Getting Started with User Profile Data Modeling

White Paper

What was old is new again **3**
Fixed schema – User Data Pattern 1 ..... **3**
Fixed schema on a wide row – User Data Pattern 2 ..... **3**
Fixed Schema with a dynamic collection – User Data Pattern 3 ..... **5**
Conclusion **6**

What was old is new again

Creating static schema in Cassandra may seem unusual, but is frequently done in production data models. Not every table created needs to have dynamic columns. In some cases it's useful to have the data stored in fixed column names with type validation. The following examples will get you started with defining static schema tables and in some cases create hybrid data models that add a dynamic column component for greater flexibility.

Fixed schema – User Data Pattern 1

To start, we'll work on a very basic table of data that you might find in any relational database. If you have created a table in MySQL or Oracle, this should look very familiar to you. In the syntax below, the *username* is the only primary key, so we put the declaration to the right of field type. You can use this reference to see the different types that can be used: http://www.datastax.com/docs/1.1/references/cql/cql_data_types

```
CREATE TABLE users (
   username text PRIMARY KEY,
   first_name text,
   last_name text,
   address1 text,
   city text,
   postal_code text,
   last_login timestamp
);
```

Once we have our table definition, it should be as equally as familiar to insert data using the INSERT INTO syntax.

```
INSERT INTO users
(username,first_name,last_name,address1,city,postal_code,last_login)
VALUES ('cstar','Cassandra','Database','123 Main St','San
Mateo','11111','2013-4-4');
```

Once the data is in, you can use the SELECT command.

```
SELECT first_name, last_name
FROM users
WHERE username = 'cstar';
```

One large difference between relational databases and Cassandra CQL is the WHERE clause syntax. Because of how the data is stored, you can only use the elements declared in the primary key. Unless you want every row, you need to specify the row key at a minimum. We'll use the next example to dive further into this difference.

Fixed schema on a wide row – User Data Pattern 2

For the this pattern, we'll again be storing what looks like static row oriented data. The subtle difference here is in how the data is stored by Cassandra. By using multiple fields in the PRIMARY KEY definition, we are specifying that this data will be stored in a wide row. Let's use a playlist model as an example:

```
CREATE TABLE user_playlist (
      username text,
      playlist_name text,
      song_name text,
      artist text,
      list_order int,
      songid UUID
      PRIMARY KEY (username, playlist_name, song_name)
);
```

At first glance, it would appear that we are creating a table that will contain multiple rows with 3 unique fields. The first element 'username', will be used as the row key. The remaining elements will be be used by CQL to create unique columns in the underlying storage engine. The result set presented to the user is a large table of data similar to a relational database. Let's insert some data and see how it works:

```
INSERT INTO user_playlist (username, playlist_name, song_name, artist,
list_order, songid)

VALUES ('cstar','Classic Rock','Stairway to heaven','Led
Zepplin',1,c9df5407-7dd8-46a4-84ee-1c358106c926);

INSERT INTO user_playlist (username, playlist_name, song_name, artist,
list_order, songid)

VALUES ('cstar','Classic Rock','Sweet home Alabama','Lynyrd
Skynyrd',2,d2fb151c-cba8-46ff-bc03-66b32c45434a);

INSERT INTO user_playlist (username, playlist_name, song_name, artist,
list_order, songid)

VALUES ('cstar','Alternative','The Queen is dead','The
Smiths',1,24106aa0-10d5-4de1-9dab-a3a652e32c49);

INSERT INTO user_playlist (username, playlist_name, song_name, artist,
list_order, songid)

VALUES ('cstar','Alternative','Blue Monday','New Order',2,7a0b82ba-
5a24-4554-903c-1970d1e3aaea);
```

All of the data inserted will be on one row with the key '*cstar*', however, we we access that data with CQL, it will appear as many rows.

```
SELECT playlist_name,song_name,artist,list_order from user_playlist

WHERE username='cstar';

playlist_name | song_name          | artist         | list_order
---------------+--------------------+----------------+------------
   Alternative |        Blue Monday |      New Order |          2
   Alternative | The Queen is dead |     The Smiths |          1
  Classic Rock | Stairway to heaven |    Led Zepplin |          1
  Classic Rock | Sweet home Alabama | Lynyrd Skynyrd |          2
```

The output is nicely formatted and easier to read by the user. The underlying storage engine has actually created a single row of data using unique column names derived from the PRIMARY KEY definition. The result is a faster access pattern on reads when all data is co-located on the same node and sequentially read from disk. You can read more in-depth here: http://www.datastax.com/docs/1.2/ddl/table

Fixed Schema with a dynamic collection – User Data Pattern 3

When you need some dynamic element to the static table, this can be accomplished by using the collection feature of CQL. A collection can be a Map, Set or List and can be read about more here: http://www.datastax.com/dev/blog/cql3_collections

The example we'll use is a user table with an added requirement to dynamically store one or more locations sorted by time. The map collection can be used to create an ordered pair of data where the key is a timestamp and the value is a location stored as text.

```
CREATE TABLE user_with_location (
      username text PRIMARY KEY,
      first_name text,
      last_name text,
      address1 text,
      city text,
      postal_code text,
      last_login timestamp,
      location_by_date map<timestamp,text>
);
```

First we insert the entire user record. Note: When inserting a collection, you will overwrite any of the previous values if the row key exists.

```
INSERT INTO user_with_location
(username,first_name,last_name,address1,city,postal_code,last_login,loc
ation_by_date)

VALUES ('cstar','Cassandra','Database','123 Main St','San
Mateo','11111','2013-4-4',{ '2013-4-4' : 'San Francisco'});
```

Then we add a couple of locations for this user. This will be an update so not overwrite the previous location data in the collection. If you use the same map key as a previous data point, you will overwrite just that map value.

```
UPDATE user_with_location SET location_by_date['2013-4-5'] = 'San
Diego'
WHERE username = 'cstar';
UPDATE user_with_location SET location_by_date['2013-4-2'] = 'Santa
Rosa'
WHERE username = 'cstar';
```

When we select the row of data for the user '*cstar*', the result set will contain the entire map of locations. Collections have to be returned completely, so keep that in mind as you design your model. Too much in one collection could cause a lot of data to be returned.

```
SELECT first_name, last_name, location_by_date
FROM user_with_location
WHERE username = 'cstar';
first_name | last_name | location_by_date
------------+-----------+-------------------------------------
  Cassandra |  Database | {2013-04-02 00:00:00-0700: Santa Rosa,
                           2013-04-04 00:00:00-0700: San Francisco,
                           2013-04-05 00:00:00-0700: San Diego}
```

You might also notice that the data in the first element of the map is sorted. This is a feature of the map to sort by the type of the key.

Conclusion

As you can see, creating user based schema can be simple and with some of the CQL features shown, powerful in creating wide row data sets. Cassandra has had the reputation of only fitting large dynamic data sets, but with CQL you can create fixed schema and strong validation typing. There are many other patterns for user profile management. These three should get you started on your user centered data model.