The DataStax Blog

Subscribe to the RSS Feed

VIEWING
Company

Search Blog

February 2, 2015

Basic Rules of Cassandra Data Modeling

FILED IN: COMPANY

DataStax

For more recent data modeling content, check out the sample data models in the Data Modeling By Example learning series, as well as our Data Modeling in Apache Cassandra™ whitepaper.

Picking the right data model is the hardest part of using Cassandra. If you have a relational background, CQL will look familiar, but the way you use it can be very different. The goal of this post is to explain the basic rules you should keep in mind when designing your schema for Cassandra. If you follow these rules, you'll get pretty good performance out of the box. Better yet, your performance should scale linearly as you add nodes to the cluster.

Non-Goals

Developers coming from a relational background usually carry over rules about relational modeling and try to apply them to Cassandra. To avoid wasting time on rules that don't really matter with Cassandra, I want to point out some non-goals:

Minimize the Number of Writes

Writes in Cassandra aren't free, but they're awfully cheap. Cassandra is optimized for high write throughput, and almost all writes are equally efficient [1]. If you can perform extra

Minimize Data Duplication

network), and Cassandra is architected around that fact. In order to get the most efficient reads, you often need to duplicate data. Besides, Cassandra doesn't have JOINs, and you don't really want to use those in a distributed fashion.

Basic Goals

These are the two high-level goals for your data model:

1. Spread data evenly around the cluster
2. Minimize the number of partitions read

There are other, lesser goals to keep in mind, but these are the most important. For the most part, I will focus on the basics of achieving these two goals. There are other fancy tricks you can use, but you should know how to evaluate them, first.

Rule 1: Spread Data Evenly Around the Cluster

You want every node in the cluster to have roughly the same amount of data. Cassandra makes this easy, but it's not a given. Rows are spread around the cluster based on a hash of the partition key, which is the first element of the PRIMARY KEY. So, the key to spreading data evenly is this: pick a good primary key. I'll explain how to do this in a bit.

Rule 2: Minimize the Number of Partitions Read

Partitions are groups of rows that share the same partition key. When you issue a read query, you want to read rows from as few partitions as possible. Why is this important? Each partition may reside on a different node. The coordinator will generally need to issue separate commands to separate nodes for each partition you request. This adds a lot of overhead and increases the variation in latency. Furthermore, even on a single node, it's more expensive to read from multiple partitions than from a single one due to the way rows

Conflicting Rules?

If it's good to minimize the number of partitions that you read from, why not put everything in a single big partition? You would end up violating Rule #1, which is to spread data evenly

Model Around Your Queries

The way to minimize partition reads is to model your data to fit your queries. Don't model around relations. Don't model around objects. Model around your queries. Here's how you do that:

Step 1: Determine What Queries to Support

Try to determine exactly what queries you need to support. This can include a lot of considerations that you may not think of at first. For example, you may need to think about:

- Grouping by an attribute
- Ordering by an attribute
- Filtering based on some set of conditions
- Enforcing uniqueness in the result set
- etc ...

Changes to just one of these query requirements will frequently warrant a data model change for maximum efficiency.

Step 2: Try to create a table where you can satisfy your query by reading (roughly) one partition

In practice, this generally means you will use roughly one table per query pattern. If you need to support multiple query patterns, you usually need more than one table. To put this another way, each table should pre-build the "answer" to a high-level query that you need to support. If you need different types of answers, you usually need different tables. This is how you optimize for reads. Remember, data duplication is okay. Many of your tables may repeat the same data.

Applying the Rules: Examples

Example 1: User Lookup

The high-level requirement is "we have users and want to look them up". Let's go through the steps: Step 1: Determine what specific queries to support Let's say we want to either be able to look up a user by their username or their email. With either lookup method, we should get the full set of user details. Step 2: Try to create a table where you can satisfy your query by reading (roughly) one partition Since we want to get the full details for the user with either lookup method, it's best to use two tables:

```
CREATE TABLE users_by_username ( username text PRIMARY KEY, email
```

Now, let's check the two rules for this model: Spreads data evenly? Each user gets their own partition, so yes. Minimal partitions read? We only have to read one partition, so yes. Now, let's suppose we tried to optimize for the non-goals, and came up with this data model instead:

```
CREATE TABLE users ( id uuid PRIMARY KEY, username text, email tex
```

This data model also spreads data evenly, but there's a downside: we now have to read two partitions, one from users_by_username (or users_by_email) and then one from users. So reads are roughly twice as expensive.

Example 2: User Groups

Now the high-level requirement has changed. Users are in groups, and we want to get all users in a group. Step 1: Determine what specific queries to support We want to get the full user info for every user in a particular group. Order of users does not matter. Step 2: Try to create a table where you can satisfy your query by reading (roughly) one partition How do we fit a group into a partition? We can use a compound PRIMARY KEY for this:

```
CREATE TABLE groups ( groupname text, username text, email text, a
```

Note that the PRIMARY KEY has two components: groupname, which is the partitioning key, and username, which is called the clustering key. This will give us one partition per

```
SELECT * FROM groups WHERE groupname = ?
```

This satisfies the goal of minimizing the number of partitions that are read, because we only need to read one partition. However, it doesn't do so well with the first goal of evenly spreading data around the cluster. If we have thousands or millions of small groups with hundreds of users each, we'll get a pretty even spread. But if there's one group with millions of users in it, the entire burden will be shouldered by one node (or one set of replicas). If we want to spread the load more evenly, there are a few strategies we can use. The basic technique is to add another column to the PRIMARY KEY to form a compound partition key. Here's one example:

```
CREATE TABLE groups ( groupname text, username text, email text, a
```

The new column, hash_prefix, holds a prefix of a hash of the username. For example, it could be the first byte of the hash modulo four. Together with groupname, these two columns form the compound partition key. Instead of a group residing on one partition, it's now spread across four partitions. Our data is more evenly spread out, but we now have to read four times as many partitions. This is an example of the two goals conflicting. You need to find a good balance for your particular use case. If you do a lot of reads and groups don't get too large, maybe changing the modulo value from four to two would be a good choice. On the other hand, if you do very few reads, but any given group can grow very large, changing from four to ten would be a better choice. There are other ways to split up a partition, which I will cover in the next example. Before we move on, let me point out something else about this data model: we're duplicating user info potentially many times, once for each group. You might be tempted to try a data model like this to reduce duplication:

```
CREATE TABLE users ( id uuid PRIMARY KEY, username text, email tex
```

Obviously, this minimizes duplication. But how many partitions do we need to read? If a group has 1000 users, we need to read 1001 partitions. This is probably 100x more expensive to read than our first data model. If reads need to be efficient at all, this isn't a good model. On the other hand, if reads are extremely infrequent, but updates to user info (say, the username) are extremely common, this data model might actually make sense. Make sure to take your read/update ratio into account when designing your schema.

Example 3: User Groups by Join Date

```
CREATE TABLE group_join_dates ( groupname text, joined timeuuid, u
```

Here we're using a timeuuid (which is like a timestamp, but avoids collisions) as the clustering column. Within a group (partition), rows will be ordered by the time the user joined the group. This allows us to get the newest users in a group like so:

```
SELECT * FROM group_join_dates WHERE groupname = ? ORDER BY joined
```

This is reasonably efficient, as we're reading a slice of rows from a single partition. However, instead of always using ORDER BY joined DESC, which makes the query less efficient, we can simply reverse the clustering order:

```
CREATE TABLE group_join_dates ( groupname text, joined timeuuid, u
```

Now we can use the slightly more efficient query:

```
SELECT * FROM group_join_dates WHERE groupname = ? LIMIT ?
```

As with the previous example, we could have problems with data being spread evenly around the cluster if any groups get too large. In that example, we split partitions somewhat randomly, but in this case, we can utilize our knowledge about the query patterns to split partitions a different way: by a time range. For example, we might split partitions by date:

```
CREATE TABLE group_join_dates ( groupname text, joined timeuuid, j
```

We're using a compound partition key again, but this time we're using the join date. Each day, a new partition will start. When querying the X newest users, we will first query today's partition, then yesterday's, and so on, until we have X users. We may have to read multiple partitions before the limit is met. To minimize the number of partitions you need to query, try to select a time range for splitting partitions that will typically let you query only one or two partitions. For example, if we usually need the ten newest users, and groups usually acquire three users per day, we should split by four-day ranges instead of a single day [2].

The basic rules of data modeling covered here apply to all (currently) existing versions of Cassandra, and are very likely to apply to all future versions. Other lesser data modeling problems, such as dealing with tombstones, may also need to be considered, but these problems are more likely to change (or be mitigated) by future versions of Cassandra. Besides the basic strategies covered here, some of Cassandra's fancier features, like collections, user-defined types, and static columns, can also be used to reduce the number of partitions that you need to read to satisfy a query. Don't forget to consider these options when designing your schema. Hopefully I've given you some useful fundamental tools for evaluating different schema designs. If you want to go further, I suggest taking Datastax's free, self-paced online data modeling course (DS220). Good luck!

Footnotes

[1]: Notable exceptions: counters, lightweight transactions, and inserting into the middle of a list collection. [2]: I suggest using a timestamp truncated by some number of seconds. For example, to handle four-day ranges, you might use something like this:

```
now = time() four_days = 4 * 24 * 60 * 60 shard_id = now - (now %
```

AUTHORED BY

Tyler Hobbs

Microsoft Azure
Google Cloud Platform
Amazon Web Services
VMware

MOST POPULAR

Spin Up a Cluster
Your First Cassandra Query
Workshops
Get Certified
Inspired Execution
Sample Apps to Get Started

Docs
Support
Academy
Community
Tools & Downloads
Professional Services

RESOURCES

Blog
Podcasts
Webinars
NoSQL
Cassandra
Cloud-Native
Resource Library
Legal
Security

United States