ReadPathForUsers

Read Path

This section provides an overview of the Cassandra Read Path for developers who use Cassandra. Cassandra developers, who work on the Cassandra source code, should refer to the Architecture Internals developer documentation for a more detailed overview.

For the sake of brevity and clarity the 'read path' description below ignores consistency level and explains the 'read path' using a single local coordinator and a single replica node. The impact of consistency level of the 'read path' is discussed further down the page, under the Consistency Level topic.

CassandraReadPath.png|alt=Cassandra Read Path|width=800 title=Cassandra Read Path|width=800!

The Local Coordinator

The local coordinator receives the read request from the client and performs the following:

1. The local coordinator determines which nodes are responsible for storing the data:
    - The first replica is chosen based on the Partitioner hashing the primary key
    - Other replicas are chosen based on replication strategy defined for the keyspace
2. The local coordinator sends a read request to the fastest replica. All Cassandra snitches utilise the dynamic snitch to monitor read latency between nodes and maintain a list of the fastest responding replicas (or more accurately, the snitch calculates and maintains a 'badness score' per node, and read requests are routed to nodes with the lowest 'badness' score).

Replica Node

The replica node receives the read request from the local coordinator and checks whether the requested row exists in the row cache. The row cache is a read-through cache and will only contain the requested data if it was previously read.

Requested data in row cache

1. If the row is in the row cache, return the data to the local coordinator. Since the row cache already contains fully merged data there is no need to check anywhere else for the data and the read request can now be considered complete.

Requested data not in row cache

The data must now be read from the SSTables and MemTable:

1. Check the bloom filter. The bloom filter is a structure that guarantees whether a row DOES NOT exist in an SSTable, but it is unable to guarantee whether a row DOES exist. If the bloom filter indicates the row does not exist in the SSTable, then we do not have to read that SSTable.
2. For each SSTable that must be read, check the key cache. Key cache entries point to the offset in the SSTable where our requested row data is located. If there is a key cache entry it saves us scanning the partition index to determine the location of the data.
3. If there is no key cache entry, we first read the partition summary to obtain an offset into the partition index for the partition key our client is requesting. The partition summary does not contain an entry for each partition key (that's the job of the partition index); instead, it contains a sampling of the partition index and provides a starting point within the partition index for us to start scanning for our partition key.
4. We then scan the partition index, which will provide the offset in the SSTable where we can find the data associated with our partition key.
5. We then read the data from the in-memory MemTable, and merge this with the data from the SSTables. Data is merged cell by cell, with the timestamp for each cell being compared and the latest timestamp selected. Tombstones are ignored.
6. Update the row cache with the merged data if we are using the row cache.
7. The merged data is returned to the local coordinator.

Performance Features

Since there are potentially many physical SSTables for a single Cassandra table, Cassandra implements a number of performance features to speed up reads that must access the SSTables. These performance features reduce the necessity of reading every SSTable in order to construct the current state for a specific partition key:

- Bloom filters guarantee that a row does not exist in a particular SSTable, making it unnecessary to read that SSTable.
- Key cache contains the offset of a partition key in the SSTable, making it unnecessary to read the partition summary and partition index.
- Partition summary contains an offset to start scanning the partition index, making it unnecessary to read the entire partition index.
- Partition index contains an offset of a partition key in the SSTable, making it unnecessary to scan the entire SSTable.
- Row cache contains the latest, merged state of a row, making it unnecessary to read SSTables or MemTable.

Can't we treat the MemTable as a cache?

It might seem an unnecessary overhead to read data from both MemTable and SSTables if the data for a partition key exists in the MemTable. After all, doesn't the MemTable contain the latest copy of the data for a partition key? It depends on the type of operation applied to the data. Consider the following examples:

1. If a new row is inserted, this new row will exist in the MemTable and will only exist in an SSTable when the MemTable is flushed. Prior to the flush the bloom filters will indicate that the row does not exist in any SSTables, and the row will therefore be read from the MemTable only (assuming it has not previously been read and exists in the row cache).

2. After the MemTable containing the new row is flushed to disk, the data for this partition key exists in an SSTable only, and no longer exists in the MemTable. The flushed MemTable has been replaced by a new, empty MemTable.
3. If the new row is now updated, the updated cells will exist in the MemTable. However, not every cell will exist in the MemTable, only those that were updated (the MemTable is never populated with the current values from the SSTables). To obtain a complete view of the data for this partition key both the MemTable and SSTable must be read.

A MemTable is therefore a write-back cache that temporarily stores a copy of data by partition key, prior to that data being flushed to durable storage in the form of SSTables on disk. Unlike a true write-back cache (where changed data exists in the cache only), the changed data is also made durable in a Commit Log so the MemTable can be rebuilt in the event of a failure.

Consistency Level and the Read Path

Cassandra's tunable consistency applies to read requests as well as writes. The consistency level determines the number of replica nodes that must respond before the results of a read request can be sent back to the client; by tuning the consistency level a user can determine whether a read request should return fully consistent data, or whether stale, eventually consistent data is acceptable. The diagram and explanation below describe how Cassandra responds to read requests where the consistency level is greater than ONE. For clarity this section does not consider multi data-centre read requests.

CassandraReadConsistencyLevel.png|alt=Cassandra Read Consistency Level|width=800 title=Cassandra Read Consistency Level|width=800!

1. The local coordinator determines which nodes are responsible for storing the data:
    - The first replica is chosen based on the Partitioner hashing the primary key
    - Other replicas are chosen based on replication strategy defined for the keyspace
2. The local coordinator sends a read request to the fastest replica.
3. The fastest replica performs a read according to the 'read path' described above.
4. The local coordinator sends a 'digest' read request to the other replica nodes; these nodes calculate a hash of the data being requested and returns this to the local coordinator.
5. The local coordinator compares the hashes from all replica nodes. If they match it indicates that all nodes contain exactly the same version of the data; in this case the data from the fastest replica is returned to the client.
6. If the digests do not match then a conflict resolution process is necessary:
    - Read data from all replica nodes (with the exception of the fastest replica, as this has already responded to a full read request) according to the 'read path' described above.
    - Merge the data cell by cell based on timestamp. For each cell, the data from all replicas is checked and the most recent timestamp wins.
    - Return the merged data to the client.
    - The local coordinator sends a read-repair request to all out-of-sync replicas to update their data based on the merged data

You can see from the above description that each replica node has to respond to three different types of read request:

- A normal request for data
- A digest read request
- A read repair request