Spotify Datasets Application

Eddie Ho, Haoyu Fu, Yixuan Zhang

Key Features

- The objective of this project is to develop a database application that effectively stores and analyzes the data obtained from the Spotify datasets.
- Ensure fast and reliable access to the data.
- Tracks and artists information retrieval based on the filter conditions (e.g. name, track's modality, estimated time signature, key signature, etc.)
- Incorporate a recommendation system that suggests similar tracks based on the query results.

Data Systems

cassandra

neo4j

redis

Data Processing

Data Processing

- Original datasets
  - Tabular (Spotify Dataset 1921-2020)
    - Artists (id, #followers, genres, name, popularity)
    - Tracks (id, name, popularity, duration_ms, explicit, artists, id_artists, release_date, danceability, energy, key, loudness, speechiness, acousticness, …)
  - Network (Spotify Artist Feature Collaboration Network)
    - Nodes (spotify_id, name, #followers, #popularity, genres, chart_hits)
    - Edges (id_0, id_1)

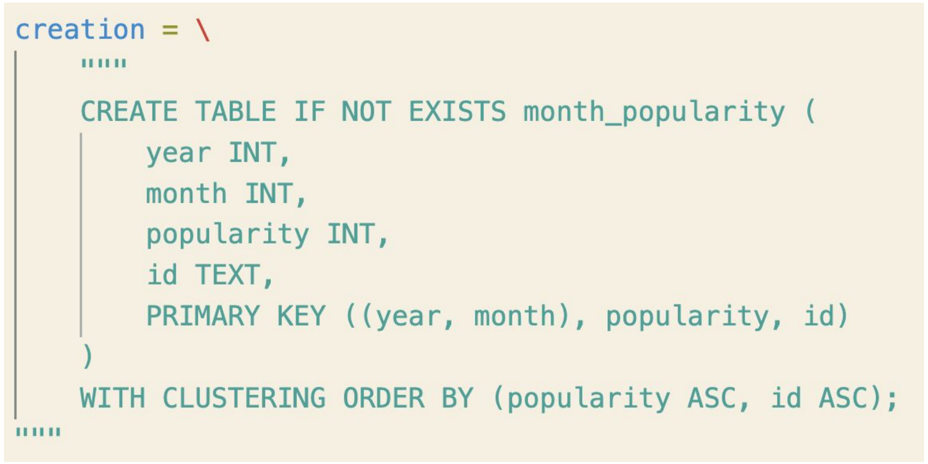
Data Processing

- Data Cleaning
  - Tabular
    - In Artists table, 11 rows with null #followers, and so we dropped those rows.
    - In Tracks table, 71 rows with null name, and so we dropped those rows.
    - Preprocessed genres column to work with COPY command in cqlsh.
  - Network
    - Dropped duplicated spotify_id and name in Nodes.csv
    - Changed the value format of the column id_artists in convenience for neo4j queries

Data Processing

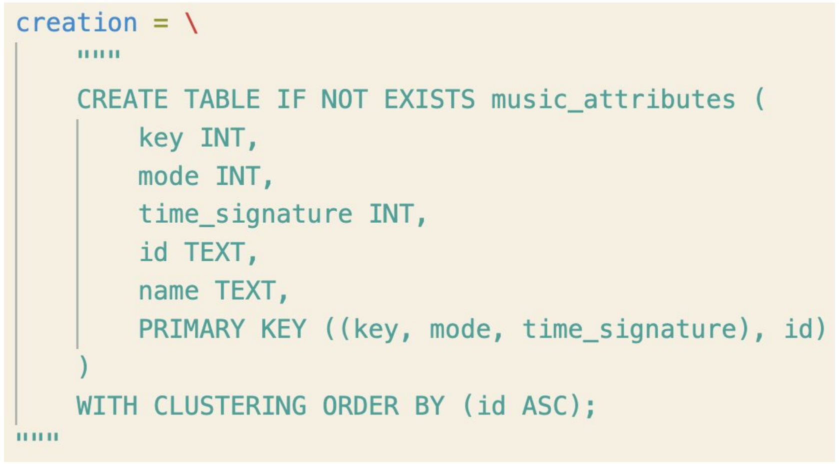
- Populate to databases
  - Cassandra
    - Output Cleaned data and copy them into docker container.
    - Create keyspace and table schema in Cassandra.

Example Populated Dataset



```
creation = \
    """
    CREATE TABLE IF NOT EXISTS month_popularity (
        year INT,
        month INT,
        popularity INT,
        id TEXT,
        PRIMARY KEY ((year, month), popularity, id)
    )
    WITH CLUSTERING ORDER BY (popularity ASC, id ASC);
"""
```

For function **popularity_by_month**



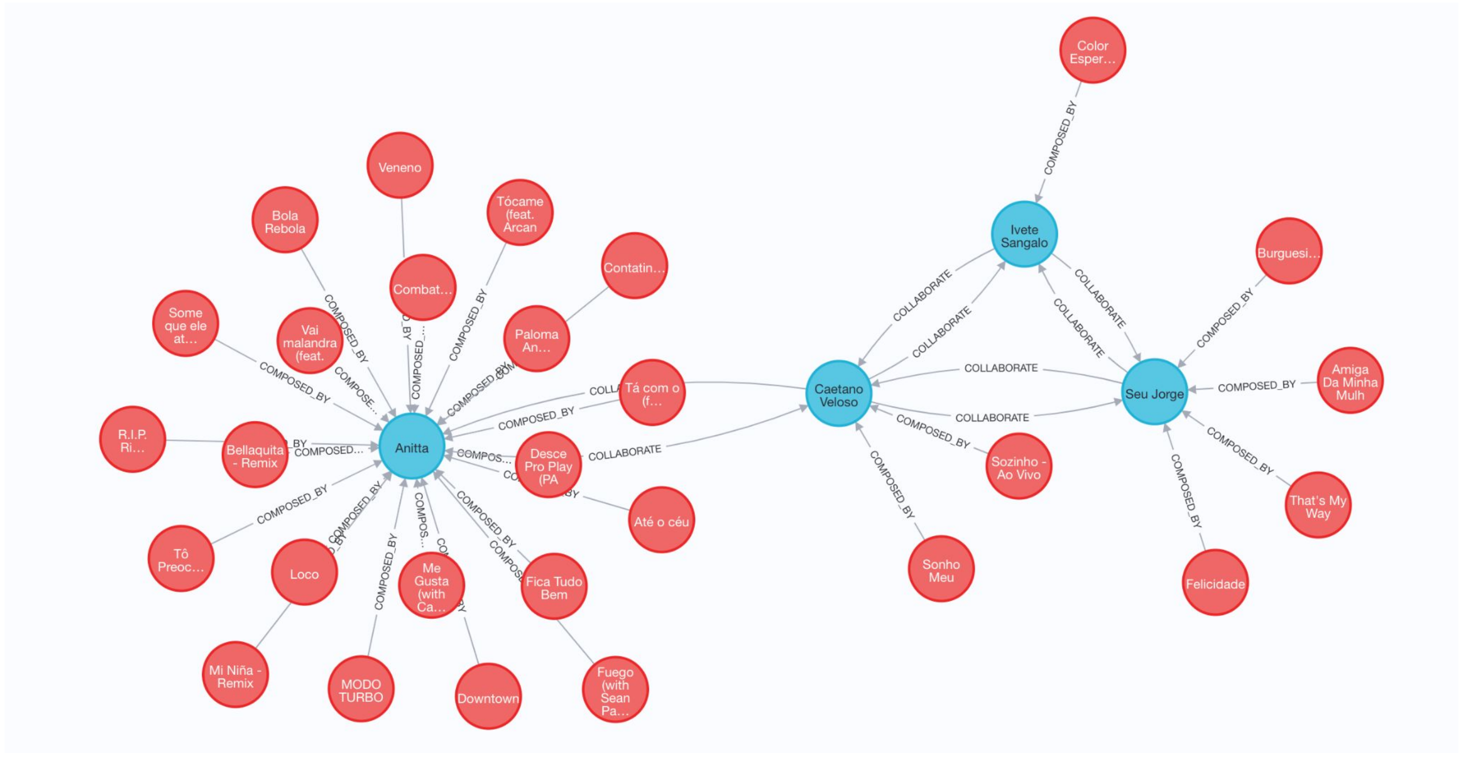
```
creation = \
    """
    CREATE TABLE IF NOT EXISTS music_attributes (
        key INT,
        mode INT,
        time_signature INT,
        id TEXT,
        name TEXT,
        PRIMARY KEY ((key, mode, time_signature), id)
    )
    WITH CLUSTERING ORDER BY (id ASC);
"""
```

For function **track_by_music_attributes**

Data Processing

- Populate to databases
  - Neo4j
    - Output Cleaned data and copy them into docker container.
      - Track nodes: cleaned tracks data
      - Artist nodes: cleaned nodes from Collaboration Network dataset
      - Collaboration edges: cleaned edges from Collaboration Network dataset
    - For runtime consideration, only artists with popularity > 60 and their tracks are populated.
      - 4,069 artists nodes
      - 166,371 tracks nodes
      - 220,521 edges
      - Full datasets (156,423 artists nodes) takes more than 10 hours to populate
    - Use LOAD CSV command for batch loading.

Example Populated Dataset



Functions

Functions Ⅰ (PostgreSQL with Redis)

- search_artist(id, followers, genres, name, popularity, projection, limit)
    - Input:
        - id (string, optional), followers (int, optional), genres (list as a string, optional), name (string, optional), popularity (int, optional), projection (list, optional), limit (int, optional)
    - Output:
        - Rows of artist data retrieved from a PostgreSQL database or cache in Redis.
    - Implementation:
        - This function retrieves artist data from a PostgreSQL database using specific input parameters. Before executing a query, it checks if the result is already cached in Redis. If found, it returns the cached result, else, it runs the query on the database, stores the result in Redis for future use, and then returns the result. The 'projection' parameter determines the attributes to be included in the result, and the 'limit' parameter defines the maximum number of rows to be returned.

Functions Ⅱ (Cassandra with Redis)

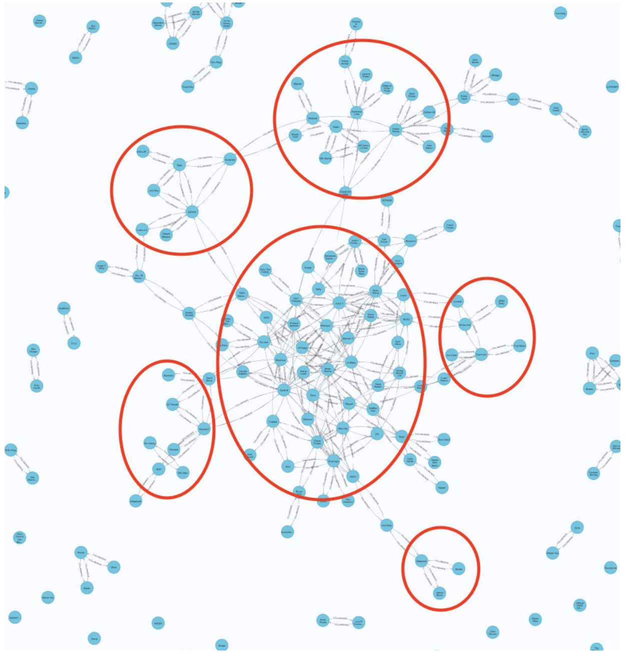
- popularity_by_month(year, month)
  - Input:
    - year (int), month (int)
  - Output:
    - The distribution of track popularity levels for a specific month, retrieved from a Cassandra database or cached in Redis.
  - Implementation:
    - The search_popularity_distribution_by_month function uses the given year and month to retrieve the popularity distribution of tracks for that specific month. The function first checks Redis to see if the query result is already cached. If it is, the cached data is returned. If not, the function performs a query on the Cassandra database, caches the result in Redis for potential future use, and then returns it. The returned data represents the number of tracks at each level of popularity for the specified month.

Functions Ⅲ (Cassandra with Redis)

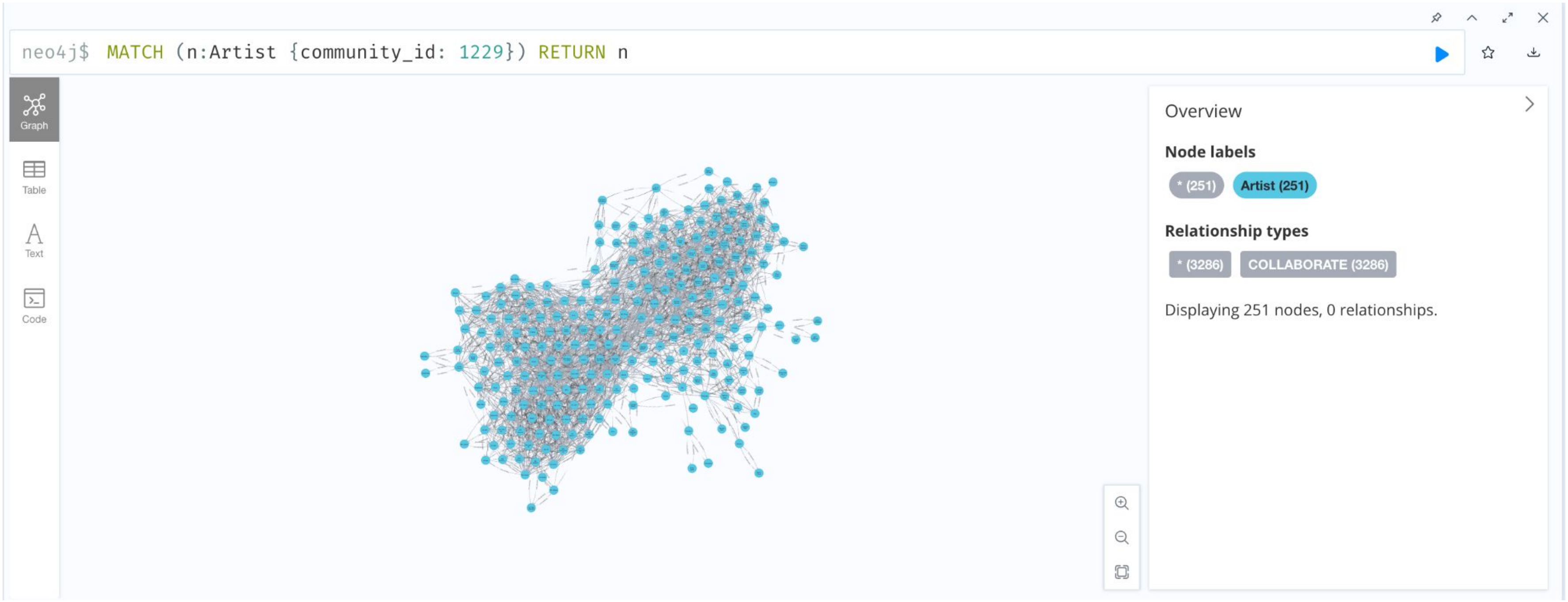
- track_by_music_attributes(key, mode, time_signature, limit)
  - Input:
    - key (int), mode (int), time_signature (int), limit (int, optional)
  - Output:
    - Track data for specific musical attributes (key, mode, and time signature) retrieved from a Cassandra database or cached in Redis.
  - Implementation:
    - The search_track_by_music_attributes function uses the provided key, mode, and time signature to find matching tracks. It first checks Redis to see if the result of this specific query is already cached. If it is, the cached data is returned. If not, the function executes the query on the Cassandra database, caches the result in Redis for future reference, and then returns it. The returned data contains track ID and name that match the given musical attributes. If a limit is provided, it limits the number of rows returned.

Functions Ⅳ (Neo4j)

- louvain_cluster()
  - Helper function, called only once in the initialization process
  - Implementation:
    - Cluster the artist nodes into communities using Louvain algorithm from GDS
    - Add community_id as a property to every artist node



Example: A Single Community



```
neo4j$ MATCH (n:Artist {community_id: 1229}) RETURN n
```

Overview

Node labels

* (251) Artist (251)

Relationship types

* (3286) COLLABORATE (3286)

Displaying 251 nodes, 0 relationships.

Functions Ⅳ (Continued)

- recommend_track(track_id)
  - Recommend similar track(s) based on the given track id.
  - Input: Track id (string)
  - Output: List of recommended track ids, sorted by descending similarity
  - Implementation:
    i. Find all the tracks from the author community that the input track's author belongs to.
    ii. Compare the input track and the tracks found in step ii, return the tracks with largest **Pearson similarity** with musical properties including **danceability, energy, key, loudness, liveness,** and **tempo**

Demo

Q&A