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:

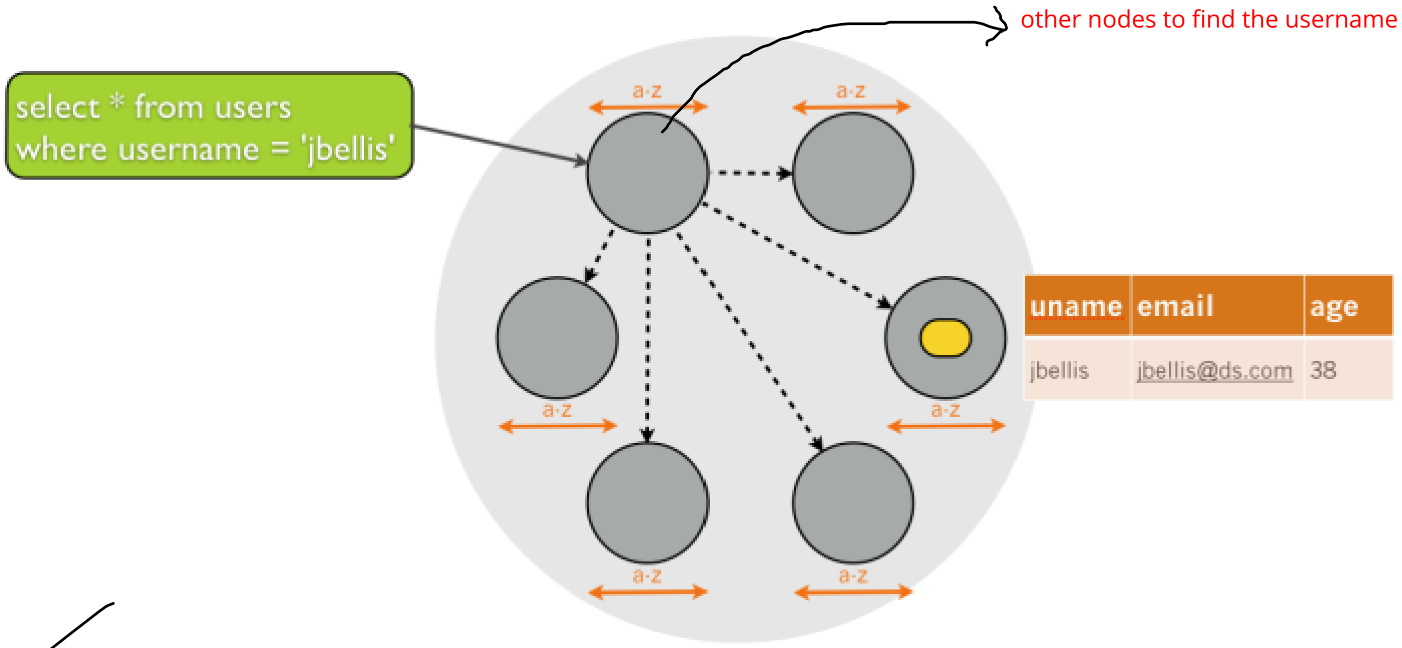
```
CREATE TABLE users (
    id uuid PRIMARY KEY,
    username text,
    email text,
    age int
);
```

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
);
```

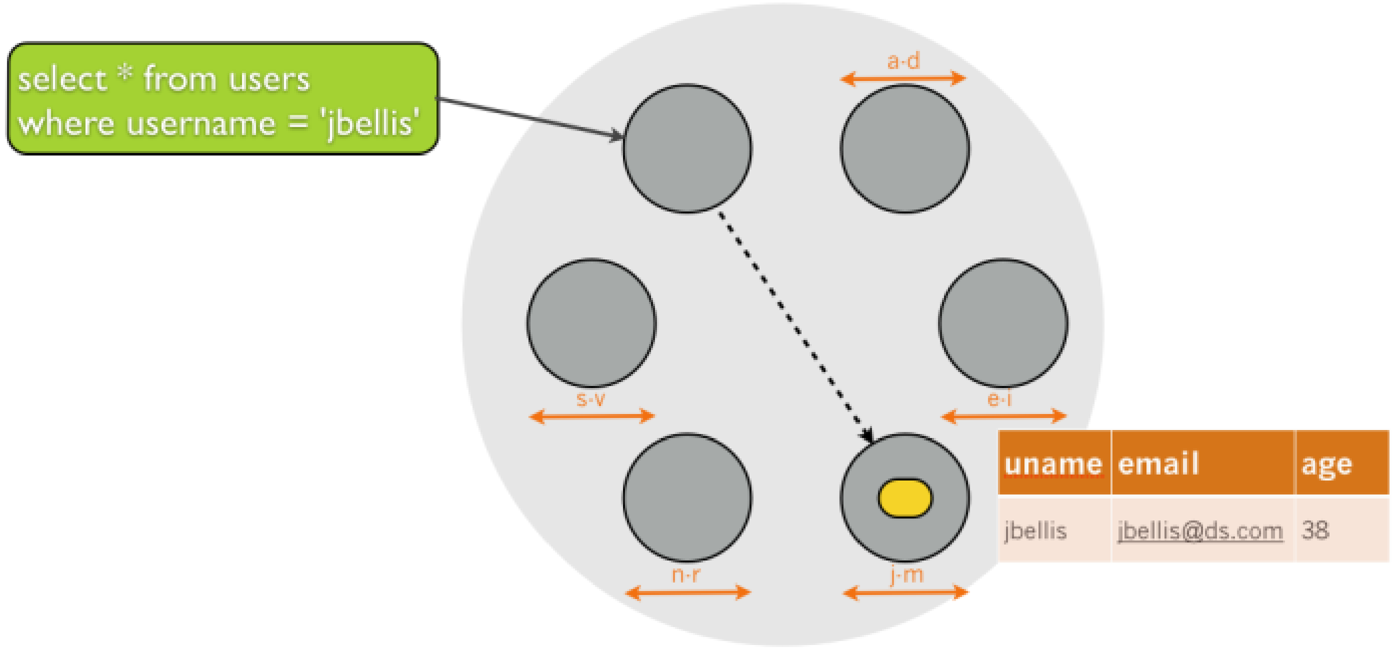
Now we can look look up users with a partitioned primary key lookup against a single node, giving us performance identical to primary key queries against the base table itself--but these tables must be kept in sync with the users table by application code.

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

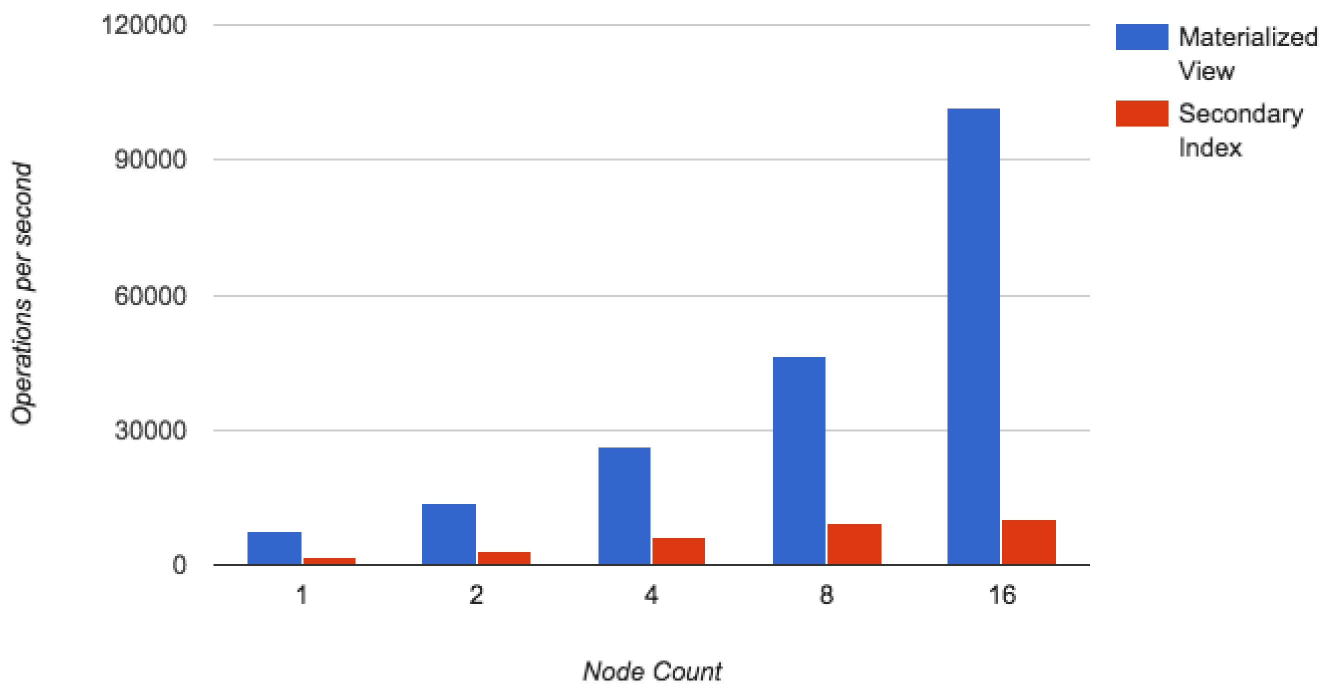
```
CREATE MATERIALIZED VIEW users_by_name AS
SELECT * FROM users
```

```
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: