Documento de Diseño: Sistema de Gestión de Streaming de Música con Apache Cassandra

Proyecto: Sistema de Gestión de Streaming de Música

Tecnología: Apache Cassandra

Integrantes:

Huaman Perales Gerson Anthony

Barrera Jeremías Jazmin Areli

Quispe Guevara Alexis

Sanchez Valenzuela , Geraldin Angela

Mancilla Flores Nataly

Fecha: 21/11/2025

1. Objetivo del Proyecto

El objetivo es diseñar e implementar un sistema de gestión de streaming de música utilizando Apache Cassandra. El sistema deberá manejar de manera eficiente grandes volúmenes de datos distribuidos, enfocándose en la alta disponibilidad y escalabilidad que ofrece Cassandra. El diseño del modelo de datos estará orientado a las consultas (Query-First Design) para garantizar un rendimiento óptimo.

2. Requisitos del Sistema y Patrones de Acceso

Tras analizar el dominio del problema, se han identificado las siguientes entidades y los patrones de acceso (consultas) principales que el sistema debe soportar:

- **Usuarios:**
    - **Requisito:** Almacenar la información de cada usuario.
    - **Patrón de Acceso:** Obtener los detalles de un usuario a partir de su ID (user_id).
- **Artistas:**
    - **Requisito:** Almacenar información detallada de los artistas.
    - **Patrón de Acceso:** Buscar la información completa de un artista por su ID (artist_id).
- **Canciones:**
    - **Requisito:** Almacenar el catálogo de canciones y su relación con los artistas.
    - **Patrón de Acceso 1:** Obtener todas las canciones de un artista específico.
    - **Patrón de Acceso 2:** Ordenar las canciones de un artista por su año de lanzamiento (de más reciente a más antiguo).
    - **Patrón de Acceso 3 (Secundario):** Buscar canciones por género musical.
- **Playlists:**
    - **Requisito:** Permitir a los usuarios crear y gestionar sus propias playlists de canciones.
    - **Patrón de Acceso 1:** Obtener todas las playlists creadas por un usuario específico.
    - **Patrón de Acceso 2 (Secundario):** Buscar playlists por su nombre.
- **Historial de Reproducciones:**
    - **Requisito:** Registrar cada vez que un usuario reproduce una canción.

    - **Patrón de Acceso:** Consultar el historial de canciones que un usuario ha escuchado, permitiendo filtrar por un rango de fechas.

- **Contador de Reproducciones:**
    - **Requisito:** Llevar la cuenta del número total de reproducciones de cada canción.
    - **Patrón de Acceso:** Incrementar el contador de una canción cada vez que se reproduce y consultar el total de reproducciones.

3. Diseño del Modelo de Datos (Familias de Columnas)

Basado en el principio de diseño orientado a consultas, se ha creado una tabla (Familia de Columnas) específica para cada patrón de acceso principal. Esto evita la necesidad de JOINs y garantiza lecturas de alto rendimiento.

A continuación, se detalla la estructura y justificación de cada tabla.

Tabla 1: artists

- **Propósito:** Almacenar la información de cada artista. Es una tabla de búsqueda simple.
- **Definición CQL:**

```
CREATE TABLE IF NOT EXISTS artists (
 artist_id uuid PRIMARY KEY,
 info frozen<artist_info>
);
-- UDT utilizado:
CREATE TYPE IF NOT EXISTS artist_info (
   name text,
   nationality text,
   birth_year int
);
```

- **Clave de Partición:** (artist_id).
- **Justificación:** Se utiliza artist_id como clave de partición para permitir la recuperación directa y eficiente de los datos de un artista. El uso de un UDT (artist_info) permite agrupar datos relacionados de forma lógica.

Tabla 2: songs_by_artist

- **Propósito:** Responder a la consulta: "Obtener todas las canciones de un artista, ordenadas por año".
- **Definición CQL:**

```
CREATE TABLE IF NOT EXISTS songs_by_artist (
 artist_id uuid,
 release_year int,
 song_id uuid,
 song_name text,
 duration int,
 genre text,
 PRIMARY KEY ((artist_id), release_year, song_id)
) WITH CLUSTERING ORDER BY (release_year DESC, song_id ASC);
```

- **Clave de Partición:** (artist_id).
- **Justificación:** Esta decisión es crucial. Al particionar por artist_id, nos aseguramos de que todas las canciones de un mismo artista se almacenen juntas en la misma partición, lo que hace que la consulta sea extremadamente rápida al leer desde un solo nodo.
- **Claves de Clustering:** (release_year, song_id).
- **Justificación:** Las claves de clustering ordenan los datos dentro de la partición.
    1. release_year DESC: Se ordena por año de lanzamiento en orden descendente. Esto permite obtener eficientemente las canciones más nuevas de un artista.
    2. song_id ASC: Actúa como un desempate para asegurar que cada fila sea única.

Tabla 3: playlists_by_user

- **Propósito:** Responder a la consulta: "Obtener todas las playlists de un usuario".
- **Definición CQL:**

```
CREATE TABLE IF NOT EXISTS playlists_by_user (
```

```
 user_id uuid,
 playlist_id uuid,
 playlist_name text,
 created_at timestamp,
 songs list<text>,
 PRIMARY KEY (user_id, playlist_id)
);
```

- **Clave de Partición:** (user_id).
- **Justificación:** Agrupa todas las playlists de un usuario en una única partición para una recuperación instantánea de sus colecciones.
- **Clave de Clustering:** (playlist_id).
- **Justificación:** Organiza las playlists de un usuario por su identificador único.

Tabla 4: history_by_user_date

- **Propósito:** Responder a la consulta: "Mostrar el historial de reproducciones de un usuario en un rango de fechas".
- **Definición CQL:**

```
CREATE TABLE IF NOT EXISTS history_by_user_date (
 user_id uuid,
 play_date date,
 song_id uuid,
 device text,
 PRIMARY KEY ((user_id), play_date, song_id)
);
```

- **Clave de Partición:** (user_id).
- **Justificación:** Permite consultar eficientemente el historial de un único usuario.
- **Claves de Clustering:** (play_date, song_id).

- **Justificación:** Ordena el historial por fecha, lo que permite consultas de rango eficientes (ej. WHERE play_date > '2025-11-01'). song_id garantiza la unicidad de la fila.

Tabla 5: play_count_by_song

- **Propósito:** Mantener un contador de cuántas veces se ha reproducido cada canción.
- **Definición CQL:**

```
CREATE TABLE IF NOT EXISTS play_count_by_song (
 song_id uuid PRIMARY KEY,
 play_count counter
);
```

- **Clave de Partición:** (song_id).
- **Justificación:** Proporciona acceso directo para leer o actualizar el contador de una canción. El uso del tipo counter está optimizado en Cassandra para operaciones de incremento atómicas y de alta concurrencia, evitando conflictos de lectura-escritura.

4. Documentación de Consultas Principales Soportadas

El modelo de datos propuesto está diseñado para soportar las siguientes consultas de manera eficiente:

1. **Buscar canciones de un artista específico (ordenadas por más reciente):**

   ```
   SELECT song_name, release_year, genre FROM songs_by_artist WHERE artist_id = ?;
   ```

2. **Buscar las playlists de un usuario:**

   ```
   SELECT playlist_name, created_at FROM playlists_by_user WHERE user_id = ?;
   ```

3. **Consultar el historial de un usuario para una fecha específica:**

   SELECT song_id, device FROM history_by_user_date WHERE user_id = ? AND play_date = '2025-11-21';

4. **Incrementar el contador de reproducción de una canción:**

   UPDATE play_count_by_song SET play_count = play_count + 1 WHERE song_id = ?;

5. **Obtener el total de reproducciones de una canción:**

   SELECT play_count FROM play_count_by_song WHERE song_id = ?;

6. **Buscar canciones por género (usando índice secundario):**

   CREATE INDEX IF NOT EXISTS idx_songs_genre ON songs_by_artist(genre);

   SELECT song_name, artist_id FROM songs_by_artist WHERE genre = 'Pop';