Session 5

SETUP 1
OVERVIEW 2
Adding and Deleting Tracks to a Playlist 2
The code 2
EXERCISE #1: Add Playlist Tracks Table 3
EXERCISE #2: Examine the Method addTrackToPlaylist 4
EXERCISE #3: Implement the DeleteTrackFromPlaylist method 4
EXERCISE #4: Delete a Whole Playlist With an Atomic Block 5
EXERCISE #5: Testing it all out 5
EXERCISE #6: Create Indexes on the Playlists 6
Appendix I: The playlist_tracks Table Solution 6
Appendix II: Code to find the playlist in a List and Remove It: 7

SETUP

You can continue using the database from the previous session, but if you would like to start fresh, we've provided the script "playlist.cql" in the scripts directory.

Here are the directions to start from scratch: Wipe out everything you've done by dropping the keyspace. You can't be "using" the keyspace that you're dropping, so either use the system keyspace:

```
use system;
```

or restart cqlsh.

To drop the keyspace:

```
drop keyspace playlist;
```

To recreate everything, there is a script in the "scripts" directory of the project. You can run the script on Unix machines with the SOURCE command. For example, if you run it from the scripts directory:

```
source 'playlist.cql';
```

Double-check the paths in playlist.cql to the .csv files in the COPY commands, and ensure that the copy command will read in the csv files.

Load the application in Eclipse as you have done in the previous sessions, and run it accordingly.

OVERVIEW

A playlist is a bit boring if we can't add any tracks to it, so in this module, we'll add a table to store playlist tracks, and develop the code to add and remove tracks from it. We've bolstered up the playlist object, and added two key things:

Adding and Deleting Tracks to a Playlist

To add tracks to the playlist, drill down (click on the playlist name). You'll see the playlist name at the top of the screen, and the music picker below. Navigate to the track that you want with the music picker, and press the "+" button. To delete a track from the playlist, press the "-" link next to the track name.

The code

1) A static inner class called PlaylistTracks. In other words playlists *have* playlistTracks. A playlist track contains the following fields:

```
private String track_name;
```

```
private String artist;
private int track_length_in_seconds;
private String genre;
private UUID track_id;
private Date sequence_no;
```

There are 2 constructors for the PlaylistTrack object. The First:

```
PlaylistTrack(Row row)
```

constructs a PlaylistTrack from a row read from Cassandra. The second:

```
PlaylistTrack(TracksDAO track) {
```

will build one from the given TrackDAO object. This is used when adding a track from the music database to a playlist.

2) The other thing is the list of playlist tracks in the PlaylistDAO class:

```
private List<PlaylistTrack> playlistTrackList;
```

EXERCISE #1: Add Playlist Tracks Table

1. Each playlist track has the following columns. The challenge is to come up with the right key.

Column Name	Type
artist	text
track_name	text
genre	text
track_length_in_seconds	int
track_id	UUID

2. In addition we need a key for each track. Our goal is the following: one partition per playlist, and we need to represent the ordering of the tracks in the playlist. The way we uniquely identify a playlist is by the key (username, playlist_name). In addition, if we want to order the tracks, we can add a sequence number to the key.

3. Think about which datatype can easily represent the sequence_number column.

4. Try to come up with the table definition here.

5. Compare your solution to our solution in Appendix I: The playlist_tracks Table Solution below .

6. Create the table using our definition, as the rest of our code depends on it.

EXERCISE #2: Examine the Method addTrackToPlaylist

The code which adds tracks to a playlist is in the file PlaylistDAO.java, method

```
addTrackToPlaylist(PlaylistTrack playlistTrack)
```

We've implemented it for you as you have already written many methods that insert tracks. In addition, there are quite a few fields, so the code can be quite tedious. The method does the following steps:

a. Generate a new timestamp called sequence_no, so this track is added to the end of the playlist:

b. Increase the playlist_length_in_seconds member variable by the length of the new playlist

```
this.playlist_length_in_seconds +=
playlistTrack.track_length_in_seconds;
```

c. Insert the playlist into Cassandra

d. This playlist has an instance variable called playlistTrackList, which is a list of playlist tracks. Add the new playlist onto the end of that list.

```
this.playlistTrackList.add(playlistTrack);
```

EXERCISE #3: Implement the DeleteTrackFromPlaylist method

In this method, we need to undo everything that is done in the AddTrackToPlaylist method.

7. First, implement a loop to search through the playlistTrackList member variable, find the one with the sequence_no that matches sequenceNumberToDelete, and

a. Remove it from the list

b. Set the variable playlistTrackToDelete to the object you found

Note that the sequence_no field of the PlaylistTrack is a Date object, and the sequenceNoToDelete is a long. You convert the Date to a long with the getTime() method. Eg:

```
sequence_no.getTime() == sequenceNumberToDelete
```

8. Now decrement the playlistLengthInSeconds by the length of the playlist track that was found.

9. Add Code to remove it from the playlist_tracks table. Think about it's primary key, and which columns you need to include in the WHERE clause of the DELETE statement to delete a single playlist.

EXERCISE #4: Delete a Whole Playlist With an Atomic Block

The method deletePlaylist deletes this playlist. It is not a static method, meaning that all of the instance variables contain the relevant playlist data. If you recall, we keep playlist data in two places

a. The playlist name in a set, in the user table

b. The playlist track data in the playlist_tracks table

We will need to delete the playlist in both places.

10. In the PlaylistDAO class, the method deletePlaylist deletes the playlist and all of the tracks. Fill in the CQL in the getsession().prepare("<fill this in here>") statements to delete the playlist from both tables. Notice the order of arguments to the bind() method just below the prepare statement.

EXERCISE #5: Testing it all out

11. Open your favorite CQL tool such as cqlsh or DevCenter.

12. Using the application, create a playlist, and add tracks to it. Now view all of your tracks using your query tool.

```
select * from playlist_tracks;
```

13. Now delete a track from the playlist, and validate the deletion with CQL. In addition, test your "delete playlist" code in the previous exercises, and again validate them with CQL.

EXERCISE #6: Create Indexes on the Playlists

Let's say playlists were all public, and we would like to find all of the playlists that contain a particular genre. We can do this by creating a secondary index on the playlist_tracks table.

14. Using a query tool, create an index on the playlist_tracks table on the genre column.

15. Add some classical music to a playlist using the web application.

16. Now search the playlist_tracks table for the playlist_name and usename for all playlists that contain classical music.

17. Some people feel that 3 minutes of classical music is minimal. Let's add an additional predicate (with an and keyword) to your query to find all playlists with long classical songs (track_length_in_seconds > 180). Notice that you get an error stating that you must add the "ALLOW FILTERING" clause to your query.

18. Re-try the query with the "ALLOW FILTERING" clause.

Appendix I: The playlist_tracks Table Solution

Column Name	Type
artist	text
track_name	text
genre	text
track_length_in_seconds	int
track_id	UUID
username	text
playlist_name	test
sequence_no	timestamp

Are you surprised by the choice of type for the sequence_no? If we simply treat the tracks as time series data, and use the time of when we insert the data, we can always maintain the order of the columns, and not need to read before we write. If we used an integer column, we would need to know the value of the highest sequence_no before we can write a new value. Since we want the data in sequence_no order, sequence_no is our cluster columns.

Now what about our partition key – we will use a composite partition key of (username, playlist_name) as the partition key as that will give us one playlist per partition. Now each track can be uniquely identified by (partition key, sequence_no), so our primary key is ((username, playlist_name), sequence_no)

This is the full table definition:

```
create table playlist_tracks
(username text,
 playlist_name text,
 sequence_no timestamp,
 artist text,
 track_name text,
 genre text,
 track_length_in_seconds int,
 track_id UUID,
 primary key ((username, playlist_name), sequence_no )
);
```

Appendix II: Code to find the playlist in a List and Remove It:

```
// Loop through all of the tracks in the playlistTrackList
// We are using a simple iterator i in this loop
for (int i = 0; i < this.playlistTrackList.size(); i++) {

// extract the time of from the current playlist track's sequence number
// and compare it to the given time

if (this.playlistTrackList.get(i).sequence_no.getTime() == sequenceNumberToDelete)
 {

    // If it's correct, set playlistTrackToDelete,
    // remove it from the playlist, and stop looping

    playlistTrackToDelete = this.playlistTrackList.get(i);
    this.playlistTrackList.remove(i);
    break;
 }
}
```