairs.
2. SSTable Accumulation: Over time, more data is written to the cluster, resulting in the creation of multiple SSTables on each node. These SSTables may have different sizes, depending on the amount of data written to them.
3. Compaction Threshold: The SizeTieredCompactionStrategy divides SSTables into different tiers or levels based on their size. Each level has a configurable threshold size. For example, let's assume the threshold size for the first level is 100 MB, the second level is 200 MB, and so on.
4. Compaction Trigger: When the total size of SSTables in a specific level exceeds its threshold, a compaction is triggered for that level. For instance, if the total size of SSTables in the first level reaches 100 MB, a compaction is triggered to merge those SSTables into a single, larger SSTable.
5. Compaction Process: During compaction, Cassandra identifies overlapping data ranges in the SSTables and merges them into a new, compacted SSTable. Tombstoned (deleted) data is also discarded during this process to reclaim disk space. The compacted SSTable replaces the original SSTables, reducing the number of SSTables and improving read performance.
6. Multiple Tiers: As more data accumulates and compaction occurs, the SSTables may move to higher tiers based on their increasing size. The higher tiers have larger threshold sizes, which means compaction in those tiers happens less frequently.
7. Ongoing Compaction: The SizeTieredCompactionStrategy continues to monitor the size of SSTables in each tier and triggers compaction whenever the thresholds are exceeded. This ongoing compaction process ensures efficient disk space utilization and optimal read performance.

The SizeTieredCompactionStrategy is a simple and effective compaction strategy that is suitable for a wide range of workloads. It focuses on optimizing storage efficiency by merging SSTables based on their size. However, it's important to note that this strategy may result in more read amplification compared to other compaction strategies like LeveledCompactionStrategy. Therefore, choosing the appropriate compaction strategy depends on your specific use case and workload characteristics.

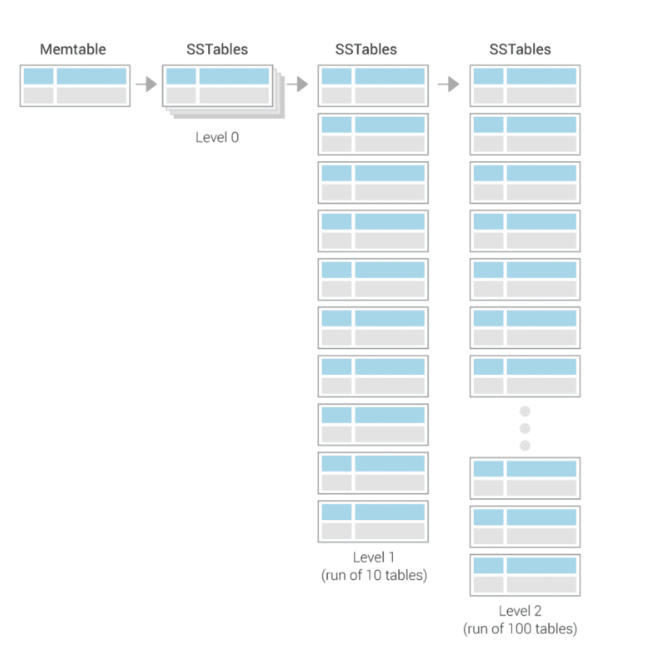
STCS works by dividing SSTables into different tiers based on their size. As new data is written and compacted, smaller SSTables are continuously merged into larger ones. However, this compaction process can result in space amplification for the following reasons:

1. Fragmentation: When smaller SSTables are merged into larger ones, there might be some unused space within the resulting SSTable. This unused space occurs due to the varying sizes of the original SSTables and the fixed size thresholds for each tier. Over time, this fragmentation can lead to wasted disk space and suboptimal space utilization.
2. Overlapping Data: In STCS, SSTables in different tiers may contain overlapping data ranges. When a read request is made, Cassandra needs to check multiple SSTables across different tiers to find the relevant data. This can result in duplicate data being stored in multiple SSTables, further contributing to space amplification.
3. Tombstones: Tombstones are markers used in Cassandra to mark deleted data. In STCS, tombstones can accumulate in SSTables and occupy disk space until compaction occurs. If there are frequent deletions or updates in the data, the space consumed by tombstones can increase, leading to space amplification.

**LeveledCompactionStrategy:**Compaction Strategy (LCS), the SSTable size remains the same across all levels. Each level in LCS contains SSTables of the same fixed size, typically set to 160 MB.

In LCS, as data is written and compacted, it progresses through the levels. When the number of SSTables in a level exceeds a certain threshold, compaction is triggered to merge the SSTables into a single SSTable in the next level. This process continues as data moves through the levels, with each level containing SSTables of the same fixed size.

The fixed SSTable size in LCS ensures that the data is evenly distributed across the levels. This helps in maintaining a more balanced storage structure and avoiding excessively large SSTables that can cause performance issues.



Write amplification can occur in LCS for the following reasons:

1. Compaction Overhead: The compaction process in LCS involves reading data from multiple SSTables, merging them, and writing the compacted SSTable. This process incurs additional disk I/O and CPU usage, which can lead to increased write amplification compared to the actual amount of data being written by the client.
2. Leveling Up: As data progresses through the levels in LCS, SSTables are promoted to higher levels based on certain criteria. This leveling up process involves rewriting the data from multiple SSTables into a single SSTable in the next level. This rewriting of data contributes to write amplification.

Write amplification in LCS can impact the overall write performance and disk utilization. Higher write amplification means more disk I/O operations, increased disk space usage, and potentially reduced write throughput.

To mitigate write amplification in LCS, it's important to carefully tune the compaction settings, such as the size thresholds for triggering compaction and the number of levels. Additionally, optimizing the data model and workload patterns can help minimize unnecessary data duplication and reduce write amplification.

**TimeWindowCompactionStrategy**

The TimeWindowCompactionStrategy (TWCS) is a compaction strategy in Cassandra that is optimized for time-series data. It is particularly useful when data is inserted with a monotonically increasing timestamp, such as event logs, sensor readings, or financial market data. The primary goal of TWCS is to optimize read performance for recent data while still retaining older data.

TWCS organizes data into multiple "windows" or "time intervals" based on a configurable time duration. Each window has its own set of SSTables. When compaction occurs, it focuses on compacting only the most recent window, ensuring that recent data is stored in a more efficient manner.

TWCS provides the following benefits:

1. Efficient Compaction: TWCS compacts only the most recent window, minimizing the amount of data that needs to be compacted during each compaction process. This improves compaction speed and reduces the overall I/O overhead.
2. Optimized Read Performance: By compacting recent data separately, TWCS improves read performance for time-series workloads. The most recent data is stored in a compacted and optimized form, making it faster to access.
3. Retention of Older Data: TWCS retains older data in separate windows without frequent compaction. This allows for efficient storage of historical data while focusing compaction efforts on the most recent data.

**Tombstones and Compaction:**

Cassandra does not delete data from the disk at once. Instead, the system writes a tombstone, a special value that indicates that data has been deleted. Tombstones mark data for tombstone compaction and prevent deleted data from being returned during reads.

**Does compaction delete tombstones? Does it wait for gray seconds?**

Yes, compaction in Apache Cassandra does handle tombstones and their removal from SSTables.

During the compaction process, tombstones (deleted or expired data markers) are considered for removal to reclaim disk space and improve read performance. Compaction merges multiple SSTables and removes redundant or overwritten data, including tombstones.

Cassandra employs a concept called "tombstone compaction threshold" to determine when tombstones should be removed during compaction. By default, Cassandra uses the tombstone compaction threshold value of 0.2, which means that once the ratio of tombstones to live data in an SSTable exceeds 20%, the tombstones are eligible for removal during compaction.

When a compaction occurs, Cassandra compares the timestamp of each tombstone with the current time to determine if it is "expired" or eligible for removal. Tombstones that have exceeded the configured "gc\_grace\_seconds" (typically 10 days by default) are considered expired and can be safely removed during compaction.

The compaction process respects the "gc\_grace\_seconds" value and waits for that period before removing expired tombstones. This grace period ensures that all replicas in the cluster have sufficient time to receive the tombstone deletion information and apply it consistently before the tombstone is permanently deleted during compaction.

By removing expired tombstones during compaction, Cassandra avoids unnecessary storage and improves read performance by reducing the amount of data to be read from disk.

It's important to note that tombstone removal during compaction depends on the specific compaction strategy being used and the configuration settings. Additionally, tombstones that are not yet expired or fall within the grace period will not be removed during compaction to maintain data consistency across replicas.

**What if one node doesn't receive the delete request and doesn't have the tombstone while other nodes have for that data, what happens then?**If one node in the cluster does not receive the delete request or fails to apply the delete operation for some reason, while other nodes have successfully processed the delete request and created tombstones, a situation known as "inconsistency" occurs.

In Cassandra, read operations are typically performed at a consistency level that ensures data consistency across replicas. By default, the consistency level is set to "QUORUM," which requires a majority of replicas to respond to the read request. In this case, if the node that is missing the tombstone is part of the replica set being queried, it may return the data as if it is not deleted, resulting in inconsistent results.

To handle such scenarios, Cassandra provides a concept called "read repair" and "hinted handoff":

1. Read Repair: Cassandra periodically performs read repair in the background to compare data between replicas and automatically reconcile any inconsistencies. During a read repair, if a replica node is missing the tombstone, it will fetch the missing data and tombstone from other replicas, ensuring data consistency.
2. Hinted Handoff: When a node that is responsible for a particular piece of data is temporarily down or unreachable, Cassandra employs hinted handoff. The coordinator node that receives the delete request will store a "hint" indicating that the node is down and will forward the delete request to the unavailable node once it comes back online. This ensures that the delete operation is eventually applied to all replicas.

With these mechanisms in place, Cassandra strives to maintain eventual consistency across replicas, even if certain nodes temporarily miss delete requests or tombstones. It may take some time for the inconsistencies to be resolved through read repair and hinted handoff processes.

**Which compaction strategy is best?**

The following questions are based on developer and user experience with the [compaction strategies](https://docs.datastax.com/en/dse/5.1/dse-arch/datastax_enterprise/dbInternals/dbIntHowDataMaintain.html#dbIntHowDataMaintain__dml_types_of_compaction).

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

If the answer is yes, use [TWCS (TimeWindowCompactionStrategy)](https://docs.datastax.com/en/dse/5.1/dse-arch/datastax_enterprise/dbInternals/dbIntHowDataMaintain.html#dbIntHowDataMaintain__twcs). If the answer is no, read the following questions.

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

[LCS (LeveledCompactionStrategy)](https://docs.datastax.com/en/dse/5.1/dse-arch/datastax_enterprise/dbInternals/dbIntHowDataMaintain.html#dbIntHowDataMaintain__lcs-compaction) is appropriate if there are twice or more reads than writes, especially randomized reads. If the reads and writes are approximately equal, the performance penalty from LCS may not be worth the benefit. Be aware that LCS can be overwhelmed by a high number of writes. One advantage of LCS is that it keeps related data in a small set of SSTables.

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

If your data is *immutable* or there are few [upserts](https://docs.datastax.com/en/glossary/doc/glossary/gloss_upsert.html), use [STCS (SizeTieredCompactionStrategy)](https://docs.datastax.com/en/dse/5.1/dse-arch/datastax_enterprise/dbInternals/dbIntHowDataMaintain.html#dbIntHowDataMaintain__stcs-compaction), which does not have the write performance penalty of LCS.

**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 and write ratio is small, and the read activity is expected to conform to a Service Level Agreement (SLA), it may be worth the LCS write performance penalty to keep read rates and latency at predictable levels. And, you may be able to overcome the LCS write penalty by adding more nodes.

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

For batched reads and 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 tables that use LCS.

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

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

**Is your system reaching its limits for input and output?**

LCS is significantly more input and output intensive than DTCS or STCS. Switching to LCS may introduce extra input and output load that offsets the advantages.

**Cassandra-Stress test:**

Put stress on your cluster using the following:   
Cd /opt/cassandra/tools/bin  
./cassandra-stress write n=100000 -rate threads=50 -node [127.0.0.1](http://127.0.0.1/)(ip of node)  
  
**Cassandra-Stress test on user defined table:**<https://cassandra.apache.org/doc/latest/cassandra/tools/cassandra_stress.html>

Make a YAML file:

| keyspace: keyspace1  # Would almost always be network topology unless running something locally keyspace\_definition: |  CREATE KEYSPACE example WITH replication = {'class': 'SimpleStrategy', 'replication\_factor': 3};  table: staff\_activities  # The table under test. Start with a partition per staff member # Is this a good idea? table\_definition: |  CREATE TABLE staff\_activities (  name text,  when timeuuid,  what text,  PRIMARY KEY(name, when)  ) WITH compaction = {'class': 'LeveledCompactionStrategy'};  columnspec:  - name: name  size: uniform(5..10) # The names of the staff members are between 5-10 characters  population: uniform(1..10) # 10 possible staff members to pick from  - name: when  cluster: uniform(20..500) # Staff members do between 20 and 500 events  - name: what  size: normal(10..100,50)  insert:  # we only update a single partition in any given insert  partitions: fixed(1)  # we want to insert a single row per partition and we have between 20 and 500  # rows per partition  select: fixed(1)/500  batchtype: UNLOGGED # Single partition unlogged batches are essentially noops  queries:  events:  cql: select \* from staff\_activities where name = ?  fields: samerow  latest\_event:  cql: select \* from staff\_activities where name = ? LIMIT 1  fields: samerow |
| --- |

Save it as LCS\_profile.yaml in /opt/cassandra/tools/bin

Then use the following command to check speed of write operation of LCS:

| ./cassandra-stress user profile=./LCS\_profile.yaml duration=1m "ops(insert=1)" truncate=once |
| --- |

And for read operation use the following:

| ./cassandra-stress user profile=./LCS\_profile.yaml duration=1m "ops(events=1)" truncate=once |
| --- |

Then just change make another yaml file for STCS strategy and compare the speeds  
Here are the results:

**STCS write:**

| Results: Op rate : 5,678 op/s [insert: 5,678 op/s] Partition rate : 5,678 pk/s [insert: 5,678 pk/s] Row rate : 6,458 row/s [insert: 6,458 row/s] Latency mean : 0.6 ms [insert: 0.6 ms] Latency median : 0.5 ms [insert: 0.5 ms] Latency 95th percentile : 1.2 ms [insert: 1.2 ms] Latency 99th percentile : 2.1 ms [insert: 2.1 ms] Latency 99.9th percentile : 7.3 ms [insert: 7.3 ms] Latency max : 32.6 ms [insert: 32.6 ms] Total partitions : 344,538 [insert: 344,538] Total errors : 0 [insert: 0] Total GC count : 0 Total GC memory : 0.000 KiB Total GC time : 0.0 seconds Avg GC time : NaN ms StdDev GC time : 0.0 ms Total operation time : 00:01:00 |
| --- |

**LCS write:**

| Results: Op rate : 5,504 op/s [insert: 5,504 op/s] Partition rate : 5,504 pk/s [insert: 5,504 pk/s] Row rate : 6,259 row/s [insert: 6,259 row/s] Latency mean : 0.7 ms [insert: 0.7 ms] Latency median : 0.5 ms [insert: 0.5 ms] Latency 95th percentile : 1.4 ms [insert: 1.4 ms] Latency 99th percentile : 2.5 ms [insert: 2.5 ms] Latency 99.9th percentile : 7.4 ms [insert: 7.4 ms] Latency max : 89.1 ms [insert: 89.1 ms] Total partitions : 333,745 [insert: 333,745] Total errors : 0 [insert: 0] Total GC count : 0 Total GC memory : 0.000 KiB Total GC time : 0.0 seconds Avg GC time : NaN ms StdDev GC time : 0.0 ms Total operation time : 00:01:00 |
| --- |

**STCS read:**

| Results: Op rate : 7,258 op/s [events: 7,258 op/s] Partition rate : 0 pk/s [events: 0 pk/s] Row rate : 0 row/s [events: 0 row/s] Latency mean : 0.5 ms [events: 0.5 ms] Latency median : 0.4 ms [events: 0.4 ms] Latency 95th percentile : 0.9 ms [events: 0.9 ms] Latency 99th percentile : 1.5 ms [events: 1.5 ms] Latency 99.9th percentile : 3.5 ms [events: 3.5 ms] Latency max : 25.9 ms [events: 25.9 ms] Total partitions : 0 [events: 0] Total errors : 0 [events: 0] Total GC count : 0 Total GC memory : 0.000 KiB Total GC time : 0.0 seconds Avg GC time : NaN ms StdDev GC time : 0.0 ms Total operation time : 00:01:00 |
| --- |

**LCS read:**

| Results: Op rate : 8,878 op/s [events: 8,878 op/s] Partition rate : 0 pk/s [events: 0 pk/s] Row rate : 0 row/s [events: 0 row/s] Latency mean : 0.4 ms [events: 0.4 ms] Latency median : 0.3 ms [events: 0.3 ms] Latency 95th percentile : 0.7 ms [events: 0.7 ms] Latency 99th percentile : 1.3 ms [events: 1.3 ms] Latency 99.9th percentile : 2.6 ms [events: 2.6 ms] Latency max : 23.4 ms [events: 23.4 ms] Total partitions : 0 [events: 0] Total errors : 0 [events: 0] Total GC count : 0 Total GC memory : 0.000 KiB Total GC time : 0.0 seconds Avg GC time : NaN ms StdDev GC time : 0.0 ms Total operation time : 00:01:00 |
| --- |