SPOTIFY FREQUENT ITEMSET MINING & RECOMMENDATION ENGINE

Aditya Gupta(210150002) & Anuj Gupta(200121005)

Video link - https://youtu.be/5VXcg2gFUOo

Outline

- Dataset & audio features
- Content based Recommendations.
- Vector Similarity Search
- Frequent Items Mining
- A-priori & FPGrowth in PySpark
- Conclusion

Spotify® Million Playlist Dataset

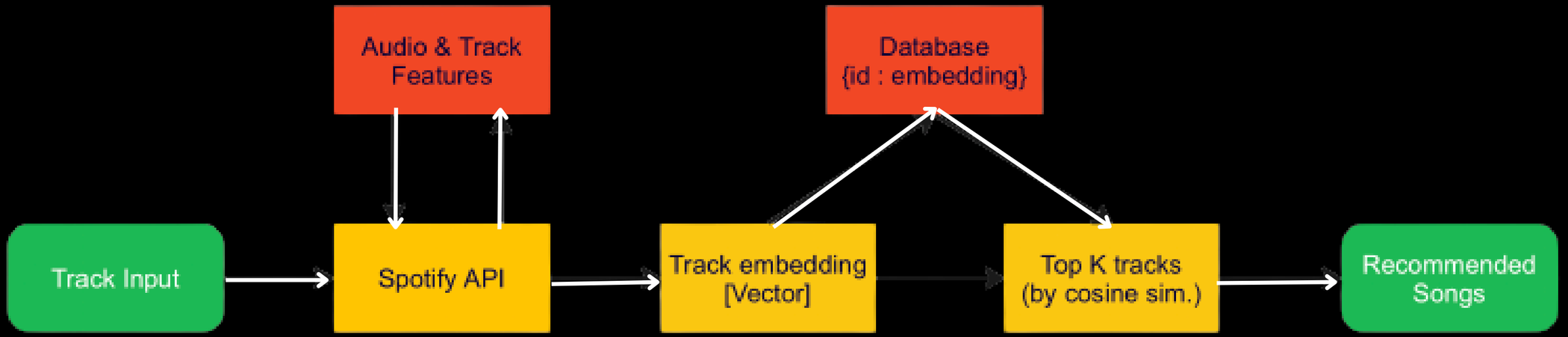
- **1 million playlists**
- **2.2 million Unique Tracks**

Dataset is provided in json format. It consists of 1M playlists collected from US users between 2010 & 2017.


```
{
        "name": "musical",
        "collaborative": "false",
        "pid": 5,
        "modified_at": 1493424000,
        "num_albums": 7,
        "num_tracks": 12,
        "num_followers": 1,
        "num_edits": 2,
        "duration_ms": 2657366,
        "num_artists": 6,
        "tracks": [
            {
                "pos": 0,
                "artist_name": "Degiheugi",
                "track_uri": "spotify:track:7vqa3sDmtEaVJ2gcvxtRID",
                "artist_uri": "spotify:artist:3V2paBXEoZIAhfZRJmo2jL",
                "track_name": "Finalement",
                "album_uri": "spotify:album:2KrRMJ9z7Xjoz1Az4O6UML",
                "duration_ms": 166264,
                "album_name": "Dancing Chords and Fireflies"
            },
            {
                "pos": 1,
                "artist_name": "Degiheugi",
                "track_uri": "spotify:track:23EOmJivOZ88WJPUbIPjh6",
                "artist_uri": "spotify:artist:3V2paBXEoZIAhfZRJmo2jL",
                "track_name": "Betty",
                "album_uri": "spotify:album:3lUSlvjUoHNA8IkNTqURqd",
                "duration_ms": 235534,
                "album_name": "Endless Smile"
            },
            {
                "pos": 2,
                "artist_name": "Degiheugi",
                "track_uri": "spotify:track:1vaffTCJxkyqeJY7zF9a55",
                "artist_uri": "spotify:artist:3V2paBXEoZIAhfZRJmo2jL",
                "track_name": "Some Beat in My Head",
                "album_uri": "spotify:album:2KrRMJ9z7Xjoz1Az4O6UML",
                "duration_ms": 268050,
                "album_name": "Dancing Chords and Fireflies"
            },
        ],

    }
```

Recommendation Methodology



ENDPOINT https://api.spotify.com/v1/audio-features/{id}

id 11dFghVXANMlKmJXsNCbNl

- **RESPONSE SAMPLE**


```
{
  "acousticness": 0.00242,
  "analysis_url": "https://api.spotify.com/v1/audio-analysis/2takcwOaAZWiXQijPHIx7B",
  "danceability": 0.585,
  "duration_ms": 237040,
  "energy": 0.842,
  "id": "2takcwOaAZWiXQijPHIx7B",
  "instrumentalness": 0.00686,
  "key": 9,
  "liveness": 0.0866,
  "loudness": -5.883,
  "mode": 0,
  "speechiness": 0.0556,
  "tempo": 118.211,
  "time_signature": 4,
  "track_href": "https://api.spotify.com/v1/tracks/2takcwOaAZWiXQijPHIx7B",
  "type": "audio_features",
  "uri": "spotify:track:2takcwOaAZWiXQijPHIx7B",
  "valence": 0.428
}
```


Spotify Audio Features

- Extracted audio features from response.
- Created vector embeddings of them.
- Then we used them for vector similarity search to find smilar tracks.

First Approach using Mongodb

- Used Mongodb to store and query vectors for song recommendation.
- Mongodb iterates over all the tracks in the dataset and compute similarity, which is too slow.
- It took approximately 6 seconds to iterate over 35k vectors.
- It would’ve taken 2 min. per query to compute similarity in subset of original dataset.

Optimized approach using Vector Database

- We used Pinecone database, which is vector database to efficiently store, index and retrieve high dimensional vectores.
- Pinecone uses vector embeddings to represent data points in high-dimensional spaces.
- It creates a vector index, which is based on approximate nearest-neighbor search algorithms to find similar vectors quickly, even in high-dimensional spaces.
- So we used Pinecone to store vectors of audio features to query dataset faster.

Amount of Data required

Using Embeddings

100k Playlists
(OR)
650k Tracks

Requires ~1GB
and 4hours to create
embeddings

Without embeddings

100k Playlists
(OR)
650k Tracks

Requires only 150MB to
store and can compute
frequent patterns in
~15mins(FPGrowth algo)

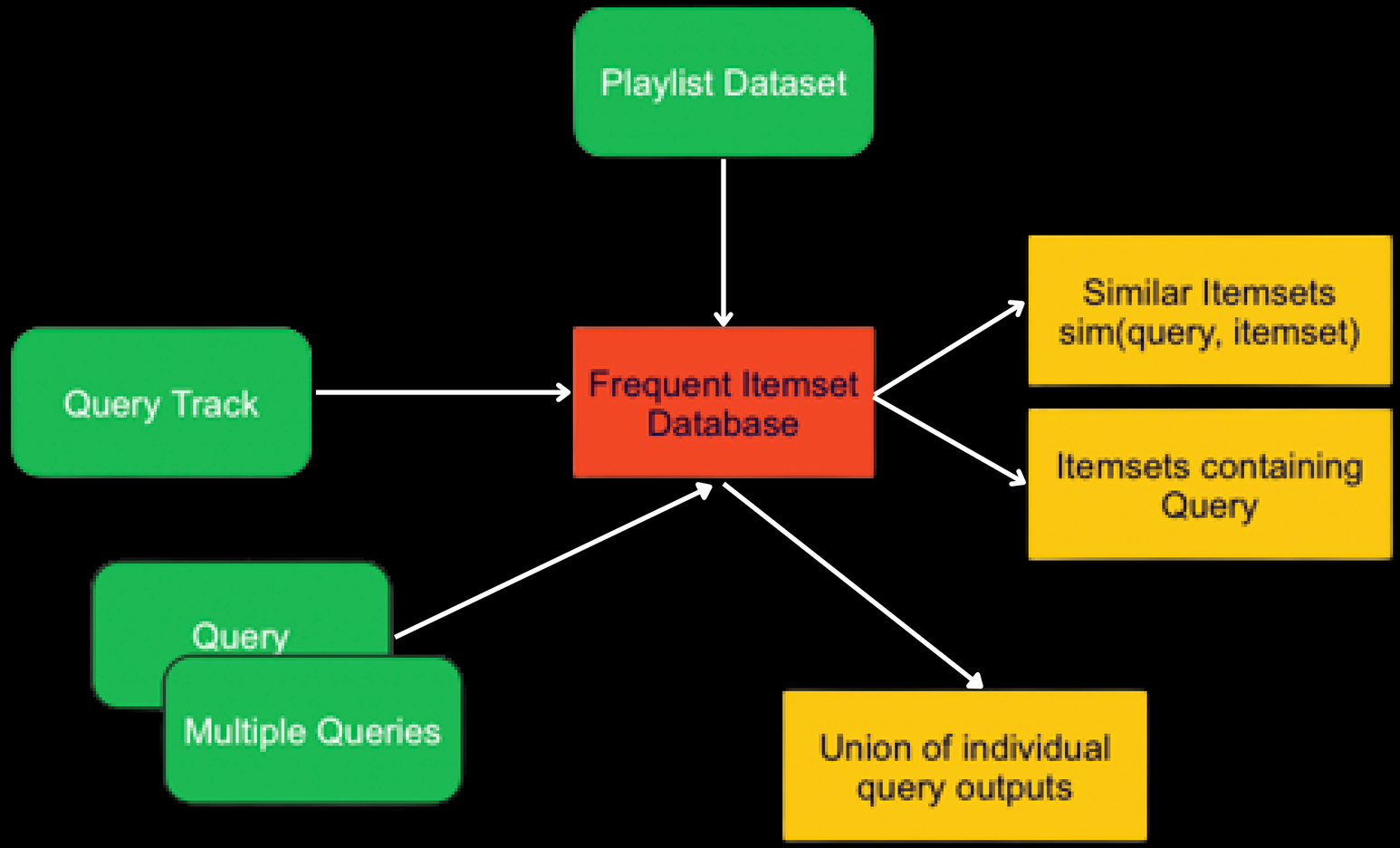
Recommendation based on frequent itemsets

- Rather than recommending songs purely based on content/features, we can also recommend songs based on how frequently they occur with other songs in playlists
- The general idea of finding sets of objects that occur in the same basket is called "**frequent itemset mining**".
- We can also do the same here, we can try to find sets of tracks that frequently occur together in playlists
- To implement frequent items mining, we have implemented the **A-priori algorithm** in PySpark(using RDDs).

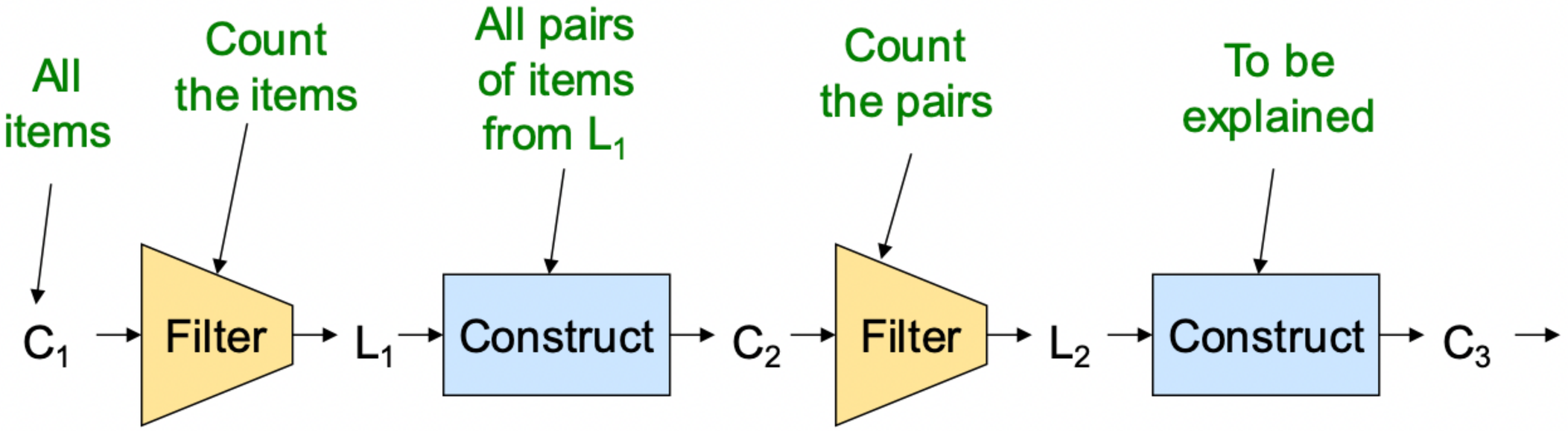
Recommendation based on frequent itemsets

- We can pre-compute frequent itemsets of different sizes and store them in a database. These frequent itemsets can be used as **suggested playlists based on popularity**.
- We can pre-compute embeddings of the frequent itemsets and return entire itemsets when a similar track is queried.
- We can also add features like allowing multiple query tracks and then return a union of frequent itemsets containing them.

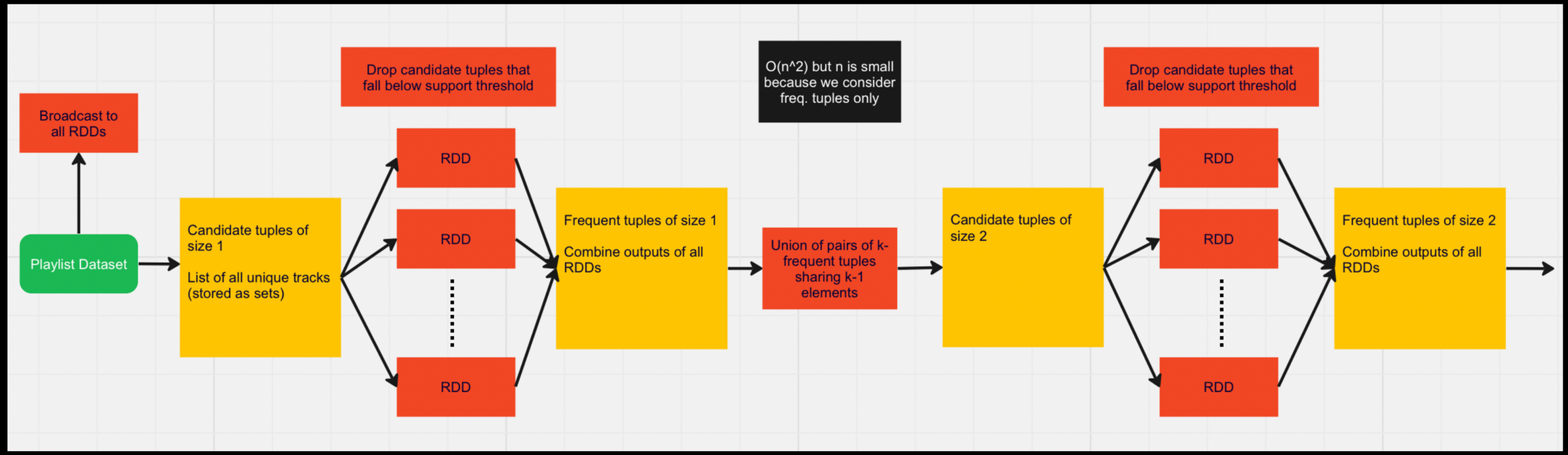
Recommendation based on frequent itemsets



- **For each *k*, we construct two sets of *k-tuples*** (sets of size *k*):
  - C_k = ***candidate k-tuples*** = those that might be frequent sets (support ≥ **s**) based on information from the pass for ***k*–1**
  - L_k = the set of truly frequent ***k***-tuples



Apriori Algorithm in PySpark



PySpark Outputs

```
itemset = apriori_dataload('data_txt_slices')
# apriori_support_estimates(itemsets)
apriori(itemset, 0.01)
```

✓ 37.5s

```
Example of frequent 1-tuples
[({'0CcQNd8CINkwQfe1RDtGV6'}, 97), ({'0jSMveIWvnhDIvzqN74Uc7'}, 61), ({'2HbKqm4o0w5wEeEFXm2sD4'}, 54), ({'4CJVkjo5WpmUAKp3R44LNb'}, 105), ({'

Example of frequent 2-tuples
[({'7yyRTcZmCiyzzJlNzGC9Ol', '5OOkp4U9P9oL23maHFHL1h'}, 66), ({'5OOkp4U9P9oL23maHFHL1h', '1xznGGDReH1oQq0xzbwXa3'}, 57), ({'5OOkp4U9P9oL23maH

Example of frequent 3-tuples
[({'7GX5flRQZVHRAGd6B4TmDO', '7KXjTSCq5nL1LoYtL7XAwS', '3a1lNhkSLSkpJE4MSHpDu9'}, 58), ({'7GX5flRQZVHRAGd6B4TmDO', '0VgkVdmE4gld66l8iyGjgx',
```

FPGrowth Algorithm

- FPGrowth is a frequent pattern mining algorithm, it is also parallelizable and implemented in PySpark.
- It utilizes a divide-and-conquer strategy, building an efficient data structure known as an FP-tree.
- The FP-tree eliminates the need for generating candidate sets, reducing computational overhead.
- FPGrowth is particularly efficient for large-scale datasets. Using FPGrowth, we were able to find frequent itemsets on roughly 100k playlists.

Conclusion

- To conclude, we successfully implemented a content based recommendation system for music tracks using audio & track features to create embeddings stored in a pinecone vectorDB
- Improved the recommendation experience by frequent itemset mining, through A-priori & FPGrowth.
- Successfully implemented A-priori from scratch in PySpark with necessary parallelisation of computation
- Developed a user friendly frontend for recommendations.

Thank You