Playlist tutorial (EOSL)
# Table of Contents

The Playlist tutorial.................................................................................................................. 4
Setting up your environment...................................................................................................... 6
Architecture of the Playlist example application........................................................................ 7
Step 1: Connecting to Cassandra from a web application............................................................ 10
Step 2: Adding artist and track features....................................................................................... 14
  Building and running step 2........................................................................................................ 18
Step 3: Adding hot tracks............................................................................................................ 21
  Building and running step 3........................................................................................................ 25
Step 4: Adding users and custom playlists.................................................................................. 27
  Building and running step 4........................................................................................................ 37
Step 5: Add statistics views to monitor Playlist......................................................................... 39
  Building and running step 5........................................................................................................ 41
Step 6: Optimize the Playlist application................................................................................... 43
  Building and running step 6........................................................................................................ 44
The Playlist tutorial

This tutorial is intended for programmers interested in learning about Apache Cassandra. It covers the techniques used to create Cassandra databases and tables, and how to access the data using the Cassandra drivers.

The Playlist application is a Java web application that manages a collection of music files. The tutorial provides step-by-step instructions for creating this application and enhancing it to demonstrate features of Cassandra.

Before you start this tutorial

Before starting, you should have a good working knowledge of programming languages, web application development, and a basic understanding of database concepts like tables, columns, and queries. You should be familiar with running commands from a Unix terminal.

Goals of the tutorial

This tutorial will give you hands-on experience creating Cassandra databases and tables, importing data, and using the Cassandra drivers to access the data from an application. You will also learn about data modeling for Cassandra and how modeling for a NoSQL database differs from relational databases.

The hands-on instructions in this tutorial are focused on changes in the data model and interactions with Cassandra as you build and enhance the application. The application code is discussed in depth, but you do not need to alter the application code to as you proceed through the steps of the tutorial.

Each chapter of the tutorial has a corresponding branch in the Git workspace. The master branch is the complete and final application as it should appear at the end of the tutorial. Breaking up the application in this manner allows you to see exactly which files and code sections changed from one chapter to the next.

Required software

The following software is required to build and run the Playlist tutorial example application:

- A Linux or OS X machine
- JDK 7 or later
- Apache Maven 3 or later
- Git 2.3 or later
- DataStax Enterprise 5.0 (page ) or later
- The playlist example code
- A code editor of your choice

This tutorial assumes you have installed DataStax Enterprise, which includes Cassandra. All the tools and instructions will work with a Cassandra installation.
Apache Maven is used to manage the example application's dependencies, and to build and run the application.

As you add functionality to the application, git is used to retrieve the example source code and compare the code between steps.
Setting up your environment

Follow these steps to set up your development environment for running the Playlist example application.

1. Install and start DataStax Enterprise if you do not have a currently running cluster.
   Use the installer on Linux (page) or OS X (page) to install a single-node cluster on your local machine.

2. Ensure DataStax Enterprise is running.

   ```
   $ nodetool status
   
   Datacenter: Cassandra
   ======================  
   Status=Up/Down
   | State=Normal/Leaving/Joining/Moving
   -- Address    Load     Owns   Host ID
   Token                                    Rack
   UN 127.0.0.1   5.49 MB  ?
   54b8ef4e-9e0f-4c0c-831a-2b3557fe5414 -9223372036854775808
   rack1
   ```

3. If you are on OS X, add /usr/share/dse/bin to your PATH.
   If you are using the BASH shell, add the following line to your ~/.bash_profile or ~/.profile file.

   ```
   export PATH = $PATH:/usr/share/dse/bin
   ```

4. Retrieve the Playlist example code using git.

   ```
   $ git clone https://github.com/DataStaxDocs/playlist.git
   ```

5. Use Maven to download and install the example application's dependencies.

   ```
   $ cd playlist && mvn verify
   ```

   This step installs the DSE Java driver and other libraries used when running the application.
Architecture of the Playlist example application

The Playlist application is a web application written in Java that can be deployed to a Java EE web container. Playlist uses a Model View Controller (MVC) design. The application model is handled by data-access objects to manage the Cassandra table data. The application view consists of JSP files that display the data. The application controller is made up of Java servlets to manage the requests from the view and the data returned from the model.

In the Playlist example, the single-node Cassandra database and the application are on the same host, which is not a typical production scenario. The design of the application allows you to easily add more nodes to the Cassandra cluster and separate the application server from the database node with minimal code changes.

Playlist components

The Playlist example application is made up of several separate components that encapsulate the functionality of an online music service:
• The song database: a collection of digital music files organized by artist, track name, and genre
• User registration: creates users of the music service
• User playlists: collections of music files created by users
• Application statistics: overall data about the service

The data model

The Playlist application’s database tables differ significantly from the tables you typically find in a relational database application. In a typical relational database, the tables are designed based on the relations between the entities represented by each table. Creating the tables and relations to avoid duplication of data between tables is called data normalization. After the data has been completely normalized, relational database administrators and application developers write queries to retrieve the data, often using joins to find the related data in multiple tables.

Cassandra databases are designed around the type of queries used in the application. Cassandra’s strength is its ability to support very fast reads and writes with large sets of data spread across many nodes in a cluster. For this reason, the database tables are denormalized: the data is broken out into different tables, and column data is often duplicated across many tables. This may seem strange to developers used to relational database applications, but it allows for much greater performance, particularly as the amount of data grows larger and nodes in the database cluster increase.

The Playlist example application uses separate tables for users, tracks organized by track name, tracks organized by artist, tracks organized by genre, artists, and playlists.

For more information on modeling data using Cassandra, see The CQL data modeling documentation (page ).

The steps of this tutorial

Each step in this tutorial represents a runnable version of the Playlist application, and is a branch of the Git project. Each successive step enhances the application. All the application code is updated for you at each successive step so you can concentrate on the changes in the queries and underlying Cassandra schema used in that step. All the application code changes, however, are explained as part of each step, and you can compare the changes between branches to see exactly what is different in the code.

In step 1, you will run a skeleton version of the application that simply connects to a Cassandra node in the configured cluster and displays information about the environment in which the node is running. In step 2, you will create the keyspace and tables used by the Playlist application, load the music catalog data to those tables, and enable the ability to browse through the music catalog in the web application.

In step 3, you will add the ability to mark a music track as a “hot track”, using CQL to set an expiration time for a change in the data.
You will add the ability for individual users to log in and create customized playlists from the music catalog in step 4. In step 5 you will add a table to collect statistics from the Playlist web application.

Finally, in step 6 you will optimize how Playlist returns query data to improve performance.
Step 1: Connecting to Cassandra from a web application

The most basic version of Playlist doesn't have any real functionality, but does show how to connect to Cassandra using the Cassandra Java driver and retrieve information from the system table.

There are two Java classes that are used in this step to connect to a Cassandra node and retrieve information about the Cassandra cluster. The playlist.model.CassandraData class defines the getSession method used throughout Playlist to establish a connection to Cassandra, represented by the com.datastax.driver.core.Session object. CassandraData is the parent class of all the other model classes.

```java
public static Session getSession() {
    if (cassandraSession == null) {
        cassandraSession = createSession();
    }

    return cassandraSession;
}
```

The getSession method calls the protected createSession method if there is no current Session object. The createSession method connects to the local Cassandra node and returns the Session object.

```java
protected static Session createSession() {
    Cluster cluster = Cluster.builder().addContactPoint("localhost").build();
    return cluster.connect();
}
```

If you are connecting to a remote cluster, or are using a multi-node cluster, replace localhost with an IP address of the remote node in addContactPoint.

The playlist.model.CassandraInfo class stores information about the Cassandra cluster used by Playlist. CassandraInfo stores the cluster name and Cassandra version used by the cluster. The constructor retrieves this information using a CQL query run against the system table on the local Cassandra node.

```java
public CassandraInfo() {
    Row row = getSession().execute("select cluster_name, release_version from system.local").one();
    cassandraVersion = row.getString("release_version");
    clusterName = row.getString("cluster_name");
}
```
Step 1: Connecting to Cassandra from a web application

The `com.datastax.driver.core.Row` class represents a row in a Cassandra table. The `Row` class provides methods for retrieving the column data by type. In this case, `Row.getString()` is used to retrieve the Cassandra version and cluster name.

The information stored in `CassandraInfo` is then made available to the view JSP files through the `playlist.controller.HomeServlet` controller. `HomeServlet` responds to HTTP GET requests and creates a new instance of `CassandraInfo`.

```java
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    String javaVersion = System.getProperty("java.version");
    CassandraInfo cassandraInfo = new CassandraInfo();

    request.setAttribute("java_version", javaVersion);
    request.setAttribute("cassandra_info", cassandraInfo);
    getServletContext().getRequestDispatcher("/home.jsp").forward(request, response);
}
```

`HomeServlet` then adds the information from `CassandraInfo` and the Java runtime version to the request headers before forwarding the response to the `home.jsp` view file to display to the user.

```
<h1>Playlist</h1>
<img class="dim" src="images/favicon.png"/>
Welcome to the playlist application. <br/>
Java version: ${java_version} <br/>
Cassandra version: ${cassandra_info.cassandraVersion} <br/>
Cluster name: ${cassandra_info.clusterName}
```

Prerequisites:
You must have set up your environment (page 6) before starting this step.

1. In a terminal, navigate to the playlist Git workspace.
   ```
   $ cd playlist
   ```

2. Check out the step1 branch using `git`.
   ```
   $ git checkout step1
   ```

3. Build and start the Playlist example application using `mvn`.
   ```
   This step compiles the Java source files, assembles the Playlist web application, and deploys it to a Jetty local instance.
   ```
Step 1: Connecting to Cassandra from a web application

$ mvn verify cargo:run

You will see the output of the Maven plugins as it compiles and assembles Playlist, and then the following when it is ready to be tested:

```
[INFO] [talledLocalContainer] Jetty 9.2.11.v20150529 started on port [8080]
[INFO] Press Ctrl-C to stop the container...
```

4. In a web browser, go to: http://localhost:8080/playlist

You will see the Java version and Cassandra cluster information displayed at the top of the page.

```
Welcome to the playlist application.
Java version: 1.7.0_80
Cassandra version: 2.1.5.448
Cluster name: Test Cluster
```

5. Stop the Jetty instance by pressing Ctrl+C in your terminal.

```
[INFO] Press Ctrl-C to stop the container...
^C[INFO] [talledLocalContainer] Jetty 9.2.11.v20150529 is stopping...
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
[INFO] Total time: 34:26.970s
[INFO] Final Memory: 14M/183M
[INFO]
```

What's next:
Proceed to Step 2: Adding artist and track features (page 14)
Step 2: Adding artist and track features

In Step 1, you connected to Cassandra from a web application and displayed some data stored in the `system` table. Now we will enhance Playlist by importing song data from a comma-separated value (CSV) file into tables, organized by the types of queries required by the Playlist application.

Setting up Cassandra keyspaces and tables

You will create a new `keyspace` in Cassandra called `playlist`. A Cassandra keyspace is similar to a named database in a relational database, but is slightly different. A Cassandra keyspace is a namespace that defines how data is replicated throughout the Cassandra cluster. You typically create one keyspace per application. When creating a keyspace, you must configure how the data is replicated by specifying a `replication strategy`. In this tutorial, we are using a simple cluster with a single node, so we are using the `SimpleStrategy`, and a `replication factor` of 1. The replication factor specifies how many nodes on which the data will be replicated. If our cluster had several nodes, we could set a higher replication factor to ensure that the table data would still be available if one of the nodes in the cluster failed.

After creating the keyspace, you will then create a table named `artists_by_first_letter` and populate it with data from a CSV file. The `artists_by_first_letter` table is designed for a query used in Playlist to list the artists by the first letter of their names, and consists of two columns, `first_letter` and `artist`. The `artists_by_first_letter` table's primary key is a `composite primary key`, also called a compound primary key, made up of these columns. The first part of the definition of a composite primary key is the `partition key`, which is used by Cassandra to distribute the table data across nodes. The second part of the definition is the `clustering key`, or `clustering column or columns`, which is used by Cassandra to sort the data within the partition. You can use multiple columns when setting the partition key and clustering key of a composite primary key. Designing the table in this way allows Cassandra to easily distribute the table data throughout the nodes in the cluster. This type of table design is quite different than a fully normalized relational database, but allows for increased performance and failover in a clustered database. See the CQL documentation (page x) for more information on composite primary keys.

<table>
<thead>
<tr>
<th>artist_by_first_letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>first_letter</td>
</tr>
<tr>
<td>artist</td>
</tr>
</tbody>
</table>

The songs are stored in two tables, `track_by_artists` and `track_by_genre`, to allow searching by artist name and type of music. These tables are also populated from a CSV file.

<table>
<thead>
<tr>
<th>track_by_artist</th>
</tr>
</thead>
<tbody>
<tr>
<td>track</td>
</tr>
</tbody>
</table>
Step 2: Adding artist and track features

<table>
<thead>
<tr>
<th>track_by_artist</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>artist</strong></td>
</tr>
<tr>
<td><strong>track_id</strong></td>
</tr>
<tr>
<td><strong>track_length_in_seconds</strong></td>
</tr>
<tr>
<td><strong>genre</strong></td>
</tr>
<tr>
<td><strong>music_file</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>track_by_genre</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>track</strong></td>
</tr>
<tr>
<td><strong>artist</strong></td>
</tr>
<tr>
<td><strong>track_id</strong></td>
</tr>
<tr>
<td><strong>track_length_in_seconds</strong></td>
</tr>
<tr>
<td><strong>genre</strong></td>
</tr>
<tr>
<td><strong>music_file</strong></td>
</tr>
</tbody>
</table>

Data access objects for artists and tracks

There are two new data-access object classes and their corresponding controller servlets, `playlist.model.ArtistsDAO` and `playlist.controller.ArtistServlet`, and `playlist.model.TracksDAO` and `playlist.controller.TrackServlet`.

The `ArtistsDAO` class is used to return a list of the artists in the music service. It has a single finder method, `listArtistsByLetter`, which is used to find all artists whose names begin with the specified letter. The `listArtistsByLetter` method can return the list ordered in ascending alphabetic order, or optionally in descending alphabetic order. The CQL query is a prepared statement bound by the first letter parameter. A prepared statement acts like a template to improve the performance of similar queries. The prepared statement is then bound with the parameter data before execution.

```java
String queryText = "SELECT * FROM artists_by_first_letter WHERE first_letter = ?" + (desc ? " ORDER BY artist DESC" : "");
PreparedStatement preparedStatement = getSession().prepare(queryText);
BoundStatement boundStatement = preparedStatement.bind(first_letter);
```
Step 2: Adding artist and track features

The CQL query will use the `ORDER BY artist DESC` fragment only if the `desc` parameter is true.

The query is executed and the results are returned as a `com.datastax.driver.core.ResultSet` object.

```java
ResultSet results = getSession().execute(boundStatement);
```

Each row in the results is parsed and added to a `List<String>` object, and then the list is returned for display in the view.

```java
List<String> artists = new ArrayList<>();
for (Row row : results) {
    artists.add(row.getString("artist"));
}
return artists;
```

The `TracksDAO` class manages the display of track information and the ability to add new tracks to the catalog. Like `ArtistsDAO`, it has finder methods, `listSongsByArtist` and `listSongsByGenre`. It also has an `add` method to add new tracks using the information entered into a form in the view.

The `SELECT` queries used in `TracksDAO` are similar to the query used in `ArtistsDAO`.

```java
public static List<TracksDAO> listSongsByArtist(String artist) {
    String queryText = "SELECT * FROM track_by_artist WHERE artist = ?";
    PreparedStatement preparedStatement = getSession().prepare(queryText);
    BoundStatement boundStatement = preparedStatement.bind(artist);
    ResultSet results = getSession().execute(boundStatement);
    List<TracksDAO> tracks = new ArrayList<>();
    for (Row row : results) {
        tracks.add(new TracksDAO(row));
    }
    return tracks;
}

public static List<TracksDAO> listSongsByGenre(String genre) {
    String queryText = "SELECT * FROM track_by_genre WHERE genre = ?";
    PreparedStatement preparedStatement = getSession().prepare(queryText);
    BoundStatement boundStatement = preparedStatement.bind(genre);
    ResultSet results = getSession().execute(boundStatement);
    List<TracksDAO> tracks = new ArrayList<>();
    for (Row row : results) {
```
Step 2: Adding artist and track features

```java
tracks.add(new TracksDAO(row));

return tracks;
}
```

These finder methods use a private constructor for TracksDAO that parses the Row object and extracts the information from the columns using getter methods according to the type of data stored in the column.

```java
private TracksDAO(Row row) {
    track_id = row.getUUID("track_id");
    artist = row.getString("artist");
    track = row.getString("track");
    genre = row.getString("genre");
    music_file = row.getString("music_file");
    track_length_in_seconds = row.getInt("track_length_in_seconds");
}
```

The add method uses INSERT statements to add new rows to the three tables used: artists_by_first_letter, track_by_artist, and track_by_genre.

```java
public void add() {

    // Compute the first letter of the artists name for the artists_by_first_letter table
    String artist_first_letter = this.artist.substring(0,1).toUpperCase();

    // insert into artists_by_first_letter
    PreparedStatement preparedStatement = getSession().prepare("INSERT INTO artists_by_first_letter (first_letter, artist) VALUES (?, ?)");
    BoundStatement boundStatement = preparedStatement.bind(artist_first_letter, this.artist);
    getSession().execute(boundStatement);

    // insert into track_by_artist
    preparedStatement = getSession().prepare("INSERT INTO track_by_artist (genre, track_id, artist, track, track_length_in_seconds) VALUES (?, ?, ?, ?, ?)");
    boundStatement = preparedStatement.bind(this.genre, this.track_id, this.artist, this.track, this.track_length_in_seconds);
    getSession().execute(boundStatement);

    // insert into track_by_genre
    preparedStatement = getSession().prepare("INSERT INTO track_by_genre (genre, track_id, artist, track, track_length_in_seconds) VALUES (?, ?, ?, ?, ?)");
    boundStatement = preparedStatement.bind(this.genre, this.track_id, this.artist, this.track, this.track_length_in_seconds);
    getSession().execute(boundStatement);
```
Step 2: Adding artist and track features

The controller and view

The two new controller servlets, ArtistServlet and TrackServlet are similar to the HomeServlet used in step 1, managing the requests and forwarding them on to the proper view for a response. ArtistServlet examines the request to determine if the user wants to display the artists in descending order, and if so, calling listArtistsByFirstLetter with the desc parameter set to true. TrackServlet responds to HTTP POST requests when users create a new track, extracting the track information from the request.

Changes from Step 1

To see all code changes from the step1 branch, enter the following command in a terminal in the playlist directory:

$ git diff step1..step2

Building and running step 2

You will now create the keyspace and tables in Cassandra, and then build and run Playlist, which has been enhanced with a music catalog and the ability to list tracks by artist.

Prerequisites:

You must have set up your environment (page 6) before starting this step.

1. In a terminal, go to the playlist workspace.

   $ cd playlist

2. Check out the step2 branch using git.

   $ git checkout step2

3. Run the cqlsh command.

   $ cqlsh

4. Create a new keyspace called playlist with a SimpleStrategy replication strategy and the replication factor set to 1.

   create KEYSPACE playlist WITH replication = 
   {'class':'SimpleStrategy', 'replication_factor': 1 };

5. In cqlsh go to the playlist keyspace.
6. Create the `artists_by_first_letter`, `track_by_artist`, and `track_by_genre` tables.

create table artists_by_first_letter (first_letter text, artist text, primary key (first_letter, artist));
create table track_by_artist (track text, artist text, track_id UUID, track_length_in_seconds int, genre text, music_file text, primary key (artist, track, track_id));
create table track_by_genre (track text, artist text, track_id UUID, track_length_in_seconds int, genre text, music_file text, primary key (genre, artist, track, track_id));

7. Populate the tables with data from the `songs.csv` and `artists.csv` files located in `playlist/scripts`.

    copy artists_by_first_letter (first_letter, artist) from 'scripts/artists.csv' WITH DELIMITER = '|';
    copy track_by_artist (track_id, genre, artist, track, track_length_in_seconds, music_file) FROM 'scripts/songs.csv' WITH DELIMITER = '|' AND HEADER=true;
    copy track_by_genre (track_id, genre, artist, track, track_length_in_seconds, music_file) FROM 'scripts/songs.csv' WITH DELIMITER = '|' AND HEADER=true;

8. Quit `cqlsh`.

    exit;


    $ mvn verify cargo:run

10. In a web browser, navigate to or reload: `http://localhost:8080/playlist`

11. Click **Visit the Song Database**. You can also add tracks by clicking **Add a Song** and filling the in the form fields.

    You now have the ability to browse the music catalog by artist name and genre.

    **Note that My Playlists and Statistics** are not yet implemented.

12. Stop the Jetty instance by pressing Ctrl+C in your terminal.

    [INFO] Press Ctrl-C to stop the container...
    ^C[INFO] [talledLocalContainer] Jetty 9.2.11.v20150529 is stopping...
Step 2: Adding artist and track features

What's next:
Proceed to Step 3: Adding hot tracks (page 21).
Step 3: Adding hot tracks

Now you will enhance Playlist to allow users to select a particular track as a hot track. This will involve creating a new table, `track_by_id`, and altering the existing `track_by_artist` and `track_by_genre` tables with a new column. The `TracksDAO` will be altered to store whether a track is a hot track.

Changes to the Playlist schema

The `track_by_artist` and `track_by_genre` tables need a new boolean column to indicate whether the track has been selected as a hot track. You can add a new column to existing tables by using the `ALTER TABLE` syntax in `cqlsh`.

```
ALTER TABLE track_by_artist ADD starred boolean;
ALTER TABLE track_by_genre ADD starred boolean;
```

<table>
<thead>
<tr>
<th>track_by_artist</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>track</td>
<td>text</td>
<td>PK</td>
</tr>
<tr>
<td>artist</td>
<td>text</td>
<td>PK</td>
</tr>
<tr>
<td>track_id</td>
<td>UUID</td>
<td>PK</td>
</tr>
<tr>
<td>track_length_in_seconds</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>genre</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td>music_file</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td>starred</td>
<td>boolean</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>track_by_genre</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>track</td>
<td>text</td>
<td>PK</td>
</tr>
<tr>
<td>artist</td>
<td>text</td>
<td>PK</td>
</tr>
<tr>
<td>track_id</td>
<td>UUID</td>
<td>PK</td>
</tr>
<tr>
<td>track_length_in_seconds</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>genre</td>
<td>text</td>
<td>PK</td>
</tr>
<tr>
<td>music_file</td>
<td>text</td>
<td></td>
</tr>
</tbody>
</table>
Step 3: Adding hot tracks

You will use an additional query to find tracks by their ID, which means you need a new table, `track_by_id`. This table is similar to the other track tables but uses a simple primary key of the track ID.

```
create table track_by_id (track text, artist text, track_id UUID, track_length_in_seconds int, genre text, music_file text, primary key (track_id));
```

Changes to the model

The `TracksDAO` must be modified for the new `starred` column in the track tables. Add a new field of type `Boolean`:

```
private Boolean starred;
```

The constructors need to be modified. If the object is being retrieved from a row from one of the tracks table, check the value of the `starred` column to set the `starred` field in `TracksDAO`:

```
try {
    starred = row.getBool("starred");
} catch (Exception e) {
    starred = false; // If the field doesn't exist or is null we set it to false
}
```
For a brand new instance of `TracksDAO`, set the `starred` field to false by default:

```java
this.starred = false;
```

Now you'll add a method to set `starred` to true when a user selects it as a hot track. You want the track to be starred for only a minute, so you will use the `USING TTL time in seconds` CQL keyword to set the expiration time for the updated column. In this case, you will set the `starred` column to true for 60 seconds.

```java
public void star() {
    PreparedStatement preparedStatement = getSession().prepare("UPDATE track_by_artist USING TTL 60 SET starred = true where artist = ? and track = ? and track_id = ?");
    BoundStatement boundStatement = preparedStatement.bind(artist, track, track_id);
    getSession().execute(boundStatement);

    preparedStatement = getSession().prepare("UPDATE track_by_genre USING TTL 60 SET starred = true where genre = ? and artist = ? and track = ? and track_id = ?");
    boundStatement = preparedStatement.bind(genre, artist, track, track_id);
    getSession().execute(boundStatement);
}
```

You need a new finder method for returning a track based on its ID. This finder method queries the new `track_by_id` table with specified track ID.

```java
public static TracksDAO getTrackById(UUID track_id) {
    PreparedStatement preparedStatement = getSession().prepare("SELECT * FROM track_by_id WHERE track_id = ?");
    BoundStatement boundStatement = preparedStatement.bind(track_id);
    ResultSet resultSet = getSession().execute(boundStatement);

    // Return null if there is no track found

    if (resultSet.isExhausted()) {
        return null;
    }

    return new TracksDAO(resultSet.one());
}
```

Finally, the `add` method needs to be updated to add any new tracks to the `track_by_id` table. The code is almost identical to the code that inserts the track to the other track tables.

```java
preparedStatement = getSession().prepare("INSERT INTO track_by_id (genre, track_id, artist, track, track_length_in_seconds) VALUES (?, ?, ?, ?, ?)");
```
Step 3: Adding hot tracks

boundStatement = preparedStatement.bind(this.genre, this.track_id, this.artist, this.track, this.track_length_in_seconds);
getSession().execute(boundStatement);

Changes to the controller

The TrackServlet needs to be enhanced to work with starred tracks. Users will click on tracks that they want starred. In the doPost method, you extract out the star parameter from the request, and check if the user starred the track. If so, call TracksDAO.getTrackById() to retrieve the TracksDAO instance, then call the star() method on that instance so it will update the track in the Cassandra tables.

String star = request.getParameter("star");
if (star != null) {
    TracksDAO.getTrackById(UUID.fromString(star)).star();

    response.sendRedirect("tracks?howmany=+ howmany
        + (artist == null ? "" : "&artist=" + URLEncoder.encode(artist, "UTF-8"))
        + (genre == null ? "" : "&genre=" + URLEncoder.encode(genre, "UTF-8"));
}

Changes to the view

The tracks.jsp file handles the view of the tracks. You need to add a button to each track so users can star the track, and display the correct image if the track is a hot track.

<table class="table">
    <c:forEach var="track" items="${tracks}"
        <tr>
            <c:set var="startype" value="${track.starred ? 'yellowstar.png' : 'emptystar.png'}"/>
            <c:if test="${empty frame}">
                <td class="field_start">
                    <button name="star" value="${track.track_id}"><img src="images/${startype}" /></button>
                </td>
            </c:if>
            <c:if test="${frame == 'true'}">
                <td class="field_plus"><input type="button" name="add" value="+" onclick="addTrack('${track.track_id}')"></td>
            </c:if>
            <td class="field_track">${track.track}</td>
            <td class="field_genre">${track.genre}</td>
            <td class="field_sec">
                <fmt:formatNumber value="${track.track_length_in_seconds div 60}" minIntegerDigits="1" maxFractionDigits="0"/>
                <fmt:formatNumber value="${track.track_length_in_seconds % 60}" minIntegerDigits="2" />
            </td>
    </tr>
</c:forEach>
</table>
Step 3: Adding hot tracks

The `<c:set var="startype" value="${track.starred ? 'yellowstar.png' : 'emptystar.png'}"/>` tag checks whether the track is a hot track, and if so, adds a yellow star image. If the track is not a hot track, the image is an empty star.

The `<button name="star" value="${track.track_id}" img src="images/ ${startype}"/></button>` tags add the track ID to the request headers if the user presses the star button to make the track a hot track.

Changes from Step 2

To see all code changes from the step2 branch, enter the following command in a terminal in the playlist directory:

```
$ git diff step2..step3
```

Building and running step 3

You will now alter the track tables with a new boolean column to indicate if a track has been starred, then modify the Playlist application to allow tracks to be starred by the user.

**Prerequisites:**

You must have set up your environment *(page 6)* before starting this step.

1. In a terminal, go to the playlist workspace.
   
   ```
   $ cd playlist
   ```

2. Check out the step3 branch using git.
   
   ```
   $ git checkout step3
   ```

3. Run the `cqlsh` command.
   
   ```
   $ cqlsh
   ```

4. In the playlist keyspace alter the `track_by_artist` and `track_by_genre` tables with a new boolean column called `starred`.
   
   ```
   use playlist;
   alter table track_by_artist add starred boolean;
   alter table track_by_genre add starred boolean;
   ```
5. Create and populate a new track table, $track_by_id$ used to query a track when it is being starred by the user.

```sql
create table track_by_id (track text, artist text, track_id UUID, track_length_in_seconds int, genre text, music_file text, primary key (track_id));
```

```sql
COPY track_by_id (track_id, genre, artist, track, track_length_in_seconds, music_file) FROM 'scripts/songs.csv' WITH DELIMITER = '|' AND HEADER=true;
```

6. Quit cqlsh.

```sql
exit;
```

7. Build and run Playlist using `mvn`.

```bash
$ mvn verify cargo:run
```

8. In a web browser, navigate to or refresh: http://localhost:8080/playlist

Now you can star hot tracks, and they will remain starred for one minute.

Note that My Playlists and Statistics are not yet implemented.

**What's next:**

Proceed to Step 4: Adding users and custom playlists (page 27).
Step 4: Adding users and custom playlists

A music service needs the ability to allow individual users to log in and create playlists. You will now enhance the Playlist application to implement user authentication and custom playlists.

Changes to the Playlist schema

A new users table holds the usernames and passwords for user authentication. We also need a place to store the playlist names for a user. Users to playlist names is a one-to-many relationship in standard data-modeling: one user can have many playlist names. In a relational database, a one-to-many relationship is usually handled by creating a playlist table with a foreign key to the user ID. In Cassandra, we will use a set<text> collection type in the users table to store the playlist names. We are using the set collection type to ensure that the playlist names are unique.

<table>
<thead>
<tr>
<th>users</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
</tr>
<tr>
<td>password</td>
</tr>
<tr>
<td>playlist_names</td>
</tr>
</tbody>
</table>

Note: To simplify the example application the passwords are stored unencrypted in the table, which is not recommended for any real, production application, which typically interface with an authentication service.

Note: Embedded collection types like set, map, and list have size limitations. See the CQL documentation (page ) for a discussion of when to use collection types.

The playlist is stored in a separate table, playlist_tracks.

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 ));

<table>
<thead>
<tr>
<th>playlist_tracks</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
</tr>
</tbody>
</table>
Step 4: Adding users and custom playlists

<table>
<thead>
<tr>
<th>column</th>
<th>type</th>
<th>key</th>
</tr>
</thead>
<tbody>
<tr>
<td>playlist_name</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td>sequence_no</td>
<td>timestamp</td>
<td>PK</td>
</tr>
<tr>
<td>artist</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td>track_name</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td>genre</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td>track_length_in_seconds</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>track_id</td>
<td>UUID</td>
<td></td>
</tr>
</tbody>
</table>

The `sequence_no` column is of the data type `timestamp`. Using the sequence number as a timestamp allows us to make sure the tracks appear in the correct order. In other words, the sequence of tracks is treated like time series data, where the each successive track has a later timestamp. This simplifies the ordering of the tracks because we don't need to read the sequence number of the last track added to the playlist.

The primary key for `playlist_tracks` is also different than our other compound primary keys used in Playlist. The composite partition key is made up of `username` and `playlist_name`, which creates one playlist per partition. The primary key is made up of that composite partition key and `sequence_no` to uniquely identify the track.

Adding the user model

Users are managed with a new model class, `playlist.model.UserDAO`. `UserDAO` contains add and delete methods for creating and removing users.

```java
public static UserDAO addUser(String username, String password) throws UserExistsException {
    String queryText = "INSERT INTO users (username, password) values (?, ?) IF NOT EXISTS";

    PreparedStatement preparedStatement = getSession().prepare(queryText);

    // Because we use an IF NOT EXISTS clause, we get back a result set with
    // 1 row containing 1 boolean column called ":[applied]"
    ResultSet resultSet = getSession().execute(preparedStatement.bind(username, password));

    // Determine if the user was inserted. If not, throw an exception.
    boolean userGotInserted = resultSet.one().getBool("[applied]");
```
if (!userGotInserted) {
    throw new UserExistsException();
}

// Return the new user so the caller can get the userid
return new UserDAO(username, password);

public void deleteUser() {
    SimpleStatement simpleStatement = new SimpleStatement("DELETE FROM users
    where username = '" + this.username + "'");

    // Delete users with CL = Quorum
    simpleStatement.setConsistencyLevel(ConsistencyLevel.QUORUM);
    getSession().execute(simpleStatement);
}

There is a finder method getUser used to retrieve existing users. Cassandra has tunable consistency that allows you to set the consistency level of the query when it executes. User data often requires a higher consistency level when executing queries than other types of data. For example, if a user has been removed, you do not want replica nodes to allow that user to log in between the time the user was removed and the cluster’s data is updated throughout the cluster’s replica nodes. In this case, we set the consistency level to QUORUM.

private static UserDAO getUser(String username) {
    String queryText = "SELECT * FROM users where username = ?";
    + username + ";";
    PreparedStatement preparedStatement = getSession().prepare(queryText);
    BoundStatement boundStatement = preparedStatement.bind(username);
    boundStatement.setConsistencyLevel(ConsistencyLevel.QUORUM);
    Row userRow = getSession().execute(boundStatement).one();

    if (userRow == null) {
        return null;
    }

    return new UserDAO(userRow);
}

The QUORUM consistency level ensures that a sufficient number of replica nodes in a cluster agree that the returned data is up-to-date. The number of nodes that make a quorum depends on the replication factor of the cluster. For example, in a single data center cluster with a replication factor of 3, 2 replica nodes must respond if the query has a consistency level of QUORUM. While our cluster is a single-node cluster so all the data is stored on one node, we
can easily modify Cassandra to run on a production cluster where the consistency levels of our queries impacts the performance and security of our application.

The `validateLogin` method authenticates the user by checking the username and password. The `validateLogin` method first calls the `getUser` method to retrieve the user data, then makes sure the passwords match.

```java
public static UserDAO validateLogin(String username, String password)
    throws UserLoginException {
    UserDAO user = getUser(username);
    if (user == null || !user.password.contentEquals(password)) {
        throw new UserLoginException();
    }
    return user;
}
```

If the passwords don’t match, a simple runtime exception, `playlist.exceptions.UserLoginException` is thrown.

Adding the playlist model

The `playlist.model.PlaylistDAO` class handles creating, retrieving, modifying, and deleting user playlists, and exposes the following methods.

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createPlaylist</td>
<td>Creates a new playlist for a user with the specified name.</td>
</tr>
<tr>
<td>deletePlayList</td>
<td>Delete the playlist.</td>
</tr>
<tr>
<td>getPlaylistForUser</td>
<td>A static finder method that retrieves the tracks in the specified playlist.</td>
</tr>
<tr>
<td>deleteTrackFromPlaylist</td>
<td>Remove the specified track from the playlist.</td>
</tr>
<tr>
<td>addTrackToPlaylist</td>
<td>Add the specified track to the playlist.</td>
</tr>
</tbody>
</table>

The CQL queries in `createPlaylist`, `deleteTrackFromPlaylist`, `getPlaylistForUser`, and `addTrackToPlaylist` are similar to the queries in the other model classes. The query in `deletePlayList`, however, is a **batch query**, using the `BEGIN BATCH` and `APPLY BATCH` keywords to combine multiple queries so they get applied at the same time.

```java
public void deletePlayList() {
    // Change single quotes to a pair of single quotes for escaping into the database
    String fixed_playlist_name = this.playlist_name.replace('"', '"');
```
Step 4: Adding users and custom playlists

PreparedStatement preparedStatement = getSession().prepare("BEGIN BATCH " +
   "UPDATE users set playlist_names = playlist_names - {'" +
fixed_playlist_name + "'} WHERE username = ? " +
   "DELETE FROM playlist_tracks WHERE username = ? and playlist_name = ? " +
   "APPLY BATCH;";

BoundStatement bs = preparedStatement.bind(this.username, this.username,
this.playlist_name);

getSession().execute(bs);
}

The users and playlist_tracks tables must both be modified if a playlist is deleted, so the
queries are batched together to ensure the tables are updated at the same time. The queries
are in between the BEGIN BATCH and APPLY BATCH keywords.

Adding the user controller

The new playlist.controller.LoginServlet class handles user authentication and
creation. HTTP POST requests are either user login attempts or user creation events. The
doPost method checks the request headers to see what button the user pressed and calls the
private doLogin or doCreateUser methods accordingly.

protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

   // User creation and login is via the post method. Logout is with a get.
   String button = request.getParameter("button");
   button = button == null ? "" : button;

   if (button.contentEquals("login")) {
      doLogin(request, response);
   } else if (button.contentEquals("newAccount")) {
      doCreateUser(request, response);
   }
}

The doLogin method checks to make sure the username passed in from the request headers
is not empty, and then authenticates the user by creating a new UserDAO instance by calling
UserDAO.validateLogin. If authentication fails, the UserLoginException is caught and the
user is redirect to the login page to enter their credentials again. If authentication succeeds, the
user is forwarded to the playlist page.

private void doLogin (HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
   String username = request.getParameter("username");
   String password = request.getParameter("password");
Step 4: Adding users and custom playlists

```java
if (username.isEmpty()) {
    request.setAttribute("error", "Username Can Not Be Blank");
    getServletContext().getRequestDispatcher("/login.jsp").forward(request, response);
    return;
}

try {
    UserDAO user = UserDAO.validateLogin(username, password);
    HttpSession httpSession = request.getSession(true);
    httpSession.setAttribute("user", user);
}
catch (UserLoginException e) {
    // Go back to the user screen with an error
    request.setAttribute("error", "Username or Password is Invalid");
    getServletContext().getRequestDispatcher("/login.jsp").forward(request, response);
    return;
}

response.sendRedirect("playlists");
}
```

The `doCreateUser` attempts to create a new user with the specified username and password. After checking to make sure the username is not empty, a new `UserDAO` instance is created by calling `UserDAO.addUser`. If the username already exists, the `playlist.exceptions.UserExistsException` is caught and the user is forwarded back to the login page. If creating the new user succeeds, the user is forwarded to the playlist page.
// Create the user's login session so this application recognizes the user as having logged in
  httpSession.setAttribute("user", newUser);

} catch (UserExistsException e) {

  // Go back to the user screen with an error
  request.setAttribute("error", "User Already Exists");
  getServletContext().getRequestDispatcher("/login.jsp").forward(request, response);
  return;

} response.sendRedirect("playlists");

User logout events are handled by the doGet method, which responds to HTTP GET requests. If the user pressed the logout button, the private doLogout method is called.

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
  StatisticsDAO.increment_counter("page hits: login");

  String button = request.getParameter("button");
  button = button == null ? "" : button;

  if (button.contentEquals("logout")) {
    doLogout(request, response);
  } else {
    getServletContext().getRequestDispatcher("/login.jsp").forward(request, response);
  }
}

Adding the playlist controllers

The playlist.controller.PlaylistsServlet and playlist.controller.PlaylistTracksServlet classes handle the requests from the views for managing playlists. The PlaylistsServlet class handles creating and deleting playlists for logged in users.

public class PlaylistsServlet extends HttpServlet {

  protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    doAction(request, response);
  }
}
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    doAction(request, response);
}

private void doAction(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    HttpSession httpSession = request.getSession(true);
    UserDAO user = (UserDAO) httpSession.getAttribute("user");

    // If we're not logged in, go to the login page
    if (user == null) {
        request.setAttribute("error", "Not Logged In");
        response.sendRedirect("login");
        return;
    }

    UserDAO userFromDB = UserDAO.getUser(user.getUsername());

    String button = request.getParameter("button");
    String playlist = request.getParameter("pl");

    if (button != null) {
        if (button.contentEquals("deletePlaylist")) {
            // Delete the playlist
            new PlaylistDAO(user.getUsername(), playlist).deletePlayList();

            // Force a re-read in this case
            response.sendRedirect("playlists");
            return;
        } else if (button.contentEquals("Add")) {
            if (playlist != null) {
                doAddPlaylist(userFromDB, playlist);
            }
        }
    }

    request.setAttribute("username", userFromDB.getUsername());
    request.setAttribute("playlist_names", userFromDB.getPlaylist_names());
    getServletContext().getRequestDispatcher("/playlists.jsp").forward(request, response);
}

private void doAddPlaylist(UserDAO user, String playlistName) {
    PlaylistDAO.createPlayList(user, playlistName);
}
The PlaylistTracksServlet handles adding and deleting tracks from a particular playlist.

```java
public class PlaylistTracksServlet extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

        // We add playlists with a post method
        HttpSession httpSession = request.getSession(true);

        String button = request.getParameter("button");
        String playlist_name = request.getParameter("pl");
        UserDAO user = (UserDAO) httpSession.getAttribute("user");

        PlaylistDAO playlist = PlaylistDAO.getPlaylistForUser(user.getUsername(), playlist_name);

        if (button != null) {
            if (button.contentEquals("addTrack")) {
                UUID track_id = UUID.fromString(request.getParameter("track_id"));
                doAddPlaylistTrack(playlist, track_id);
            }
        }

        request.setAttribute("username", user.getUsername());
        request.setAttribute("playlist", playlist);
        getServletContext().getRequestDispatcher("/playlist_tracks.jsp").forward(request, response);
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

        HttpSession httpSession = request.getSession(true);
        UserDAO user = (UserDAO) httpSession.getAttribute("user");

        // If we're not logged in, go to the login page
        if (user == null) {

            request.setAttribute("error", "Not Logged In");
            response.sendRedirect("login");
            return;
        }

        // Initialize the parameters that are returned from the web page
        //
        String playlist_name = request.getParameter("pl");
        PlaylistDAO playlist = PlaylistDAO.getPlaylistForUser(user.getUsername(), playlist_name);
    }
```
Step 4: Adding users and custom playlists

```java
String button = request.getParameter("button");
String deleteTrack = request.getParameter("deleteTrack");

//
// If a button was pressed, carry out the button's action
//

if (deleteTrack != null) {
    // Delete one track
    long sequence_no = Long.parseLong(deleteTrack);
    doDeleteTrack(playlist, sequence_no);
}

request.setAttribute("username", user.getUsername());
request.setAttribute("playlist", playlist);
getServletContext().getRequestDispatcher("/playlist_tracks.jsp").forward(request, response);
}

void doAddPlaylistTrack(PlaylistDAO playlist, UUID track_id) throws ServletException {
    // Grab the PlaylistTrack information from the DB
    TracksDAO track = TracksDAO.getTrackById(track_id);
    PlaylistDAO.PlaylistTrack newPlaylistTrack = new PlaylistDAO.PlaylistTrack(track);
    try {
        playlist.addTrackToPlaylist(newPlaylistTrack);
    } catch (Exception e) {
        throw new ServletException("Couldn't add track to playlist");
    }
}

void doDeleteTrack(PlaylistDAO playlist, long sequence_no) {
    playlist.deleteTrackFromPlaylist(sequence_no);
}
```

Changes from Step 3

To see all code changes from the step3 branch, enter the following command in a terminal in the playlist directory:
Building and running step 4

You will not modify Playlist to allow user logins and custom playlists.

**Prerequisites:**
You must have [set up your environment](page 6) before starting this step.

1. In a terminal, go to the playlist workspace.
   
   ```
   $ cd playlist
   ```

2. Check out the step4 branch using git.
   
   ```
   $ git checkout step4
   ```

3. Run the cqlsh command.
   
   ```
   $ cqlsh
   ```

4. In the playlist keyspace create the **users** and **playlist_tracks** tables.
   
   ```
   use playlist;
   create table users (username text primary key, password text, 
   playlist_names set<text>);
   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 ));
   ```

5. Quit cqlsh.
   
   ```
   exit;
   ```

6. Build and run Playlist using **mvn**.
   
   ```
   $ mvn verify cargo:run
   ```

7. In a web browser, navigate to or refresh: [http://localhost:8080/playlist](http://localhost:8080/playlist)
   
   Now you can create new users and playlists.

**What's next:**
Step 4: Adding users and custom playlists

Proceed to Step 5: Add statistics views to monitor Playlist (page 39).
Step 5: Add statistics views to monitor Playlist

We will now statistics-gathering components to monitor how often users view the different parts of Playlist.

The page hits will be stored in a table with a special counter data type column. A Cassandra counter column stores numbers that are changed in increments. Counters must be stored in a separate table that consists only of the primary key column and the counter column. Counters need to be implemented this way so Cassandra can coordinate count values across clusters. Updates to a counter need to be replicated to other nodes.

Changes to the Playlist schema

The statistics will be stored in a new table, statistics, with two columns: counter_name (the primary key) and counter_value of type counter. This structure allows you to define different counters for each page in the application.

The CQL used to create the table is:

```
CREATE TABLE statistics (counter_name text PRIMARY KEY, counter_value counter);
```

<table>
<thead>
<tr>
<th>statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter_name</td>
</tr>
<tr>
<td>counter_value</td>
</tr>
</tbody>
</table>

Changes to the Playlist model

A new model class, playlist.model.StatisticsDAO is used to increment counters that are placed throughout the controller methods of the application. The two methods in StatisticsDAO used to change the counter value are increment_counter and decrement_counter, with the counter name as the parameter.

```java
public static void increment_counter(String counter_name) {
    String queryText = "UPDATE statistics set counter_value = counter_value + 1 where counter_name = ?";
    PreparedStatement preparedStatement = getSession().prepare(queryText);
    BoundStatement boundStatement = preparedStatement.bind(counter_name);
    getSession().execute(boundStatement);
}

public static void decrement_counter(String counter_name) {
```
In our case, the CQL queries increases or decreases the counter by one whenever the increase_counter or decrement_counter methods are called.

Adding counters to the controllers

Counters are added to the controller servlets to track page hits, users, and playlists. Here is how the home page tracks page views in HomeServlet:

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    StatisticsDAO.increment_counter("page hits: home");
    String javaVersion = System.getProperty("java.version");
    CassandraInfo cassandraInfo = new CassandraInfo();

    request.setAttribute("java_version", javaVersion);
    request.setAttribute("cassandra_info", cassandraInfo);
    getServletContext().getRequestDispatcher("/home.jsp").forward(request, response);
}

The counter name is page hits: home.

In LoginServlet there are counters to log page hits to the login page, failed login attempts, and successful logins.

Viewing the statistics

All the statistics are viewable by clicking on the Statistics link on the home page. The controller class for the view and model is playlist.controller.StatisticsServlet.

Changes from step 4

To see all code changes from the step4 branch, enter the following command in a terminal in the playlist directory:
Building and running step 5

Add a table to store the statistics collected for the Playlist application, and build and run the modified application.

Prerequisites:
You must have set up your environment (page 6) before starting this step.

1. In a terminal, go to the playlist workspace.
   ```bash
cd playlist
   ```

2. Check out the step5 branch using git.
   ```bash
git checkout step5
   ```

3. Run the cqlsh command.
   ```bash
cqlsh
   ```

4. In the playlist keyspace create the users and playlist_tracks tables.
   ```sql
   use playlist;
   CREATE TABLE statistics (counter_name text PRIMARY KEY, counter_value counter);
   ```

5. Quit cqlsh.
   ```bash
   exit;
   ```

   ```bash
   mvn verify cargo:run
   ```

7. In a web browser, navigate to: http://localhost:8080/playlist
   On the home page click Statistics to view the current statistics. As you browse through the database, create new playlists, and attempt to log-in as a user, the statistics are updated.

What's next:
Step 5: Add statistics views to monitor Playlist

Proceed to Step 6: Optimize the Playlist application (*page 43*).
Step 6: Optimize the Playlist application

Now that the Playlist application is functionally complete, you will optimize its performance by limiting the number of items returned in our queries and enabling caching in Cassandra.

The location of the `cassandra.yaml` file depends on the type of installation:

<table>
<thead>
<tr>
<th>Package installations</th>
<th>/etc/cassandra/cassandra.yaml</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tarball installations</td>
<td>install_location/resources/cassandra/conf/cassandra.yaml</td>
</tr>
</tbody>
</table>

Using paging in queries

When fetching large result sets, Cassandra caches the entire result set and sends it to the application in a single block. The Cassandra Java driver, however, has a paging feature that will retrieve the results of a query in configurable chunks. In this step, we will configure the paging feature to return sets 200 rows at a time for the query that returns the largest set, in `TracksDAO.listSongsByGenre`.

```java
public static List<TracksDAO> listSongsByGenre(String genre, int num_tracks) {
    String queryText = "SELECT * FROM track_by_genre WHERE genre = ? LIMIT ?";
    PreparedStatement preparedStatement = getSession().prepare(queryText);
    BoundStatement boundStatement = preparedStatement.bind(genre, num_tracks);
    boundStatement.setFetchSize(200);
    ResultSet results = getSession().execute(boundStatement);

    List<TracksDAO> tracks = new ArrayList<>();
    for (Row row : results) {
        tracks.add(new TracksDAO(row));
    }

    return tracks;
}
```

The `setFetchSize` method is called on the statement object and set to the number of rows returned at a time. Choosing the correct number of rows depends on your data set and your application. Setting the paging size too small results in more queries being resent to Cassandra as the data set is traversed, resulting in poor performance.

Enabling row caching in Cassandra

When you enable row caching on a table, Cassandra will detect frequently accessed partitions and store rows of data into a RAM cache. A cache increases the performance of queries that access those rows by limiting the number of times Cassandra needs to read from disk storage.
Step 6: Optimize the Playlist application

You can configure how many rows to cache per partition by setting the `rows_per_partition` attribute of the `caching` option when creating or altering a table. Setting `rows_per_partition` to `ALL` caches all the rows in the partition.

```sql
CREATE TABLE my_table (
  id uuid PRIMARY KEY,
  status text
) WITH caching = {'rows_per_partition':'100'}
```

In this case, you will alter the `track_by_genre` and `track_by_artist` tables to cache the first 100 rows on each partition.

```sql
ALTER TABLE track_by_genre WITH caching = {'rows_per_partition':'100'};
ALTER TABLE track_by_artist WITH caching = {'rows_per_partition':'100'};
```

Using a row cache requires more memory on each node. The amount of memory Cassandra dedicates to the row cache is configured in the `row_cache_size_in_mb` option in `cassandra.yaml`.

Cassandra also supports `key caching`, which helps Cassandra find the location of a partition on disk to decrease disk seek times. Key caches are enabled by default, so we don’t need to explicitly turn the key cache on.

Changes from Step 5

To see all code changes from the `step5` branch, enter the following command in a terminal in the `playlist` directory:

```
$ git diff step5..step6
```

Building and running step 6

Optimize the performance of the queries in Playlist by enabling row caching and using paging in the queries that return large data sets.

The location of the `cassandra.yaml` file depends on the type of installation:

<table>
<thead>
<tr>
<th>Package installations</th>
<th><code>/etc/cassandra/cassandra.yaml</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tarball installations</td>
<td><code>install_location/resources/cassandra/conf/cassandra.yaml</code></td>
</tr>
</tbody>
</table>

Prerequisites:

You must have set up your environment (page 6) before starting this step.

1. In a terminal, go to the `playlist` workspace.

   ```
   $ cd playlist
   ```
2. Check out the step6 branch using `git`.

   `$ git checkout step6`

3. Run the `cqlsh` command.

   `$ cqlsh`

4. In the `playlist` keyspace alter the `tracks_by_genre` and `tracks_by_artist` tables to cache the first 100 rows in the partition.

   ```
   use playlist;
   ALTER TABLE track_by_genre WITH caching = 
   {'rows_per_partition':'100'};
   ALTER TABLE track_by_artist WITH caching = 
   {'rows_per_partition':'100'};
   ```

5. Quit `cqlsh`.

   `exit;`

6. Using a text editor, modify the `row_cache_size_in_mb` option in `cassandra.yaml` to enable the row cache and set the amount of memory in the row cache to 50 MB.

   ```
   # Default value is 0, to disable row caching.
   row_cache_size_in_mb: 50
   ```

7. **Restart** (page) DataStax Enterprise.

   For package installs on Debian or Ubuntu:

   `$ sudo service dse restart`

   For tarball installs, see the topics on stopping (page) and starting (page) nodes.

8. Build and run Playlist using `mvn`.

   `$ mvn verify cargo:run`

Navigate through the artist or genre catalog several times. This will add the table rows to the row cache.

10. In a separate terminal use the `nodetool info` command to see the row cache entries and size.

```
$ nodetool info
Row Cache: entries 3, size 84.82 KB, capacity 50 MB, 4 hits, 7 requests, 0.571 recent hit rate, 0 save period in seconds
```

Each time you click on a genre or artist the cache size will increase.