GDMA Project: CDDB as Graph Database

Julian Schelb (1069967)

June 2022

1 Importing Data

This chapter describes design decisions made to map the relational CDDB database to a graph database. See Jupyter notebook `01_Import_Data.ipynb`.

1.1 Mappping of Tables

Entries in tables *artists*, *albums*, *songs* and *cds* represent entities and are mapped to nodes of appropriate type. The cross table *cdtracks* is mapped as relations between CDs and Songs with the property *track*. The cross table *artist2album* is mapped as a triangular relationships, using six relations each to connect Artists, Albums and CDs in both directions.

1.2 Bidirectional Relations

To consgtruct the queries necessary for subsequent tasks, it was structly speaking not necessary to create relations for both directions. However, doing so allows for more readable queries. In addition, the relational SQL database does not define a direction for relations either, and this property is thereby preserved.

1.3 Unique Identifiers

The unique identifiers present in the SQL database are preserved as node properties. Although Neo4j automatically assigns different IDs internally, this preserves compatibility with the SQL version. It is conceivable that in the real worlds, queries may reference those IDs directly. For example as hardcoded list as part of a GUI.

1.4 Entity Names

As in the SQL version, the actual name of a *song*, *genre*, *album* and *artist* is a property named after the entity type and therefore different per node label. This was done to map the SQL version as closely as possible to the graph

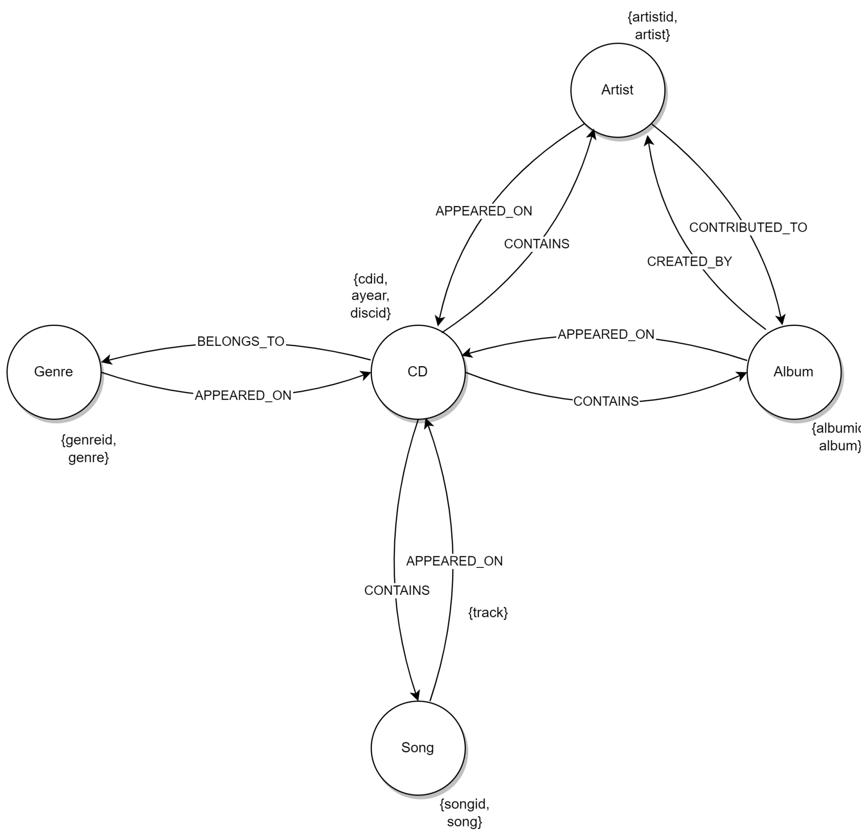

Figure 1: CDDB Graph Schema

version. In real world scenarios it may be beneficial to rename these properties to *`name`* across all nodes. This would allow for simpler queries in cases where nodes of different types are filtered by name. Also, a joint full-text index would be possible.

1.5 Indices

To speed up `MERGE` and `MATCH` operations, the IDs for nodes with label *`Artist`*, *`Album`*, *`Genre`*, *`Song`* or *`CD`* are indexed. In addition, a full text index is used for the names of *`Artists`*, *`Albums`* and *`Songs`* to allow for imprecise searches (fuzzy search).

1.6 Importing Songs

Songs with a trailing backslashcoused problems because the closing quote is escaped. To mitigate this, an extra whitespace character is added to those song titles. Effected rows can be found with the following sql statement:

```
SELECT *
from cddb.songs s
where song like '%\\'
  and song not like '%\\\\'
```

Statement to update the rows:

```
UPDATE cddb.songs
SET song = song || '␣'
WHERE song LIKE '%\\'
  AND song NOT LIKE '%\\\\'
```

2 Cypher Queries

See Jupyter notebook `02_Cypher_Queries.ipynb`.

3 SQL to Cypher

See Jupyter notebook `03_SQL_to_Cypher.ipynb`.

4 Searching and Ranking

See Jupyter notebook `04_Searching_and_Ranking.ipynb`.

4.1 Search Implementation Details

The search accepts a user supplied query string and returns an ordered list of the most relevant CDs. Three features are considered to rank CDs by relevance: Text similarity with the query string, user-preferred CDs and user-preferred genres. Since the user's preference is known prior to searching, the corresponding preference scores can be computed in advance. This leads to a two stage process:

4.1.1 Stage 1: Precompute User Preference

Preferred CDs and genres are determined based on content previously liked by the user. To find preferred CDs, a subgraph of artists, albums and songs liked by the user and associated CDs is extracted. Subsequently, a centrality score is calculated for each node in both. The basic idea is that the most relevant CDs will have a high centrality score. To determine the preferred genres, we count how often genres are associated with the user's liked nodes. The basic idea is that the user prefers content of a few genres and therefore likes many albums, artists and songs belonging to those genres.

4.1.2 Stage 2: Search based on User Input

The most significant feature is the text similarity to the search query provided by the user. By using "Full-Text Indices" of Neo4J, the text similarity between artist name, album title and song title is calculated. The similarity score of all nodes associated with a CD are aggregated and used as a "Content Match Score".

$$S_{Combined} = S_{Content} + (S_{CD} * S_{Content}) + (S_{Genre} * S_{Content}) \tag{1}$$

The "Content Match Score" is then combined with the previously computed user specific "CD Preference Score" and "Genre Preference Score". The ten CDs with the highest combined score are presented to the user.

4.2 Example Search Queries

The following example demonstrates how user 1 is searching for CDs related to "Jimmi Hendrix." See Jupyter notebook `05_Search_Demonstration.ipynb` for more detailed examples.

```
# User specified search query
search_input = "Jimi Hendrix purple haze are you experienced"
user_id = 1
search_mask = "all"

# Search for relevant CDs
searchFor(driver, database_name = "cddb",
          user_id, search_input, search_mask)
```

		Score				CD Details	
	ID	Content	CD	Genre	Comb.	Genre	Artists
0	677	31.53	0.5663	0.6632	**70.31**	[rock]	[jimi hendrix]
1	35734	31.53	0.5541	0.6632	**69.93**	[rock]	[jimi hendrix]
2	136907	31.53	0.5541	0.6632	**69.93**	[rock]	[jimi hendrix]
3	46232	26.72	0.6927	0.6632	**62.96**	[rock]	[jimi hendrix]
4	162186	26.72	0.6581	0.6632	**62.03**	[rock]	[jimi hendrix]
5	7923	56.69	0.0000	0.0714	**60.73**	[blues]	[signature licks]
6	1936	24.34	0.6927	0.6632	**57.35**	[rock]	[jimi hendrix]
7	33321	28.27	0.2123	0.6632	**53.03**	[rock]	[jimi hendrix]
8	2023	26.72	0.6927	0.0714	**47.14**	[blues]	[jimi hendrix]
9	17266	20.76	0.3463	0.6632	**41.73**	[rock]	[jimi hendrix]

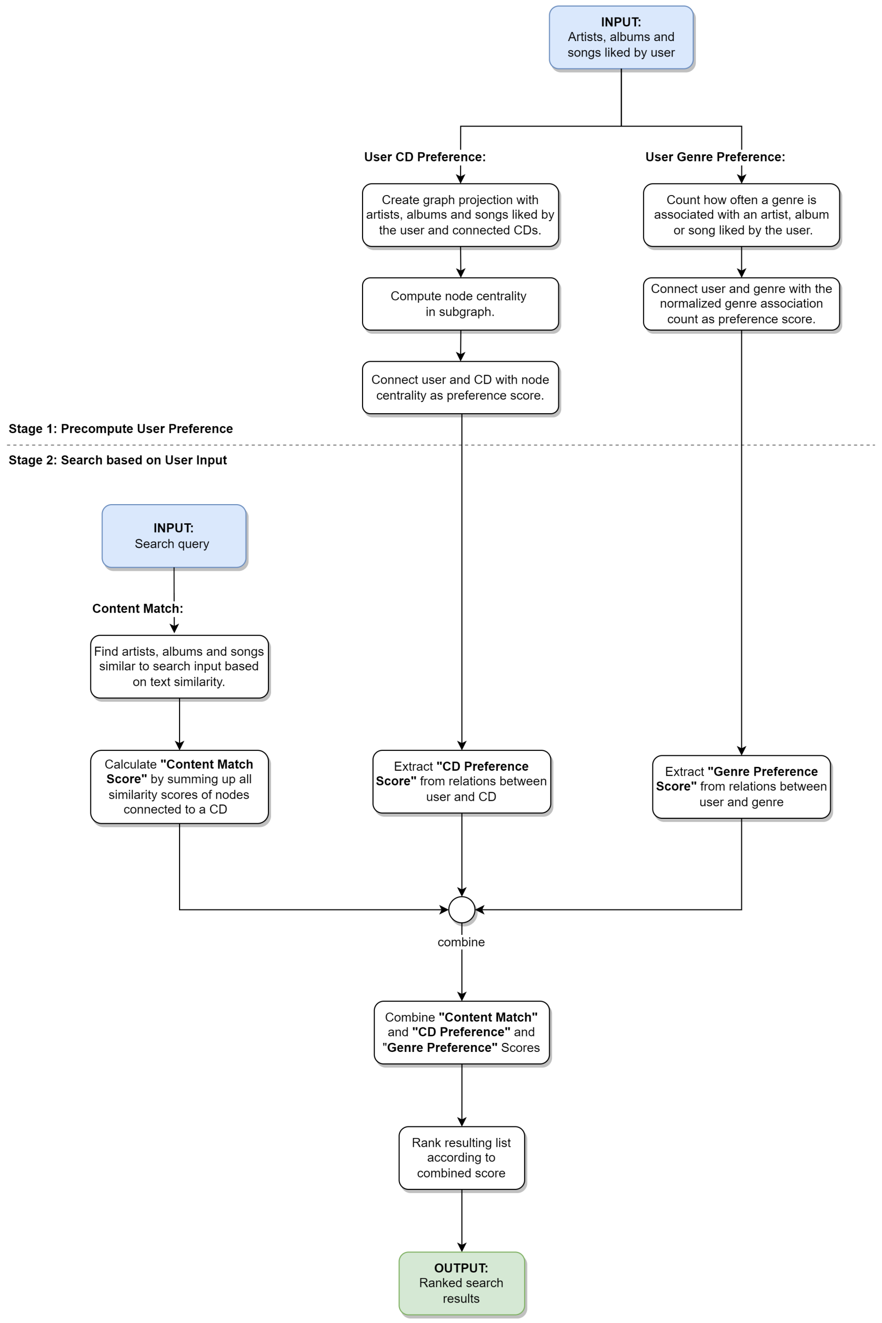
INPUT: Artists, albums and songs liked by user
User CD Preference:
User Genre Preference:
Create graph projection with artists, albums and songs liked by the user and connected CDs.
Count how often a genre is associated with an artist, album or song liked by the user.
Compute node centrality in subgraph.
Connect user and genre with the normalized genre association count as preference score.
Connect user and CD with node centrality as preference score.
Stage 1: Precompute User Preference
Stage 2: Search based on User Input
INPUT: Search query
Content Match:
Find artists, albums and songs similar to search input based on text similarity.
Calculate "Content Match Score" by summing up all similarity scores of nodes connected to a CD
Extract "CD Preference Score" from relations between user and CD
Extract "Genre Preference Score" from relations between user and genre
combine
Combine "Content Match" and "CD Preference" and "Genre Preference" Scores
Rank resulting list according to combined score
OUTPUT: Ranked search results

Figure 2: Two stage process of calculating a CD relevance score by using text similarity and user preference as features.