Apache Cassandra Materialized View Design

Overview:

A materialized view in Cassandra is, feature-wise, a very simplistic version of a traditional RDBMS materialized view. Specifically, it only allows a linear transformation of a single base cassandra table to a different cassandra table. Like a RDBMS materialized view it cannot be written to directly. It's data will be derived from the base table. Unlike an RDBMS materialized view, it does not handle complex query joins in its definition.

In this document we will cover the specific design decisions and failure modes for materialized views.

Abbreviations:

MV - Materialized View
C* - Cassandra
BL - BatchLog
UE - UnavailableException
WTE - WriteTimeoutException
CAS - Compare and Swap

Definitions:

Base Table - The table that is used as the source of the materialized view.
View Table - The table that is the materialized view destination

High Level Design Goals:

- Maintain eventual consistency between base table and view tables
- Provide mechanisms to repair consistency between base and views
- Aim to keep convergence between base and view fast without sacrificing availability (low MTTR)

High Level Summary:

```
CREATE TABLE example_base (
      partition1 text,
      partition2 text,
      clustering1 text,
      clustering2 text,
      value1 text,
      PRIMARY KEY((partition1, partition2), clustering1, clustering2));
```

Take the table definition above, if we want to find a particular “value1” value we have no way of searching for it other than iterating all partitions in the database or we could use a secondary index on value1. The former is not a solution for OLTP, the latter is less of an problem but it still requires potentially querying all nodes in the cluster if the value has high cardinality. Instead, we want to build a projection of the base table that let’s us query for value1 as a partition key:

```
CREATE MATERIALIZED VIEW example_view AS
      SELECT * from example_base WHERE
            value1 IS NOT NULL AND
            partition1 IS NOT NULL AND
            partition2 IS NOT NULL AND
            clustering1 IS NOT NULL AND
            clustering2 IS NOT NULL
      PRIMARY KEY(value1, partition1, partition2, clustering1,
      clustering2)
```

Now we can query example_view directly for any value1. If we wanted to search for a range of value1 per partition we could have written the PRIMARY KEY as `((partition1, partition2), value1, clustering1, clustering2)`.

Views contain a copy of a subset or complete columns (other than the partition and clustering keys which are required).

When a view is created, the values that are currently in the base need to be copied from the base into the view. Any new values need to also start being propagated to the view.

When an update is made to a base table, it is possible that the view will contain rows which are no longer valid. This is because, even when no tombstones are made for the base, the values that make up the view's primary keys may have changed thus invalidating the old row in the view. In this case, we must create range tombstones to remove the old view row in addition to adding the new row. This inherently implies a read-before-write, as we need to know which values the old view row had, which we can get from the value of the base from before the update is made.

Example:

```
INSERT INTO example_base
      (partition1, partition2, clustering1, clustering2, value1)
      VALUES ("a", "b", "c", "d", "e");

               |      PK      |        CK         |  VALUE
-------------------------------------------------------------
example_base   |   "a"   "b"  |   "c"     "d"     |    "e"
example_view   |      "e"     |  "a" "b" "c" "d"  |
```

```
INSERT INTO example_base
      (partition1, partition2, clustering1, clustering2, value1)
      VALUES ("a", "b", "c", "d", "f");

                |        PK        |        CK           | VALUE
------------------------------------------------------------------
example_base    |    "a"     "b"   |     "c"      "d"    |  "e"   <-  not unique
example_base    |    "a"     "b"   |     "c"      "d"    |  "f"
example_view    |        "e"       |  "a"  "b"  "c" "d"  |        <-  range tombstone
example_view    |        "f"       |  "a"  "b"  "c" "d"  |
```

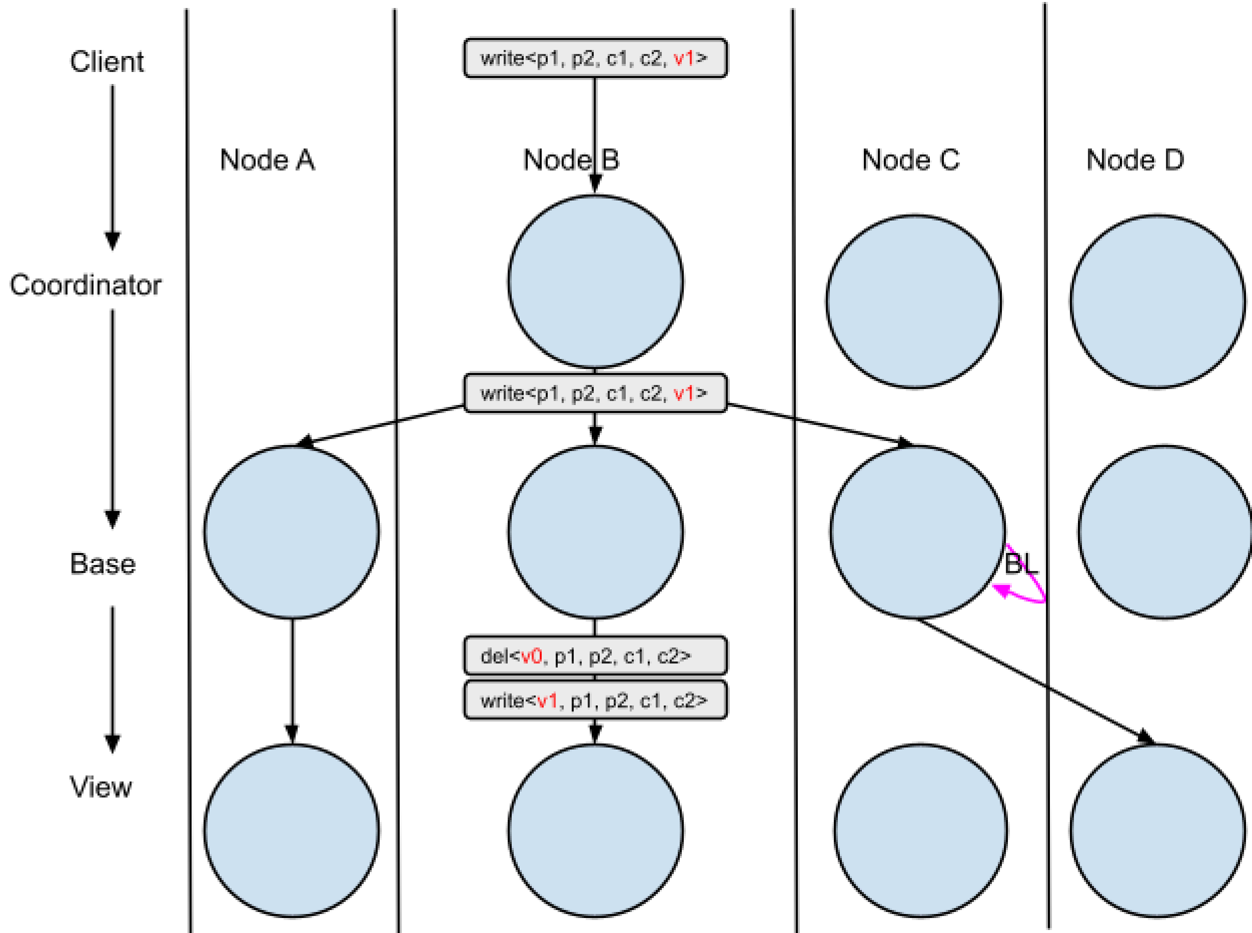
The old example_base record is replaced by the new due to PK uniqueness constraints. However, the old example_view record must be removed by a range tombstone applied after a read before write.

Materialized View Builder:

The builder is a compaction task, much like secondary index building. If this is the first time that the MV builder is running, it will flush the memtable and persist an entry into the materialized view builds table which references the sstable which was just flushed and marks the generations of sstables which will be included. All future sstables' contents will go through the write path, so there is no need to use the builder to write those values.

After each partition, a progress entry is written to the materialized view build in progress, so that if the builder is restarted it will not restart from the beginning. If the build is interrupted, it restarts after 5 minutes from the last checkpointed token. When the build is complete, a record is written to the materialized views built table. When a node is started, a new build will be kicked off; if there is a record of the build being done, we stop. If not, we will build from the last checkpointed token.

Write Path:



In the graphic above we depict the high level flow of a mutation through the cluster for a table with a materialized view on it (the one defined in the example above). We show what happens in a 4 node cluster with RF=3.

At the client:

There is no client level difference between a normal mutation and one that cascades to a materialized view. The server handles it transparently.

At the coordinator:

In `StorageProxy.mutateWithTriggers`, we see whether the mutation could modify a view. If it could not, we continue to apply the update as we normally would. If an update could modify a view, we write the mutation through the existing batchlog first with a special requirement that the batch must meet a consistency level of at least QUORUM. By adding this requirement we ensure faster convergence of the view.

At the base table replica:

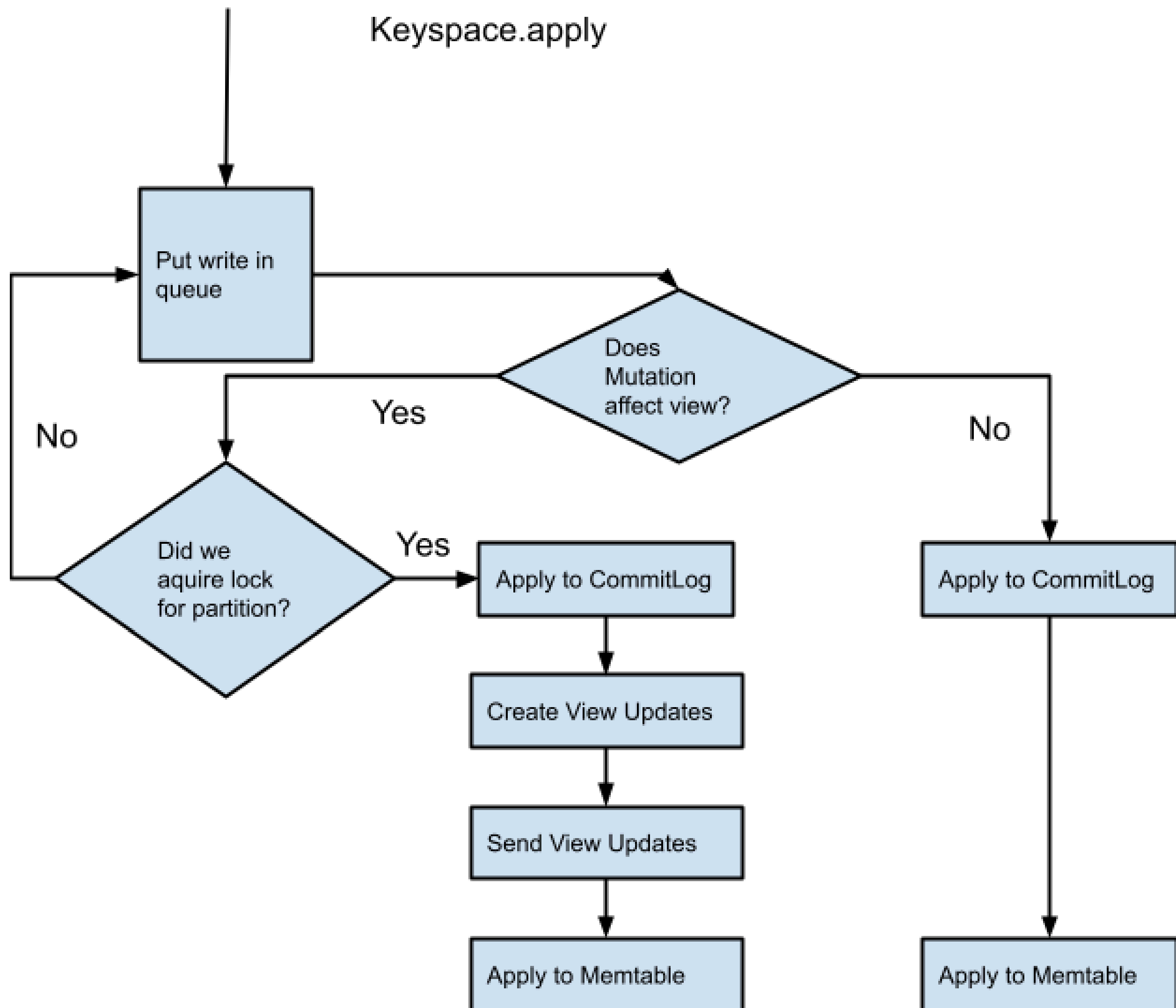
Once the mutation reaches the base table replica we transform each table row into it's materialized view as well as clean up any old values that no longer apply.

It is a requirement that we do not miss cleaning up any previous state. To do this we must apply only a single mutation to a base partition at a time. Without this assurance, it is possible that we would have two updates applied, each of which update the column which requires a tombstone to a different value, while both would get read a previous value which is no longer the most recently applied.

While a single CAS would be great, we don't read previous values off of disk in the CAS operation, only what is currently in the memtable. Since we will need to know what the previous value is before we apply the current value in order to push out the value to the view, we use a lock to ensure that we apply all values serially.

Once a partition has been locked we read the existing state off disk for the given partition and clustering keys and use that information to generate a complete picture of the existing base row vs the proposed change (which has yet to hit the memtable).

The specific order of operations is as follows (In *Keyspace.apply*)



The flow shows we try to acquire a partition lock before updating a base table with a view attached. If we can't access the lock because it's being used by another mutation, we simply put the mutation back onto the write queue (We check for timeouts based on how much time has elapsed since the instantiation of the object).

The main thing we want to protect against is losing the view updates, since we have written to the commit log and MUST update the memtable with the base mutation. Since the view replica is likely on a different node we can't guarantee we won't have a UE or WTE. Also if we send these messages synchronously we would block the time the mutation is being applied and effectively starve the system of mutation threads.

To avoid these issues we utilize the batchlog again to store the view mutations locally and then send the mutations async to the chosen replica. If the chosen replica is local then we can

bypass this step and write to the local view memtable (There is an exception to this rule if there are pending endpoints, then we still write to BL).

We only store the batchlog locally on the base table replica. This is safe since we have paired the view replica 1 to 1 with the base replica (explained in the next section). Since each base replica only cares about its local state and is paired with a single view replica then we can safely keep the batchlog local (If we lose the node we also lose the base state anyway).

All this happens in StorageProxy.mutateMV

View Replica Selection and Pairing

To cut down the number of replicas that we have to communicate, as well as providing the same consistency as the base update had, we only communicate with a single view replica. We want this replica to remain the same for a certain partition (as long as membership does not change in the cluster), so we need to have a consistent way to select the replica. If the local node is included in the list of replicas, it will be the view replica selected for the base. Otherwise, we select the replica which has the same cardinality for the view token as we have for the base token in the replication strategy, excluding replicas which overlap in the base and view tokens (this is to take into account the local replica pairings). Cardinality in replication factor is the number at which this node would start to include the key (if we would have the key with RF=3, but not RF=2, our cardinality is 3).

Read Path

The read path is the same one taken for any normal table. This allows for a fast retrieval with no fanout, since the base mutations have been copied to this table. Repair and read repair work as usual for the view. However it won’t reconcile inconsistencies from the base table only will make other view replicas consistent with each other. For base to view level repair we reply on the batchlog from the base mutation as well as regular repair of the base table.

Repair

Repairing the base table will also repair any view inconsistencies. We do this by streaming the inconsistent base table partitions based on the merkle tree mismatches as we normally do. But rather than simply swapping in the streamed sstables to the live view we iterate the sstable and apply each mutation through the normal write path. This ensures we have a serial view of the base table and generate any outstanding tombstones for the view. Once the mutations have been applied we discard the streamed sstable.

CommitLog Replay

Replaying the commitlog for a table with a materialized view attached is not complicated. We bypass all the materialized view handling and apply writes though the regular mutation path. We can do this because we have already stored the batchlog for the view updates which are guaranteed to eventually be written.

Multi-DC Support

When using a topology which has multiple data centers, the materialized view will select replicas which are local to the DC to prevent intra-DC latency. We upgrade the local DC batchlog to quorum, regardless of the DC. So each DC will get updates.

It might be necessary to change the batchlog implementation to accommodate multiple datacenters better.

Restrictions:

- No Static columns
- No Counter tables
- No MultiCell columns in Primary Keys

User Implications of using MV:

The main implication of using MVs for a user is knowing when the mutations will hit the View. As mentioned previously, synchronously waiting for replication from batch to view is not happening we must rely on the batchlog processing to handle this. If the load is low on the cluster than views should be updated quickly. We have added metrics to help operators know how many async view updates are failing Mention these here

We've added a new nodetool command 'replaybatchlog' to force replay of the nodes batchlog if the operator so chooses todo so.

Failure Modes:

- When a node goes down
- When a node is replaced
- When a disk is lost

These can be fixed by repairing the base table and the view.

- When a node is added
- When a node is decomm'd

When the topology of the ring is changed the base to view replica pairs change. Since the base tables ranges will be sent to their new node and streaming will apply the same logic as we describe in the 'repair' section and the new view replica will be updates. This may not really be required since repairing the view tables itself will bring consistency across the view.

Any outstanding Batchlog writes will still make it to the new node since the BL replay sends mutations to all replicas.

F.A.Q.

Why the coordinator batchlog?

Having read repair on the base table will also fix view inconsistencies but the problem is the view will likely be read over the base table so we are more likely to have persistent inconsistencies until we repair.

By having a batchlog for each base mutation we guarantee faster consistency between the base and the view since the batchlog will keep trying to apply the mutation until it achieves QUORUM.

The following would lead to inconsistency between the view and the base table:

- Coordinator receives writes, sends to replicas (but does not apply to batchlog)
- One base replica receives update, creates batchlog entry for view and applies locally
- View receives and applies update
- Coordinator dies, such that it doesn't detect that quorum was not achieved and cannot issue batchlog now
- Client does not retry the query, or dies here
- Base replica dies and cannot be restored; the replica batchlog and the base data is lost, repair cannot retrieve the mutation that should have been applied

Why don't you support sync updates?

- Sync updates would sacrifice availability which is something Cassandra doesn't like (esp. if you have many views to update)
- Blocking the base write to wait for the replica to be updated would starve the write workers under load (esp. if you have many views to update)
- Under a sync view update UE or WTE scenario, we've already written to the commit log for the base mutation which will never be applied to the memtable.