Big Data systems - Apache Cassandra Assignment

Stamatis Pitsios, BAPT1502

M.Sc. Business Analytics, Part Time 2015 – 2016

7/11/2016

Table of Contents

Abstract

The purpose of this assignment is the design and the implementation of a database system using Apache Cassandra, in order to support a simple music streaming service.

Apache Cassandra is a free and open-source distributed database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. Cassandra offers robust support for clusters spanning multiple datacenters, with asynchronous masterless replication allowing low latency operations for all clients [2].

Cassandra is a NoSQL database management system and its data model is a partitioned row store with tunable consistency. Rows are organized into tables; the first component of a table's primary key is the partition key; within a partition, rows are clustered by the remaining columns of the key. Other columns may be indexed separately from the primary key. [2]

Cassandra is implemented via CQL (Cassandra Query Language), which is very close to normal SQL. However, the hardest part in Cassandra, is the data modeling. The way it works is completely different from relational database systems when coming to design. We will try to explain as best as possible some common goals and non-goals of Apache Cassandra [3]. 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, let's point out some *non*-goals:

1. 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. If we can perform extra writes to improve the efficiency of our read queries, it's almost always a good tradeoff. Reads tend to be more expensive and are much more difficult to tune.

2. Minimize data duplication
Denormalization and duplication of data is a fact of life with Cassandra. We should not be afraid of it. Disk space is generally the cheapest resource (compared to CPU, memory, disk IOPs, or network), and Cassandra is architected around that fact. In order to get the most efficient reads, we often need to duplicate data.

Besides, Cassandra doesn't have JOINs, and you don't really want to use those in a distributed fashion.

When coming to goals, there are two very important rules to keep in mind.

1. Spread the data evenly around the cluster
We 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.

2. Minimize the number of partitions read
Partitions are groups of rows that share the same partition key. When we issue a read query, we 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 we 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 are stored.

Those 2 rules often conflict with each other. We can clearly understand this if we try to answer the question: "If it's good to minimize the number of partitions that you read from, why not put everything in a single big partition?". Because would end up violating Rule #1, which is to spread data evenly around the cluster.

Keeping all those things in mind, we can proceed with the data modeling. To achieve this, we should, first of all, determine the queries what we want to support. Then we create the necessary tables needed to answer each query, by applying the non-goals, as well as the basic goals.

The problem

As we already mentioned earlier, we would like to build a system using Cassandra in order to support a music streaming service. This service stores information about users, songs and playlists. In particular, for each song we need to store:

- Song name
- Artist
- Album
- Genre
- Year
- Play count
- The song binary file

For each user we need to store:

- Name
- Address
- Payment information
- Complete history of songs played
- Users that the specific user "follows"

For each playlist we need to store:

- Name
- Description
- Genre
- Creator
- Songs in the playlist
- Followers

In a relational database, we would firstly design the tables and then we would try to answer the different queries. In Cassandra, we will start with the queries and then design the tables. In the section that follows, we describe the queries that we would like to support, as well as the procedure that we followed in order to result to a specific design of the tables.

Queries

After analyzing the requirements, it was found that the service needs to support 13 queries, which are listed below.

Query 1 – Search for users by their names

For this query we could design a table which contains everything about a user as we mentioned in description. However, the history of the songs of a user, as well as its followers, are asked in queries 3 and 8 respectively. Since as rule of thumb we create a different table for each query, at this point we only want table that keeps the user's name, it's address, as well as it's payment information. We can make the assumption that since this is an online service, the users provide a username which is unique. Consequently, we will have an attribute user_name, which is going to be the primary and the partition key.

Address is a complex attribute that can have the street name, the city, the name of state or province, the postal code and finally the country. Cassandra allows us to create custom types that contain many attributes. For our needs, we can create a custom type named address which contains all the information mentioned above. To add a custom type in a Cassandra table, we need to put it inside a *frozen<>* declaration.

For the payment information we can follow the same approach as we did for address. We can create a custom type which stores the payment type (Visa, Master Card, American Express), the number of the card, the card's expiration date, as well as the card's security code. In a real case scenario, we should be careful to store the card's number as well as it's code encrypted, but at this point we are only interested in the design of the tables. Since a user can provide many payment information, we can store those in a map, where the key is name provide by the user, e.g. "My Visa Card" and the value is the actual payment information.

The table that we explained can be seen in the *Table 1* below.

users_by_name
user_name text
address frozen<address>
payment_information map<text, frozen<payment_information>
Primary Key (user_name)

Table 1

Query 2 – Search for songs by their names

For this query we can follow exact the same logic for Query 1. We would like to get information about a song by providing its name. We can use all the information about a song, but for now we will keep the play count attribute aside, since this is requested in Query 11. The name of the song can't be considered unique, so as a primary key we can use the name of the song together with the artist. The song's name will be the partitioning key and the artist name the clustering one. The table can be seen in *Table 2*.

songs_by_name
song_name text
artist text
album text
year smallint
genre text
song_file blob
Primary Key ((song_name), artist)

Table 2

Query 3 – Find the songs played by a user, in reverse chronological order

For this query we would like to provide a username and get in return all the songs that the specific user has listened to, in reverse chronological order. A first approach would be to create a table with two attributes, the username together with a list of songs that have been played by the user. However, collections in Cassandra can store up to 64K items, and theoretically a user may have listened to more than that. For this reason, we will create a table that stores the username and the information of songs played. Minimizing the number of writes and data duplication are non-goals as we mentioned earlier, so we do not mind to repeat information about the songs. Our approach can be seen in *Table 3*.

Users_and_songs
user_name text
song_name text
artist text
album text
year smallint
genre text
played_on timestamp
Primary Key ((user_name), played_on)

Table 3

As a primary key we use the username as well as the time that the specific user listened to a song, since a user cannot have listened to two different songs at the same time. We also define a decreasing clustering order in the played_on attribute, so we can retrieve the songs sorted in decreasing order of time played. With this design we achieve the two goals that we want. First of all, since the user_name is the partition key, the data will be evenly spread among the clusters because every year will reside in its own cluster. Secondly, when we will perform the necessary query, we will read at most one partition if we provide a user name, the one that the specific user will reside in.

Query 4 – Search for playlists by their names

For this query, we will use the same approach as for queries 1 and 2. We want to create a table to look for playlist information based on its name. However, we do not want all of the attributes. The songs that are contained in the playlist, as well as the followers of the playlist are needed later in other queries, so we will exclude them for now. For this table, using just the playlist name as a primary key, we achieve the two basic goals. Design can be seen in *Table 4*.

playlist_by_name
playlist_name text
description text
genre text
user_name text
Primary Key (playlist_name)

Table 4

Query 5 – Search for playlists by their genre

In order to support this query, it is not necessary to create a new table. One could use the previous one and provide a where clause to filter on the genre attribute. Nevertheless, Cassandra does not allow to use a where condition on attributes that are not part of the partition key. There are two ways to address this issue. We can either add a secondary index on the genre attribute, or creating a materialized view using genre as a partition key. Secondary indexes are more suitable for columns with low cardinality. Music genre can be limited in tens of different values, such as pop, rock, jazz etc. This makes it an ideal case for creating a secondary index on it. After the creation, we can use the queries wanted without the need of creating a new table.

Query 6 – Search for playlists by their creator

Playlists creator on the other hand have a high cardinality, a lot higher than the genre. This means that can create a materialized view based on *Table 4* that defines as the partition key the playlist's creator and as a clustering key the playlist's name. Again we create a copy of the original table to support our queries, without the need to create a completely new table.

Query 7 – Find the followers of a playlist

To be able to answer this this query, we can use almost the same approach as we did with Query 3. We just can't add a playlist name and a list of followers, since lists can only contain up to 64K records. For this reason, we are going to store tuples of the form (playlist_name, follower_name). This design can help us once again to achieve our goals and non-goals. The table's design can be seen in *Table 5*.

playlist_follower
playlist_name text
follower_name text
Primary Key ((playlist_name), follower_name)

Table 5

Query 8 – Find the followers of a user

We can approach this query by using the exact same logic as for Query 7. We will create a table that stores tuples of the form (user_name, follower_name). *Table 6* reflects this design.

user_follower
user_name text
follower_name text
Primary Key ((user_name), follower_name)

Table 6

Query 9 – Find the songs contained in a playlist

One more action that we would like to support, is to give the name of a playlist and get back all the songs contained in it. Again we come upon the limitation that Cassandra collections can store up to 64K items. Despite the fact that it is not realistic that a playlist can have more than 64K songs, it would better to design a table to address this issue, just for the case. The table to support this query is going to have a partition key the name of the playlist, in order to spread our data equally among the nodes, based on this name. By doing this we also achieve the goal of reading only partition per query, the one that contains the data for the specific playlist. For each playlist, we will also store the information of the songs that it contains. *Table 7* shows the necessary design.

songs_by_playlist
playlist_name text
song_name text
artist text
album text
year smallint
genre text
Primary Key ((playlist_name), song_name, artist_name)

Table 7

Another requirement that we did not mentioned, is that we want the songs to be sorted by their names in increasing order. That's why we add a clustering key consisting of the song_name and the artist_name as well. As we mentioned earlier, the song_name is not unique and that's why we also added the name of the artist. We then define an ascending clustering order to the song name and in cases of the same name, we use an ascending order as well in the artist's name.

This table is a good example to demonstrate the conflicting rules that can appear in our goals. If we have a lot of songs and only a few playlists that contain a lot of songs, then we do not have an equal data spread, since some partitions will be burdened with more records. We could try to add an additional attribute to the partition key, in order to get a better split [3]. This however will cause the problem that we would now need to read data from many partitions to answer a query, violating goal no. 2. Thus we must be very careful with the design we follow. In our case we can expect to have many different playlists, so hopefully, our modeling will not cause any problems.

Query 10 – Find how many times a playlist has been played

Now we are going to see something that we haven't come upon so far. Counting how many times a specific event has occurred. Cassandra support this by a special data modeling that is called *Counter Table* [1]. A counter table is a special structure that stores only two columns. A single attribute primary key, as well as a counter column that counts the frequency of a specific item. We are going to this approach for our needs. We will create a table that will have the playlist's name as a primary key, together with a counter that keeps track of how many times the specific playlist has been played. Counter columns are a great choice when coming to counting objects that update very often. Another thing worth mentioning, is that we cannot insert data to a counter table, only to update records in this table [4]. Our model that supports this query can be seen in *Table 8*.

playlist_counter
playlist_name text
played counter
Primary Key (playlist_name)

Table 8

Query 11 – Find how many times a song has been played

Using exact the same approach as we did in Query 10, we can create a counter table that keeps track of how many times a song has been played.

song_counter
song_name text
played counter
Primary Key (song_name)

Table 9

Query 12 – List playlists in decreasing popularity

This query is one example of many queries that Cassandra is not designed to implement efficiently. The way we are going to so address this problem is one of Cassandra's bad practices. Let's try to explain this in more detail. In a usual relation database, since we have a table like *Table 8*, answering this query could be straightforward, since we could issue an order by command on the played column. The problem is that Cassandra does not allow us to sort tables based on counter attributes [5] [6]. The roots of this limitation is that counter columns cannot be part of the primary key and hence it is impossible to create a clustering order on them.

One solution would be to get all data from all clusters and then sort them in a driver program using a programing language, or a tool like Spark. This however is a bad approach, since we may have millions of records, making it infeasible for a common computer to handle. We should find a way to solve this in Cassandra. Another limitation with counter tables, is that we can't create indexes on them, neither we can create materialized views.

A solution that introduces a lot of data duplication, is to create a same table as before, but now the played column will be an integer-type and not a counter. This will be used as a lookup table only and will be updated each time we want to answer this query, using the data from the original table. One could wonder we don't use an integer value instead of a counter from the first place, instead of creating two different tables. The truth is that we could do this, but tables like the ones in Queries 10 and 11 are ideal for updating counter values that increase as the time goes by. So we will proceed with the idea of the second, copy table. The first thing we need to do is to find a way to copy the data from table one to table two. A very good idea is to use the built-in COPY command [7]. Another idea was to use insert statement combined with a select statement that gets the data from the first table. But unfortunately, this is another thing that Cassandra does not support [8]. In our cases where we have only a few rows just for demonstration, we will use simple insert statements.

If we now try to get all data, we can see that the results are not sorted by the total number of times played. The problem is that sorting happens per partition and not for all data at the same time. Cassandra is well designed to answer queries that get data from one or few partitions and not from all of them. This is the reason why a query like this is not supposed to be answered with Cassandra. The only possible way to let this happen, is to find a way to let all records get inserted to a specific cluster, violating goal no. 1. We can do this by adding an additional attribute in the primary key which will be the partition key and give it a constant value for all inserted data, so that we can put everything in a single partition. We can then define a clustering order on the played attribute and then retrieve all the records sorted [3] [9] .

The model that we described can be seen in *Table 10*.

Playlist_counter_2
hash_prefix bigint
playlist_name text
played bigibt
Primary Key ((hash_prefix), played, playlist_name)

Table 10

Now we can easily answer the query.

Query 13 – List users in decreasing popularity

For this query we can follow exact the same approach as before, since it is the same case. Our model can be seen in *Table 11*.

user_follower_count
hash_prefix bigint
user_name text
followers bigint
Primary Key ((hash_prefix), followers, user_name)

Table 11

References

[1] [Online]. Available: https://docs.datastax.com/en/cql/3.3/cql/cql_using/useCountersConcept.html.

[2] [Online]. Available: https://en.wikipedia.org/wiki/Apache_Cassandra.

[3] [Online]. Available: http://www.datastax.com/dev/blog/basic-rules-of-cassandra-data-modeling.

[4] [Online]. Available: https://docs.datastax.com/en/cql/3.3/cql/cql_using/useCounters.html.

[5] [Online]. Available: http://stackoverflow.com/questions/8428364/how-to-get-sorted-counters-from-cassandra.

[6] [Online]. Available: http://stackoverflow.com/questions/14455145/how-to-sort-by-counter-in-cassandra.

[7] [Online]. Available: https://docs.datastax.com/en/cql/3.3/cql/cql_reference/copy_r.html.

[8] [Online]. Available: http://stackoverflow.com/questions/21363046/how-to-select-data-from-a-table-and-insert-into-another-table.

[9] [Online]. Available: http://www.planetcassandra.org/blog/we-shall-have-order/.