Lab 5. Effective CQL

In this lab, we will examine common approaches to data modeling and interacting with data stored in Apache Cassandra. This will involve us taking a close look at the Cassandra Query Language, otherwise known as **CQL**. Specifically, we will cover and discuss the following topics:

- The evolution of CQL and the role it plays in the Apache Cassandra universe
- How data is structured and modeled effectively for Apache Cassandra
- How to build primary keys that facilitate high-performing data models at scale
- How CQL differs from SQL
- CQL syntax and how to solve different types of problems using it

Once you have completed this lab, you should have an understanding of why data models need to be built in a certain way. You should also begin to understand known Cassandra anti-patterns and be able to spot certain types of bad queries. This should help you to build scalable, query-based tables and write successful CQL to interact with them.

In the parts of this lab that cover data modeling, be sure to pay extra attention. The data model is the most important part of a successful, high-performing Apache Cassandra cluster. It is also extremely difficult to change your data model later on, so test early, often, and with a significant amount of data. You do not want to realize that you need to change your model after you have already stored millions of rows. No amount of performance-tuning on the cluster side can make up for a poorly-designed data model!

An overview of Cassandra data modeling

Understanding how Apache Cassandra organizes data under the hood is essential to knowing how to use it properly. When examining Cassandra's data organization, it is important to determine which version of Apache Cassandra you are working with. Apache Cassandra 3.0 represents a significant shift in the way data is both stored and accessed, which warrants a discussion on the evolution of CQL.

Before we get started, let's create a keyspace for this lab's work:

```
CREATE KEYSPACE fenago_ch3 WITH replication =
{'class': 'NetworkTopologyStrategy', 'ClockworkAngels':'1'};
```

To preface this discussion, let's create an example table. Let's assume that we want to store data about a music playlist, including the band's name, albums, song titles, and some additional data about the songs. The CQL for creating that table could look like this:

```
CREATE TABLE playlist (
 band TEXT,
 album TEXT,
 song TEXT,
 running_time TEXT,
 year INT,
 PRIMARY KEY (band,album,song));
```

Now we'll add some data into that table:

```
INSERT INTO playlist (band,album,song,running_time,year)
  VALUES ('Rush','Moving Pictures','Limelight','4:20',1981);
INSERT INTO playlist (band,album,song,running_time,year)
  VALUES ('Rush','Moving Pictures','Tom Sawyer','4:34',1981);
INSERT INTO playlist (band,album,song,running_time,year)
  VALUES ('Rush','Moving Pictures','Red Barchetta','6:10',1981);
INSERT INTO playlist (band,album,song,running_time,year)
  VALUES ('Rush','2112','2112','20:34',1976);
INSERT INTO playlist (band,album,song,running_time,year)
  VALUES ('Rush','Clockwork Angels','Seven Cities of Gold','6:32',2012);
INSERT INTO playlist (band,album,song,running_time,year)
  VALUES ('Coheed and Cambria','Burning Star IV','Welcome Home','6:15',2006);
```

Cassandra storage model for early versions up to 2.2

The original underlying storage for Apache Cassandra was based on its use of the Thrift interface layer. If we were to look at how the underlying data was stored in older (pre-3.0) versions of Cassandra, we would see something similar to the following:

Figure 3.1: Demonstration of how data was stored in the older storage engine of Apache Cassandra. Notice that the data is partitioned (co-located) by its row key, and then each column is ordered by the column keys.

As you can see in the preceding screenshot, data is simply stored by its row key (also known as the **partitioning key**). Within each partition, data is stored ordered by its column keys, and finally by its (non-key) column names. This structure was sometimes referred to as a **map of a map**. The innermost section of the map, where the column values were stored, was called a **cell**. Dealing with data like this proved to be problematic and required some understanding of the Thrift API to complete basic operations.

When CQL was introduced with Cassandra 1.2, it essentially abstracted the Thrift model in favor of a SQL-like interface, which was more familiar to the database development community. This abstraction brought about the concept known as the **CQL row**. While the storage layer still viewed from the simple perspective of partitions and column values, CQL introduced the row construct to Cassandra, if only at a logical level. This difference between the physical and logical models of the Apache Cassandra storage engine was prevalent in major versions: 1.2, 2.0, 2.1, and 2.2.

Cassandra storage model for versions 3.0 and beyond

On the other hand, the new storage engine changes in Apache Cassandra 3.0 offer several improvements. With version 3.0 and up, stored data is now organized like this:

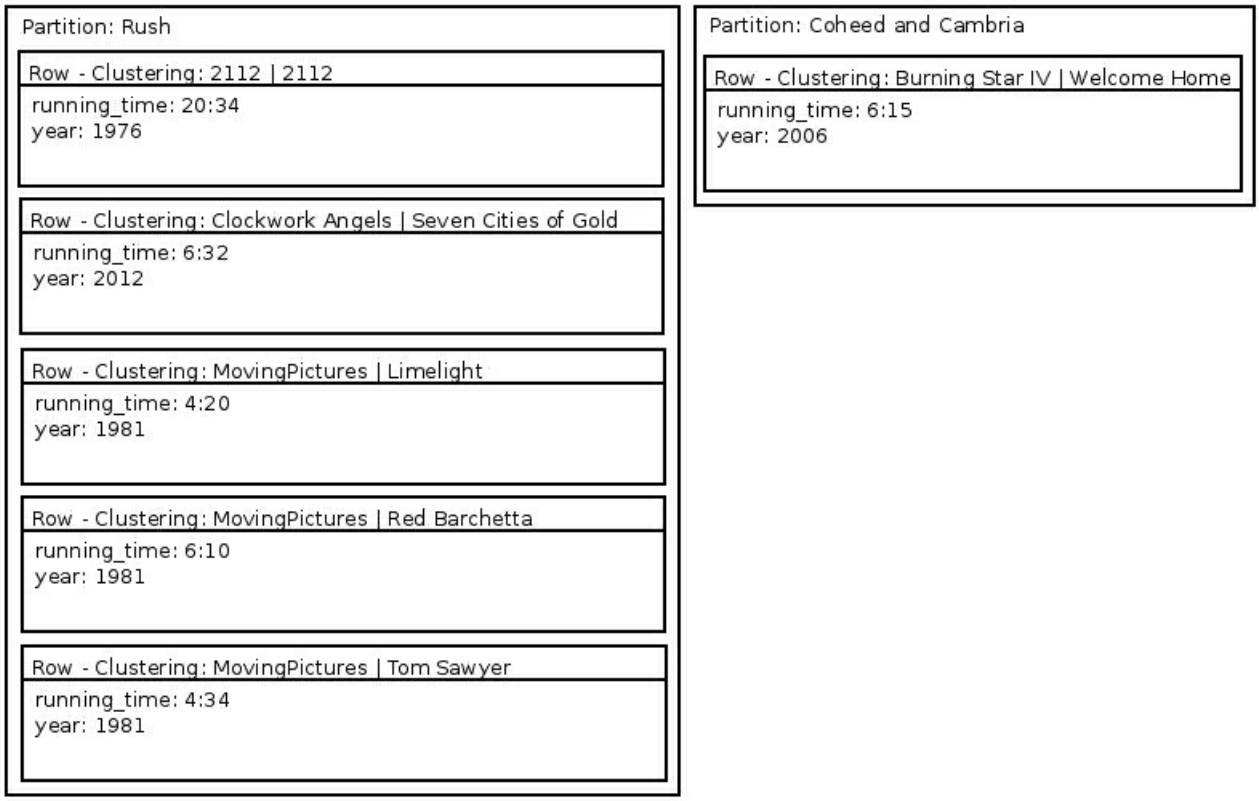

Figure 3.2: Demonstration of how data is stored in the new storage engine used by Apache Cassandra 3.0 and up. While data is still partitioned in a similar manner, rows are now first-class citizens.

The preceding figure shows that, while data is still partitioned similarly to how it always was, there is a new structure present. The row is now part of the storage engine. This allows for the data model and the Cassandra language drivers to deal with the underlying data similar to how the storage engine does.

Note

An important aspect not pictured in the preceding screenshot is the fact that each row and column value has its own timestamp.

In addition to rows becoming first-class citizens of the physical data model, another change to the storage engine brought about a drastic improvement. As Apache Cassandra's original data model comes from more of a key/value approach, every row is not required to have a value for every column in a table.

The original storage engine allowed for this by repeating the column names and clustering keys with each column value. One way around repeating the column data was to use the `WITH COMPACT STORAGE` directive at the time of table creation. However, this presented limitations around schema flexibility, in that columns could no longer be added or removed.

Note

Do not use the `WITH COMPACT STORAGE` directive with Apache Cassandra version 3.0 or newer. It no longer provides any benefits, and exists so that legacy users have an upgrade path.

With Apache Cassandra 3.0, column names and clustering keys are no longer repeated with each column value. Depending on the data model, this can lead to a drastic difference in the disk footprint between Cassandra 3.0 and its prior versions.

Note

I have seen as much as a 90% reduction in disk footprint by upgrading from Cassandra 2.x to Cassandra 3.0 and as little as 10% to 15%. Your experience may vary, depending on the number of columns in a table, size of their names, and primary key definition.

Data cells

Sometimes structures for storing column values are referred to as **cells**. Queries to Cassandra essentially return collections of cells. Assume the following table definition for keeping track of weather station readings:

```
CREATE TABLE weather_data_by_date (
  month BIGINT,
  day BIGINT,
  station_id UUID,
  time timestamp,
  temperature DOUBLE,
  wind_speed DOUBLE,
  PRIMARY KEY ((month,day),station_id,time));
```

In this model, the `month` and `day` keys are used to make up a composite partition key. The clustering keys are `station_id` and `time`. In this way, for each partition, the number of cells will be equal to the total unique combinations of tuples:

- `station_id`, `time`, `temperature`
- `station_id`, `time`, `wind_speed`

Understanding a cell comes into play when building data models. Apache Cassandra can only support 2,000,000,000 cells per partition. When data modelers fail to consider this, partitions can become large and ungainly, with query times eventually getting slower. This is why data models that allow for unbound row growth are considered to be an anti-pattern.

Note

The maximum of 2,000,000,000 cells per partition is a hard limit. But, practically speaking, models that allow that many cells to be written to a single partition will become slow long before that limit is reached.

Getting started with CQL

With some quick definitions of CQL data modeling and cqlsh completed, now we'll take a look at CQL. The basic commands for creating data structures (keyspaces, tables, and so on) will be covered right away, with command complexity increasing as we build more useful structures.

Creating a keyspace

Keyspaces are analogous to the logical databases of the relational world. A keyspace contains tables, which are usually related to each other by application, use case, or development team. When defining a keyspace, you also have the ability to control its replication behavior, specifying pairs of data center names and a numeric **Replication Factor** (**RF**).

Creating a keyspace is a simple operation and can be done like this:

```
CREATE KEYSPACE [IF NOT EXISTS] <keyspace_name>
 WITH replication =
 {'class': '<replication_strategy>',
  '<data_center_name>':'<replication_factor>'}
 AND durable_writes = <true/false>;
```

Here is the detailed explanation of the preceding query:

- `keyspace_name` : Valid keyspace names must be composed of alphanumeric characters and underscores.
- `replication_strategy` : Either `SimpleStrategy` or `NetworkTopologyStrategy` .
- `data_center_name` : Valid only for `NetworkTopologyStrategy` , must be the name of a valid data center. If using `SimpleStrategy` , specify `replication_factor` .
- `replication_factor` : A numeric value representing the number of replicas to write for the key with which it is paired.
- `durable_writes` : A Boolean indicating whether writes should be written to the commit log. If not specified, this defaults to `true` .

Note

Names for keyspaces, tables, columns, and custom structures in Apache Cassandra must be alphanumeric. The only exception to that rule is that an underscore (`_`) is the only valid special character that can be used.

Single data center example

The following example will create the `fenago_ch3` keyspace. When a write occurs, one replica will be written to the cluster. Writes will also be sent to the commit log for extra durability:

```
CREATE KEYSPACE IF NOT EXISTS fenago_ch3
 WITH replication =
 {'class': 'NetworkTopologyStrategy', 'ClockworkAngels':'1'}
 AND durable_writes = true;
```

Note

`SimpleStrategy` isn't very useful, so my advice is not to use it. It's a good idea to get into the habit of using `NetworkTopologyStrategy` , as that's the one that should be used in production. `SimpleStrategy` offers no advantages over `NetworkTopologyStrategy` , and using `SimpleStrategy` can complicate converting into a multi-data center environment later.

Multi-data center example

The following example will create the `fenago_ch3b` keyspace, as long as it does not already exist. When a write occurs, two replicas will be written to the `ClockworkAngels` data center, and three will be written to the `PermanentWaves` and `MovingPictures` data centers. Writes will also be sent to the commit log for extra durability:

```
CREATE KEYSPACE fenago_ch3_mdc
 WITH replication = {'class': 'NetworkTopologyStrategy',
  'ClockworkAngels':'2', 'MovingPictures':'3', 'PermanentWaves':'3'}
 AND durable_writes = true;
```

Once you have created your keyspace, you can use it to avoid having to keep typing it later. Notice that this will also change your Command Prompt:

```
 cassdba@cqlsh> use fenago_ch3 ;
cassdba@cqlsh:fenago_ch3>
```