Cassandra Data Modeling

- Basic Rules of Cassandra Data Modeling

Non-Goals

1. Minimize the Number of Writes
2. Minimize Data Duplication

Basic Goals

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

These two goals often conflict, so you'll need to try to balance them.

Model Around Your Queries

1. Determine What Queries to Support
2. Try to create a table where you can satisfy your query by reading (roughly) one partition

Applying the Rules: Examples

Example 1: User Lookup

The high-level requirement is "we have users and want to look them up".

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.

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 text,
    age int
)

CREATE TABLE users_by_email (
    email text PRIMARY KEY,
    username text,
    age int
)
```

- 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 text,
    age int
)

CREATE TABLE users_by_username (
    username text PRIMARY KEY,
    id uuid
)

CREATE TABLE users_by_email (
    email text PRIMARY KEY,
    id uuid
)
```

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.

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.

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,
    age int,
    PRIMARY KEY (groupname, username)
)
```

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 groupname. Within a particular partition (group), rows will be ordered by username. Fetching a group is as simple as doing the following:

```
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,
    age int,
    hash_prefix int,
    PRIMARY KEY ((groupname, hash_prefix), username)
)
```

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.

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 text,
    age int
)

CREATE TABLE groups (
    groupname text,
    user_id uuid,
    PRIMARY KEY (groupname, user_id)
)
```

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

Suppose we continue with the previous example of groups, but need to add support for getting the X newest users in a group.

We can use a similar table to the last one:

```
CREATE TABLE group_join_dates (
    groupname text,
    joined timeuuid,
    username text,
    email text,
    age int,
    PRIMARY KEY (groupname, joined)
)
```

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 DESC
    LIMIT ?
```

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,
    username text,
    email text,
    age int,
    PRIMARY KEY (groupname, joined)
) WITH CLUSTERING ORDER BY (joined DESC)
```

```
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,
    join_date text,
    username text,
    email text,
    age int,
    PRIMARY KEY ((groupname, join_date), joined)
) WITH CLUSTERING ORDER BY (joined DESC)
```

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