Materialized View Performance in Cassandra 3.x

BY JONATHAN ELLIS - MAY 10, 2016 | 8 COMMENTS

Materialized views (MV) landed in Cassandra 3.0 to simplify common denormalization patterns in Cassandra data modeling. This post will cover what you need to know about MV performance; for examples of using MVs, see Chris Batey's post here.

✓ How Materialized Views Work

Let's start with the example from Tyler Hobbs's introduction to data modeling:

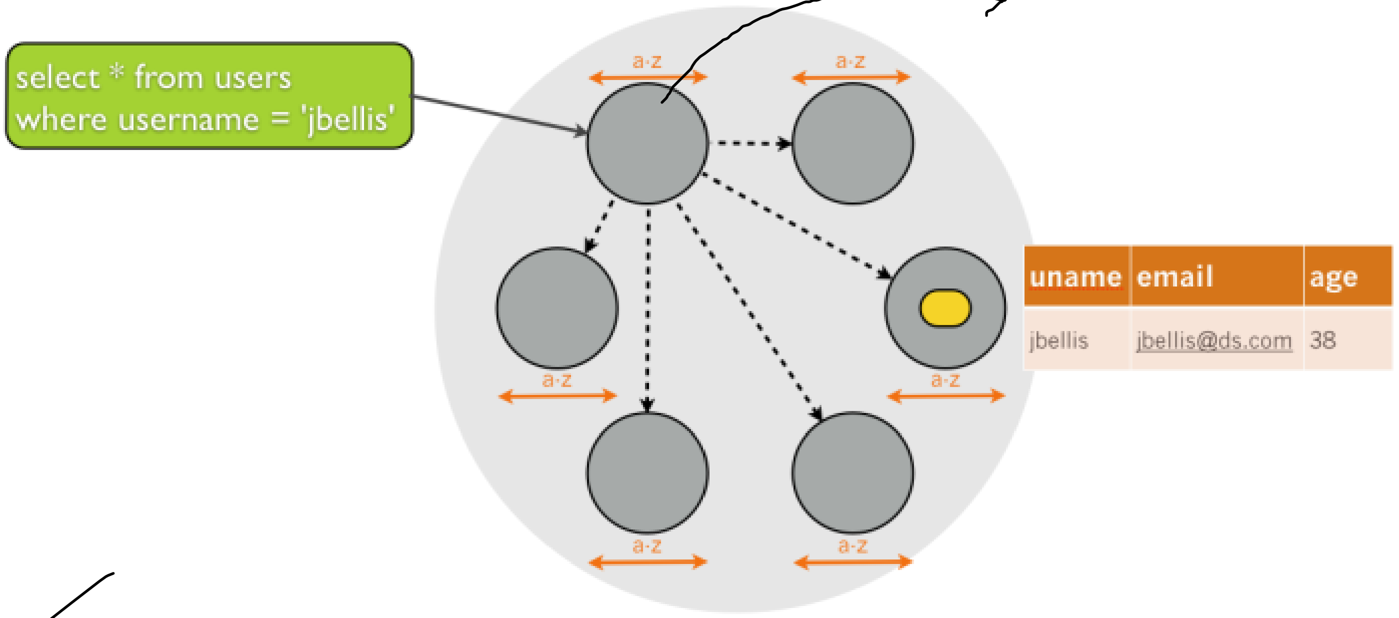
```
CREATE TABLE users (
    id uuid PRIMARY KEY,
    username text,
    email text,
    age int
);
```

We want to be able to look up users by username and by email. In a relational database, we'd use an index on the users table to enable these queries. With Cassandra, an index is a poor choice because indexes are local to each node. That means that if we created this index:

```
CREATE INDEX users_by_name ON users (username);
```

… a query that accessed it would need to fan out to each node in the cluster, and collect the results together. Put another way, even though the username field is unique, the coordinator doesn't know which node to find the requested user on, because the data is partitioned by id and not by name. Thus, each node contains a mixture of usernames across the entire value range (represented as a-z in the diagram):

This is the co-ordinator node which talks to all other nodes to find the username = 'jbellis'



uname	email	age
jbellis	jbellis@ds.com	38

This causes index performance to scale poorly with cluster size: as the cluster grows, the overhead of coordinating the scatter/gather starts to dominate query performance.

Thus, for performance-critical queries the recommended approach has been to denormalize into another table, as Tyler outlined:

```
CREATE TABLE users_by_name (
    username text PRIMARY KEY,
    id uuid
);
```

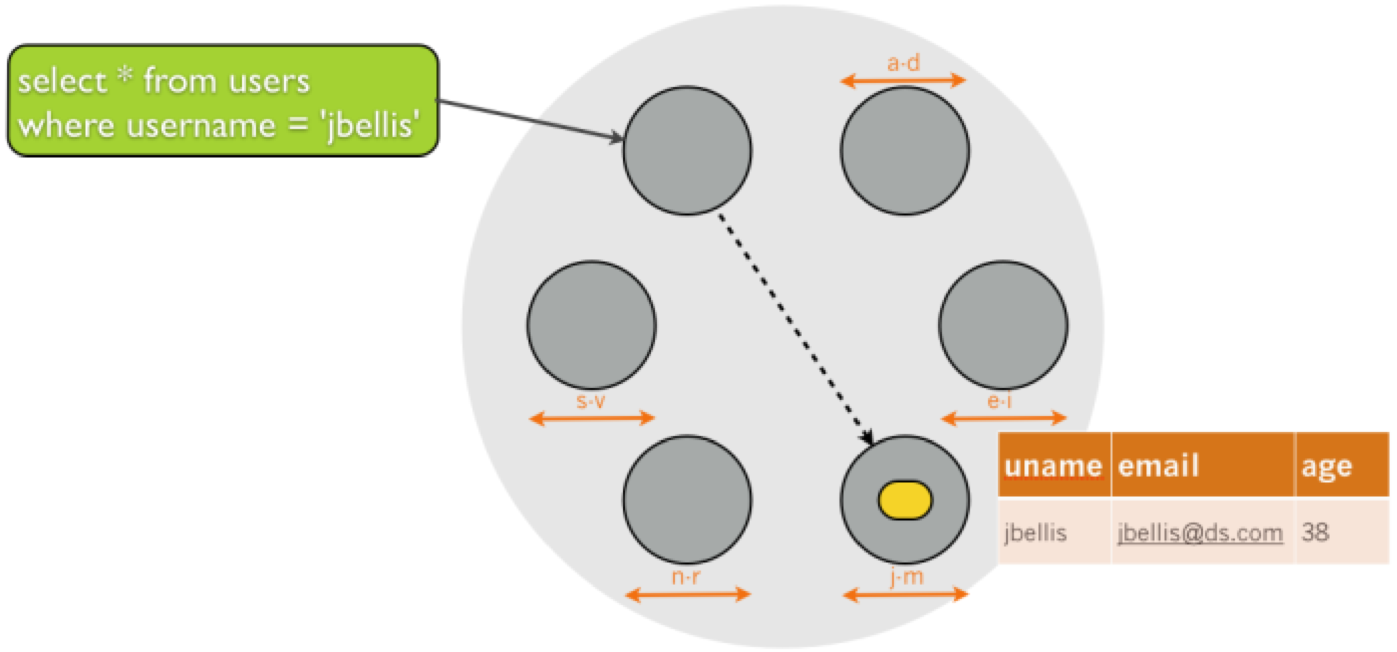
Now we can look look up users with a partitioned primary key lookup against a single node, giving us performance identical to primary key queries against the base table itself--but these tables must be kept in sync with the users table by application code.

Materialized views give you the performance benefits of denormalization, but are automatically updated by Cassandra whenever the base table is:

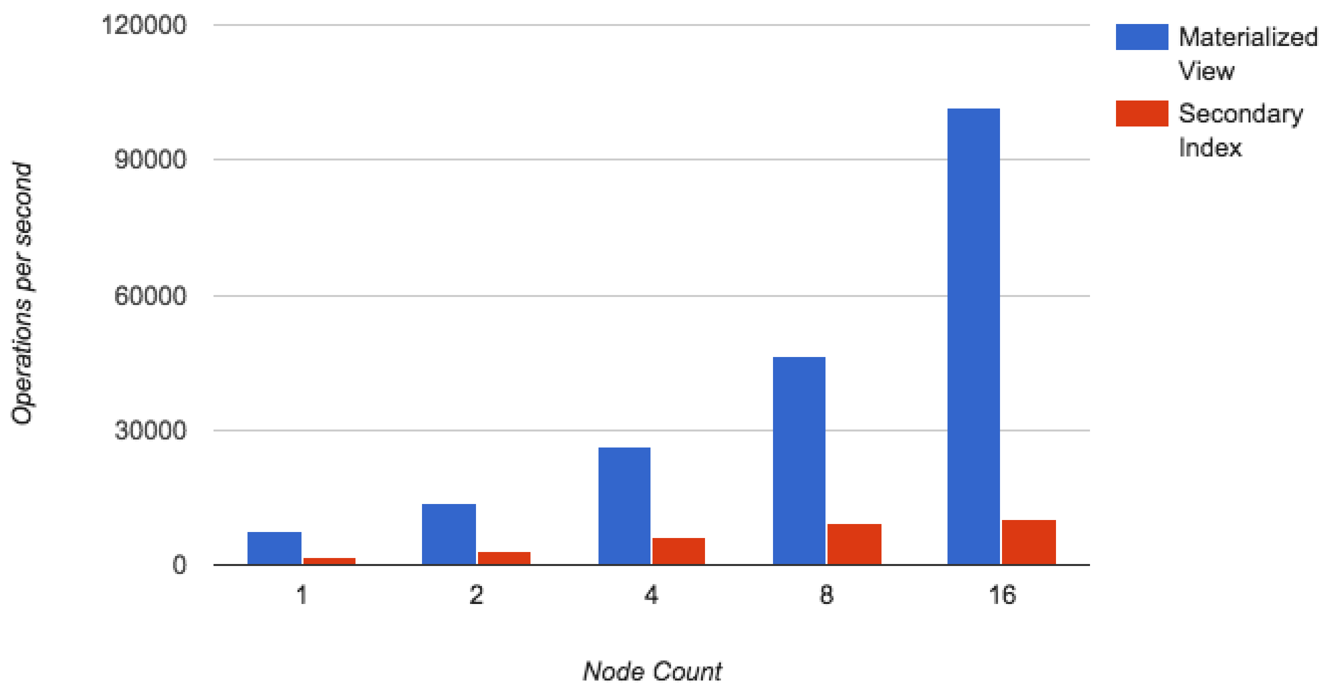
```
CREATE MATERIALIZED VIEW users_by_name AS
SELECT * FROM users
```

```
WHERE username IS NOT NULL
PRIMARY KEY (username, id);
```

Now the view will be repartitioned by username, and just as with manually denormalized tables, our query only needs to access a single partition on a single machine since that is the only one that owns the j-m username range:



The performance difference is dramatic even for small clusters, but even more important we see that indexed performance levels off when doubling from 8 to 16 nodes in the (AWS m3.xl) cluster, as the scatter/gather overhead starts to become significant:



Sidebar: When are Indexes Useful?

Indexes can still be useful when pushing analytical predicates down to the data nodes, since analytical queries tend to touch all or most nodes in the cluster anyway, making the primary advantage of materialized views irrelevant.

Indexes are also useful for full text search--another query type that often needs to touch many nodes--now that the new SASI indexes have been released.

Performance Impact of Materialized Views on Writes

What price do we pay at write time, to get this performance for reads against materialized views?

Recall that Cassandra avoids reading existing values on UPDATE. New values are appended to a commitlog and ultimately flushed to a new data file on disk, but old values are purged in bulk during compaction.

Materialized views change this equation. When an MV is added to a table, Cassandra is forced to read the existing value as part of the UPDATE. Suppose user jbellis wants to change his username to jellis:

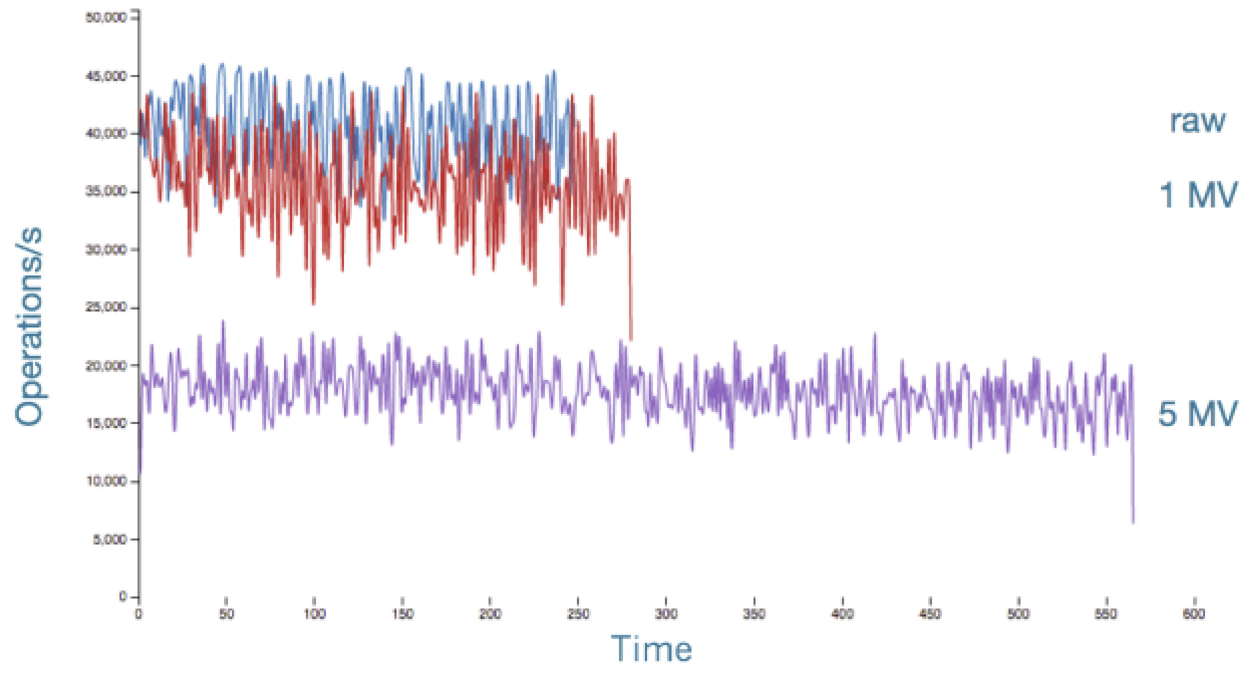
```
UPDATE users
SET username = 'jellis'
WHERE id = 'fcc1c301-9117-49d8-88f8-9df0cdeb4130';
```

Cassandra needs to fetch the existing row identified by fcc1c301-9117-49d8-88f8-9df0cdeb4130 to see that the current username is jbellis, and remove the jbellis materialized view entry.

(Even for local indexes, Cassandra does not need to read-before-write. The difference is that MV denormalizes the entire row and not just the primary key, which makes reads more performant at the expense of needing to pay the entire consistency price at write time.)

Materialized views also introduce a per-replica overhead of tracking which MV updates have been applied.

Added together, here's the performance impact we see adding materialized views to a table. As a rough rule of thumb, we lose about 10% performance per MV:

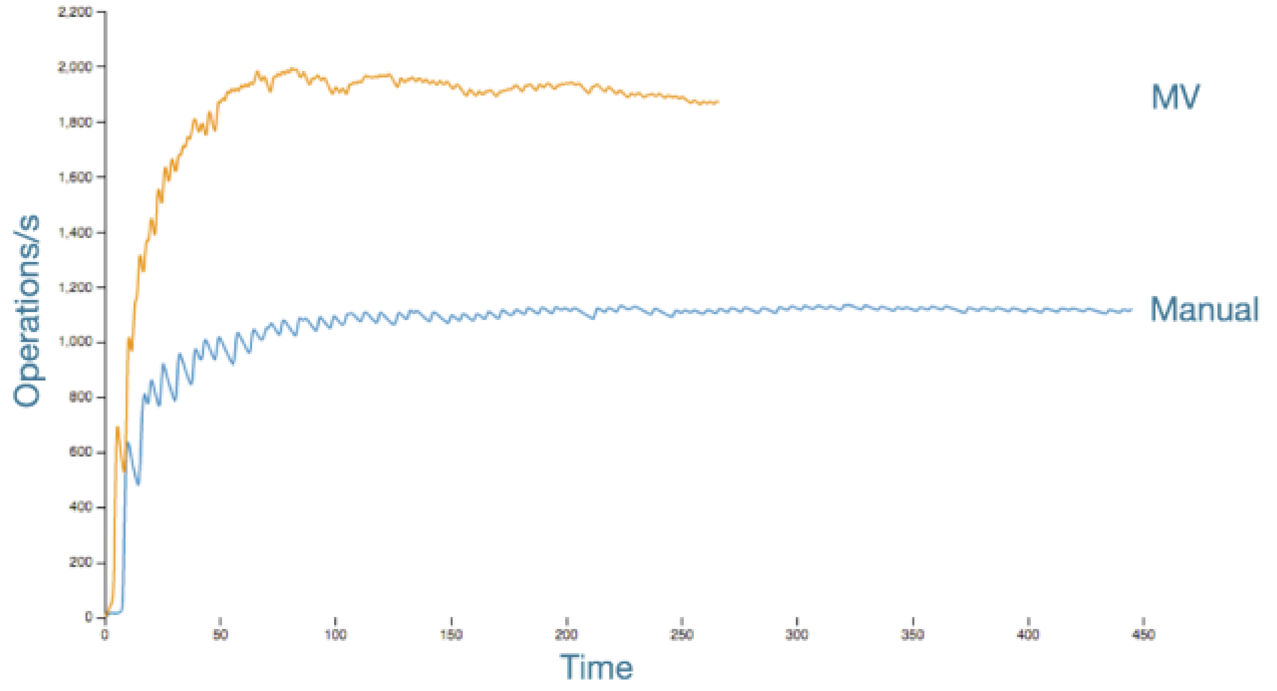


Materialized Views vs Manual Denormalization

Denormalization is necessary to scale reads, so the performance hits of read-before-write and batchlog are necessary whether via materialized view or application-maintained table. But can Cassandra beat manual denormalization?

We wrote a custom benchmarking tool to find out. mvbench compares the cost of maintaining four denormalizations for a playlist application for manual updates and MV.

Here’s what manual vs MV looks like in a 3 node, m4.xl ec2 cluster, RF=3, in an insert-only workload:



What we see is that after the initial JVM warmup, the manually denormalized insert (where we can “cheat” because we know from application logic that no prior values existed, so we can skip the read-before-write) hits a plateau and stays there. The MV, while faster on average, has performance that starts to decline from its initial peak.

To understand these results, we need to explain what the mvbench workload looks like. The data model is a table of playlists and four associated MV:

```
CREATE TABLE user_playlists
(
    user_name           text,
    playlist_name       text,
    song_id             text,
    added_time          bigint,
    artist_name         text,
    genre               text,
    last_played         bigint,
    PRIMARY KEY (user_name, playlist_name, song_id)
);
```

The MV created are song_to_user, artist_to_user, genre_to_user, and recently_played. For the sake of brevity I will show only the last:

```
CREATE MATERIALIZED VIEW IF NOT EXISTS mview.recently_played AS
      SELECT song_id, user_name
      FROM user_playlists
      WHERE song_id IS NOT NULL
      AND playlist_name IS NOT NULL
      AND user_name IS NOT NULL
      AND last_played IS NOT NULL
      PRIMARY KEY(user_name, last_played, playlist_name, song_id
);
```

What is important to note here is that the base `user_playlists` table has a compound primary key. What is happening to cause the deteriorating MV performance over time is that our sstable-based bloom filter, which is keyed by partition, stops being able to short circut the read-old-value part of the MV maintenance logic, and we have to perform the rest of the primary key lookup before inserting the new data.

MV Performance Summarized

As a general rule then, you can apply the following rules of thumb for MV performance:

- Reading from a normal table or MV has identical performance.
- Each MV will cost you about 10% performance at write time.
- For simple primary keys (tables with one row per partition), MV will be about twice as fast as manually denormalizing the same data.
- For compound primary keys, MV are still twice as fast for updates but manual denormalization can better optimize inserts. The crossover point where manual becomes faster is a few hundred rows per partition. CASSANDRA-9779 is open to address this limitation.

DataStax has many ways for you to advance in your career and knowledge.

You can take free classes, get certified, or read one of our many white papers.

REGISTER FOR CLASSES

GET CERTIFIED

DBA'S GUIDE TO NOSQL

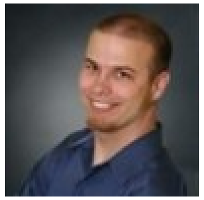
Comments

1. *Zhichun Wu* says:

   May 15, 2016 at 5:41 pm

   Great post! Why not making leveled compaction as default for materialized views, since there'll be many updates?

   Reply

2.  *Jonathan Ellis* says:

   May 16, 2016 at 6:32 am

First of all, the premise here is wrong: other things being equal, LCS is the wrong choice for update-intensive workloads because of its much higher write amplification compared to STCS.

That said, I'm not sure that you can infer anything about the relative frequency of updates to an MV in the general case.

Reply

3. *Tri Bui* says:

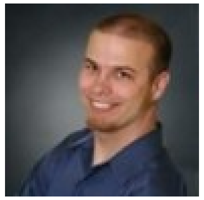
   May 26, 2016 at 2:08 pm

   There is a lot of articles addressing performance of the MV.

   What about storage implications? If you have rows with hundreds of fields and you have dozens of MVs, about how much storage overhead will it add?

   Thanks.

   Reply

4.  *Jonathan Ellis* says:

   May 26, 2016 at 2:11 pm

   Storage overhead is exactly the same as for manual denormalization.

   You can materialize as few or as many columns from the base table as are needed.

   Reply

5. Benjamin Roth says:

August 30, 2016 at 1:42 am

Ich guess the 10% cost is only true for MVs with the same PK as the base table.

I made the experience that a cluster, that runs smoothly and stable (with MVs that have the same PK as the base table) can easily be completely f***ed up by adding some MVs with a different PK. I guess this is due to the fact, that with a different PK, the MV write path is not local any more.

Some observations:
– In normal operations, the write latency nearly doubles
– Bulk operations like a repair stream or mass inserts drive the load massively up
– Running “a bit too much” heavy operations can kill a node completely. E.g. in my case a reaper-repair-job and a bulk insert job (~10k inserts/s) kills a node within about 15 minutes. Process looks like: Load goes steadily up => write timeouts occur => hints start to pile up => mutations are queuing up => heap is filled => node is GCed to death => total cluster response goes up

When all MVs have the same PK, then everything is still smooth. Even with heavy jobs and loads of writes. Server load still goes up and response times go slightly up but at the end the system remains stable.

Our cluster runs 6 nodes each 96GB RAM, 16 cores, 800GB SSD. Without MVs, that cluster is able to process 200k writes/s.

Reply

1. Nicolas Gullstrand says:

   October 16, 2016 at 11:47 pm

   PK? Primary Key or Partition Key?

This is an interesting observation but what is PK? To me it doesn't make sense to create a MV with same Primary Key as the base table.

Maybe something I miss. Please, elaborate since I'm right now choosing between manual redundancy and MV.

Reply

6. *Eugene Strokin* says:

   September 9, 2016 at 11:56 am

   What about wide rows in MV? I mean if I have millions of documents with the same username (as per your example), wouldn't this cause problems with memory or tombstones in frequent updates?

   Reply

7. *Rittick Gupta* says:

   May 14, 2017 at 5:37 pm

   Is there a limit to number of MV's for a table ?
   My application design would require 50MV's for a table – is this recommended ? Are there any technical issues that has to be considered ?

   Reply

Comments

Your email address will not be published. Required fields are marked *

Leave a comment...

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

Post Comment

Subscribe for newsletter:

Email Address

SUBMIT

« Python Driver 3.3.0 Released

DataStax Ruby Driver 3.0.0 Released! »

Offices

United States

France

Germany

Japan

United Kingdom

Contact

+1 (650) 389-6000

Connect

Newsletter

Subscribe