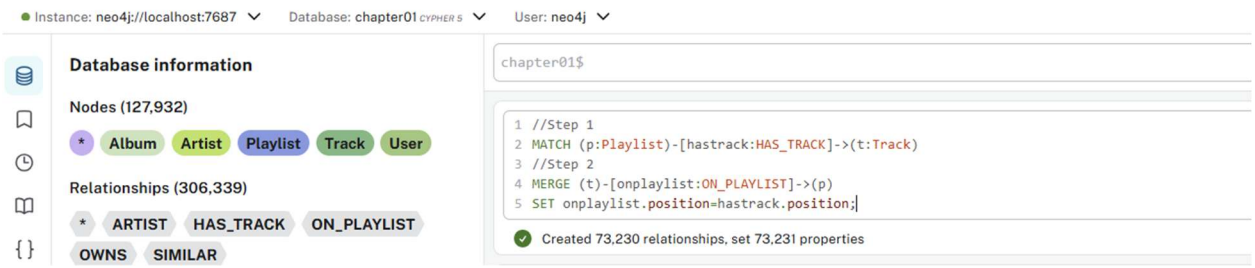
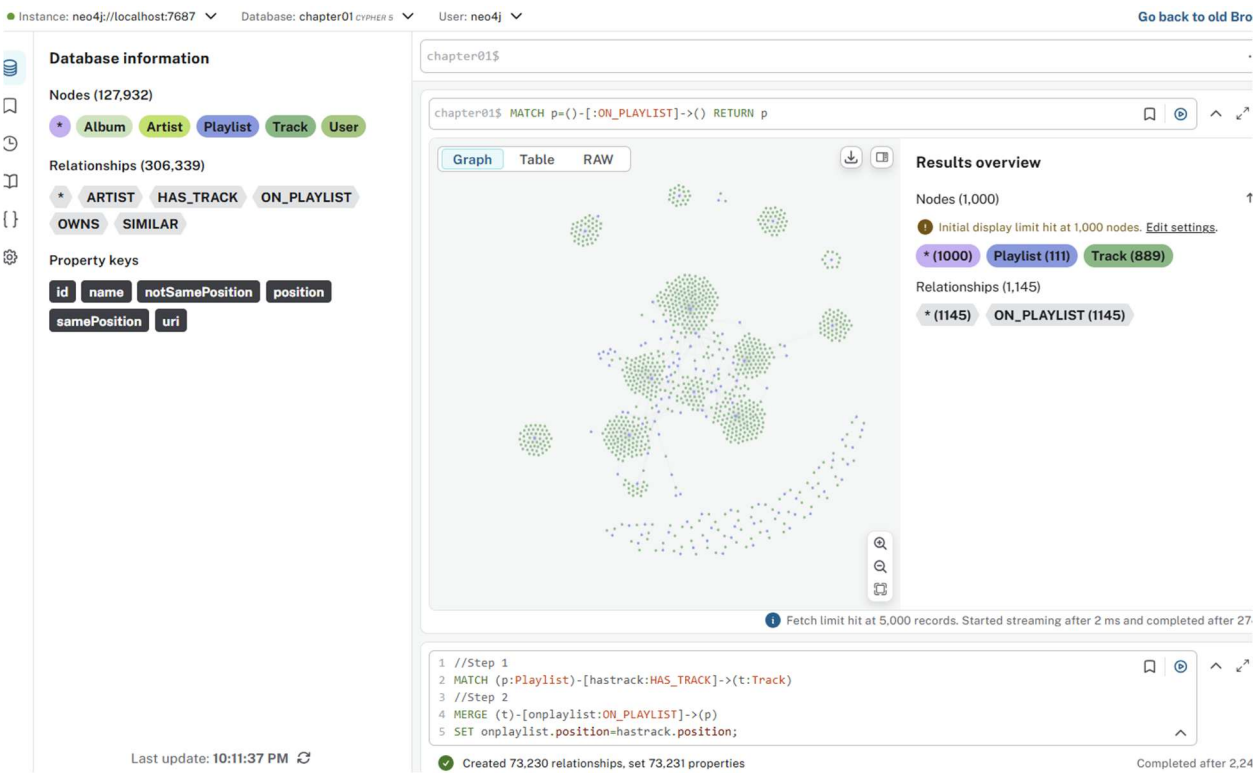
001-rel-type-refactor-1

chapter01$

```
//Step 1
MATCH (p:Playlist)-[hastrack:HAS_TRACK]->(t:Track)
//Step 2
MERGE (t)-[onplaylist:ON_PLAYLIST]->(p)
SET onplaylist.position=hastrack.position;
```





There are important case sensitivity rules in Cypher:

1. **Keywords and functions**: Case-insensitive (MATCH, RETURN, collect, COLLECT)
2. **Variables, labels, and relationship types**: Case-sensitive

002-recommendation-1

chapter01$

```
MATCH (popularTrack:Track)-[:ON_PLAYLIST]-(:Playlist)
WITH popularTrack, count(*) as playlistCount
WHERE playlistCount > 5
RETURN popularTrack.name, playlistCount
```

chapter01$

```
MATCH (p)<-[r:ON_PLAYLIST]-(t)
WHERE r.position  = COUNT { (p)<-[:ON_PLAYLIST]-(t)}
RETURN p.name, t.name,  r.position
limit 10
```

Find tracks (t) on playlists (p) where the track's position in the playlist equals the total number of tracks in that playlist. In other words, find tracks that are in the **last position** of their playlists

chapter01$

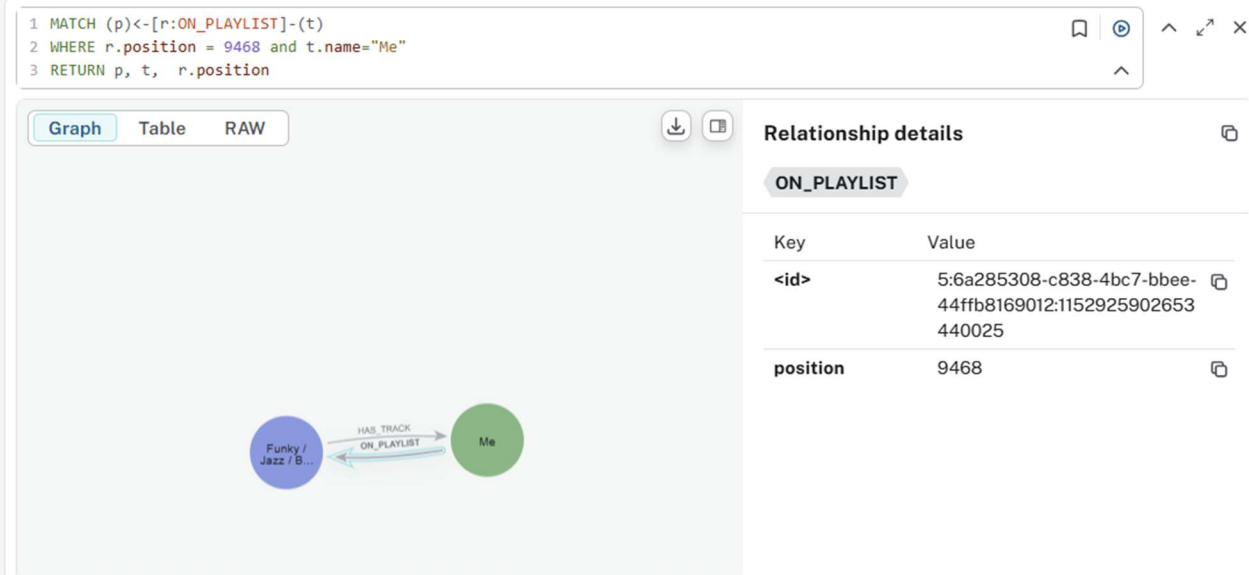
```
MATCH (p)<-[r:ON_PLAYLIST]-(t)
WHERE r.position = COUNT { (p)<-[:ON_PLAYLIST]-()}
RETURN p.name, t.name,  r.position
```

This is a **count subquery** that counts all :ON_PLAYLIST relationships to node p. However, there are two problems:

1. **Subqueries in** WHERE **clauses** need to be boolean expressions, not numeric comparisons
2. **Aggregation context** - you're mixing per-row matching with aggregated counts

chapter01$

```
MATCH (p)<-[r:ON_PLAYLIST]-(t)
WHERE r.position = COUNT {
 MATCH (p)<-[r2:ON_PLAYLIST]-(t2)
 RETURN r2
}
RETURN p.name, t.name,  r.position
```



chapter01$

```
MATCH (p)<-[:ON_PLAYLIST]-(t)
WITH p, COUNT(t) AS total_tracks
MATCH (p)<-[r:ON_PLAYLIST]-(t)
WHERE r.position = total_tracks
RETURN p.name, t.name, r.position
```

chapter01$

```
MATCH (p)<-[r:ON_PLAYLIST]-(t)
WHERE r.position = size([(p)<-[:ON_PLAYLIST]-(other_t) | other_t])
RETURN p.name, t.name, r.position
```

Query Overview

The goal is to recommend tracks that:

1. Appear in playlists similar to "all that jazz"
2. Aren't already in "all that jazz"
3. Aren't overly popular tracks
4. Are weighted by how frequently they appear in similar contexts

chapter01$

```
//Find popular tracks
MATCH (popularTrack:Track)-[:ON_PLAYLIST]-(:Playlist)
WITH popularTrack, count(*) as playlistCount
ORDER BY playlistCount DESC
LIMIT 10
WITH collect(elementId(popularTrack)) as popularTracks
//Finds the 10 tracks that appear in the most playlists
//Stores their IDs to exclude them from recommendations (avoid recommending overplayed tracks)
// For a given Playlist
```

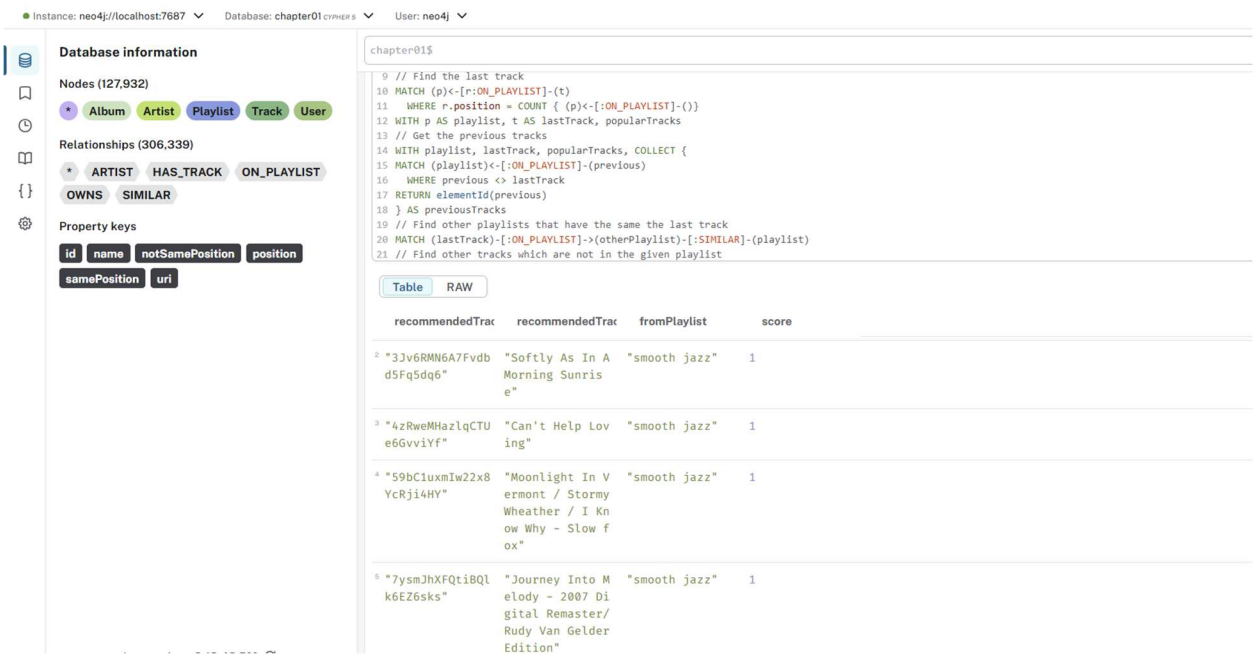
```
MATCH (p:Playlist) WHERE p.name = "all that jazz"
// Find the last track
MATCH (p)<-[r:ON_PLAYLIST]-(t)
WHERE r.position = COUNT { (p)<-[:ON_PLAYLIST]-()}
```
//Finds the playlist named "all that jazz"
//Finds the last track in that playlist (where position = total track count)
//Note: This has the same syntax issue mentioned earlier - should use a subquery

```
 WITH p AS playlist, t AS lastTrack, popularTracks
// Get the previous tracks
WITH playlist, lastTrack, popularTracks, COLLECT {
MATCH (playlist)<-[:ON_PLAYLIST]-(previous)
  WHERE previous <> lastTrack
RETURN elementId(previous)
} AS previousTracks
```
//Collects IDs of all tracks in "all that jazz" EXCEPT the last track
//This creates an exclusion list of tracks already in the playlist

```
// Find other playlists that have the same the last track
MATCH (lastTrack)-[:ON_PLAYLIST]->(otherPlaylist)-[:SIMILAR]-(playlist)
```
- //Finds other playlists that:
  - Contain the same last track as "all that jazz"
  - Are marked as SIMILAR to "all that jazz"
- //This finds contextually relevant playlists

```
// Find other tracks which are not in the given playlist
MATCH (otherPlaylist)<-[:ON_PLAYLIST]-(recommendation)
WHERE NOT elementId(recommendation) IN previousTracks
AND NOT elementId(recommendation) IN popularTracks
```
//Gets tracks from these similar playlists
//Filters out tracks that are:
- Already in "all that jazz" (using previousTracks)
- Overly popular (using popularTracks)

```
// Score them by how frequently they appear
RETURN  recommendation.id as recommendedTrackId, recommendation.name AS
recommendedTrack, otherPlaylist.name AS fromPlaylist, count(*) AS score
ORDER BY score DESC
LIMIT 5
```
//Scores recommendations by **frequency** - how many similar playlists contain each track
//Returns top 5 recommendations with highest scores



The Recommendation Logic

This query uses **collaborative filtering**:

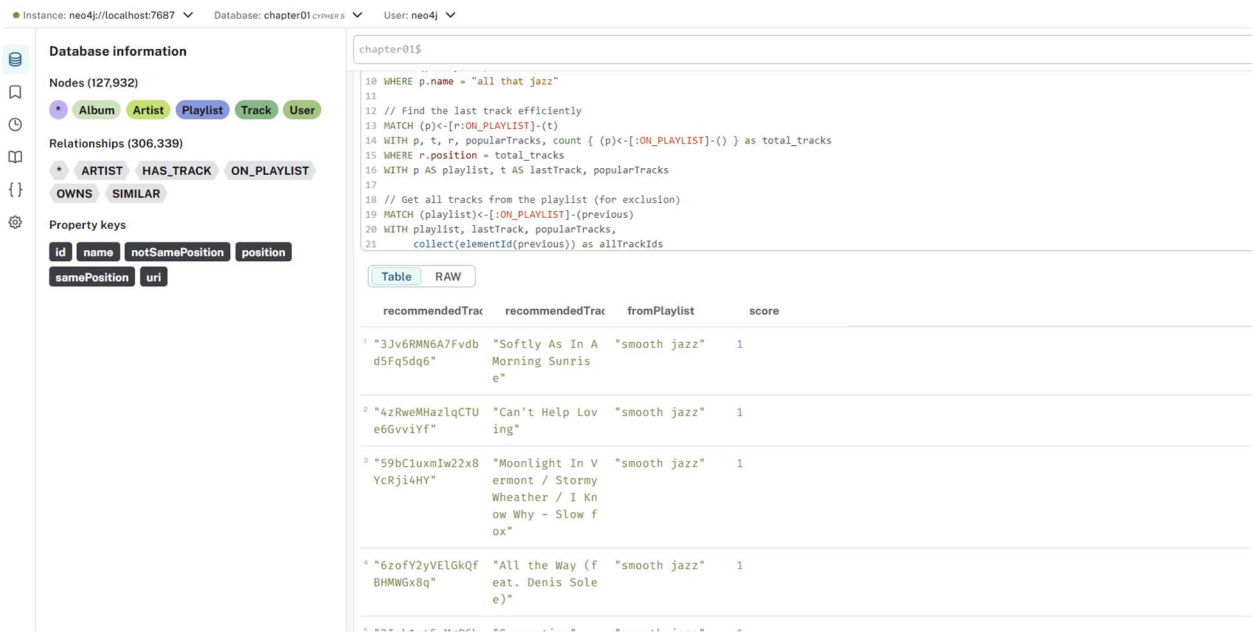
- "Playlists that are similar to yours and share your last track probably have other tracks you'd like"
- It avoids recommending tracks you already have or that are too mainstream
- It prioritizes tracks that appear frequently across multiple similar playlists

Fixed Querry:

chapter01$

```
// Find popular tracks to exclude
MATCH (popularTrack:Track)-[:ON_PLAYLIST]-(:Playlist)
WITH popularTrack, count(*) as playlistCount
ORDER BY playlistCount DESC
LIMIT 10
WITH collect(elementId(popularTrack)) as popularTracks

// For the target playlist
MATCH (p:Playlist)
WHERE p.name = "all that jazz"

// Find the last track efficiently
MATCH (p)<-[r:ON_PLAYLIST]-(t)
```

```
WITH p, t, r, popularTracks, count { (p)<-[:ON_PLAYLIST]-() } as total_tracks
WHERE r.position = total_tracks
WITH p AS playlist, t AS lastTrack, popularTracks

// Get all tracks from the playlist (for exclusion)
MATCH (playlist)<-[:ON_PLAYLIST]-(previous)
WITH playlist, lastTrack, popularTracks,
    collect(elementId(previous)) as allTrackIds

// Find similar playlists that share the last track
MATCH (lastTrack)-[:ON_PLAYLIST]->(otherPlaylist)-[:SIMILAR]-(playlist)
WHERE otherPlaylist <> playlist

// Find recommendation candidates
MATCH (otherPlaylist)<-[:ON_PLAYLIST]-(recommendation)
WHERE NOT elementId(recommendation) IN allTrackIds
  AND NOT elementId(recommendation) IN popularTracks

// Score by frequency across similar playlists
RETURN recommendation.id as recommendedTrackId,
     recommendation.name AS recommendedTrack,
     otherPlaylist.name AS fromPlaylist,
     count(*) AS score
ORDER BY score DESC
LIMIT 5
```

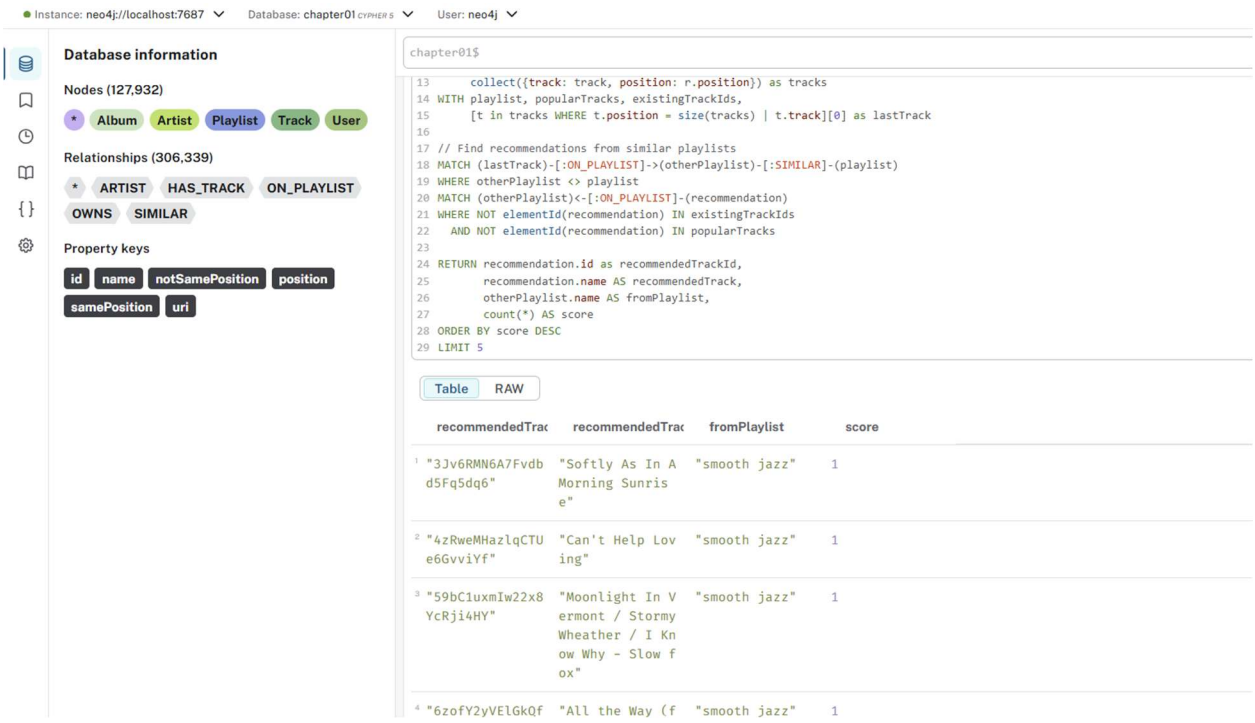


Optimized Querry:

chapter01$

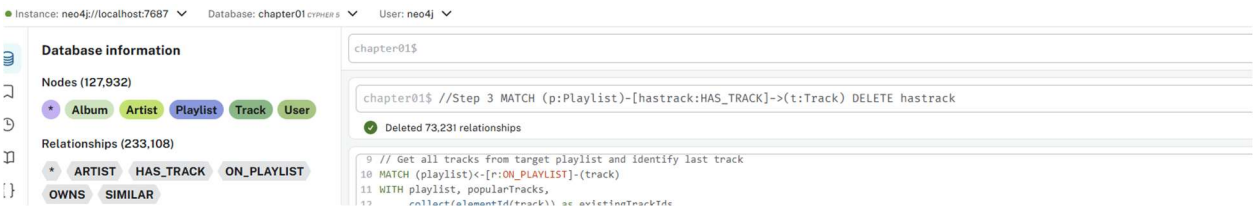
```
// Find popular tracks and target playlist in parallel
MATCH (popularTrack:Track)-[:ON_PLAYLIST]-(:Playlist)
WITH popularTrack, count(*) as playlistCount
ORDER BY playlistCount DESC
LIMIT 10
WITH collect(elementId(popularTrack)) as popularTracks
MATCH (playlist:Playlist {name: "all that jazz"})

// Get all tracks from target playlist and identify last track
MATCH (playlist)<-[r:ON_PLAYLIST]-(track)
WITH playlist, popularTracks,
   collect(elementId(track)) as existingTrackIds,
   collect({track: track, position: r.position}) as tracks
WITH playlist, popularTracks, existingTrackIds,
   [t in tracks WHERE t.position = size(tracks) | t.track][0] as lastTrack

// Find recommendations from similar playlists
MATCH (lastTrack)-[:ON_PLAYLIST]->(otherPlaylist)-[:SIMILAR]-(playlist)
WHERE otherPlaylist <> playlist
```

```
MATCH (otherPlaylist)<-[:ON_PLAYLIST]-(recommendation)
WHERE NOT elementId(recommendation) IN existingTrackIds
 AND NOT elementId(recommendation) IN popularTracks

RETURN recommendation.id as recommendedTrackId,
    recommendation.name AS recommendedTrack,
    otherPlaylist.name AS fromPlaylist,
    count(*) AS score
ORDER BY score DESC
LIMIT 5
```



003-rel-type-refactor-2

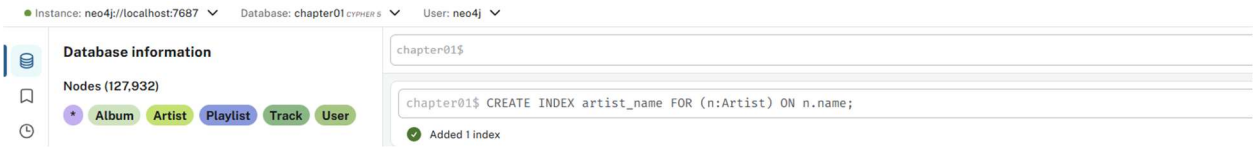
```
chapter01$
    MATCH (p:Playlist)-[hastrack:HAS_TRACK]->(t:Track)
    DELETE hastrack
```



004-artistNameIndex

chapter01$

```
CREATE INDEX artist_name FOR (n:Artist) ON n.name;
```



005-loadGenres

chapter01$

```
LOAD CSV WITH HEADERS FROM "file:///genres.csv" AS row
WITH row
WITH row.Artist as artist, split(row.Genre,"/") AS genreList
UNWIND genreList AS genre
WITH artist, collect(trim(toLower(genre))) AS genres
MATCH (a:Artist {name:artist})
SET a.genres=genres
```

chapter01$

```
MATCH (a)
where a.genres is NOT NULL
RETURN  a.name , a.genres
```

```
chapter01$
MATCH (a)<-[:ARTIST]-(t)
WHERE a.genres is NOT NULL
RETURN  a.name , a.genres , t.name
LIMIT 100
```

```
chapter01$
MATCH (n:Artist {name:"David Bowie"
 }) RETURN n;
```

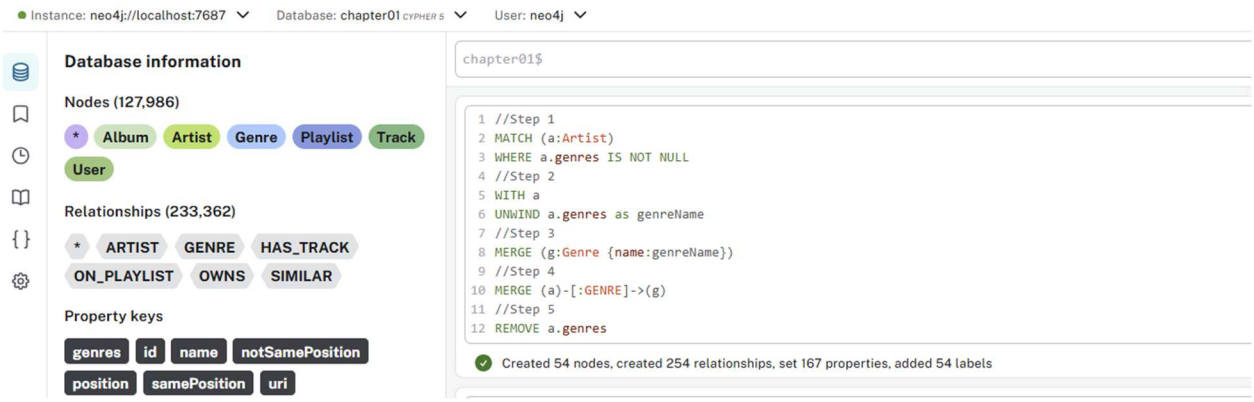
006-genreConstraint

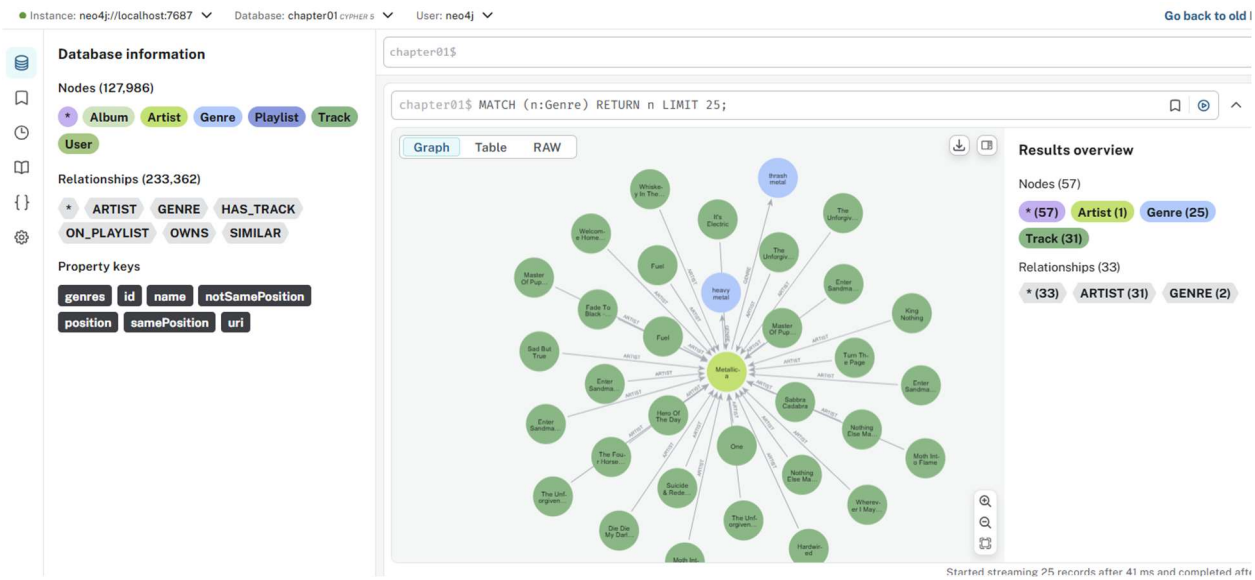
```
chapter01$
CREATE CONSTRAINT genre_name
FOR (genre:Genre) REQUIRE genre.name IS UNIQUE
```

007-refactor-genre

```
chapter01$
//Step 1
MATCH (a:Artist)
WHERE a.genres IS NOT NULL
//Step 2
WITH a
UNWIND a.genres as genreName
//Step 3
MERGE (g:Genre {name:genreName})
//Step 4
```

MERGE (a)-[:GENRE]->(g)
//Step 5
REMOVE a.genres





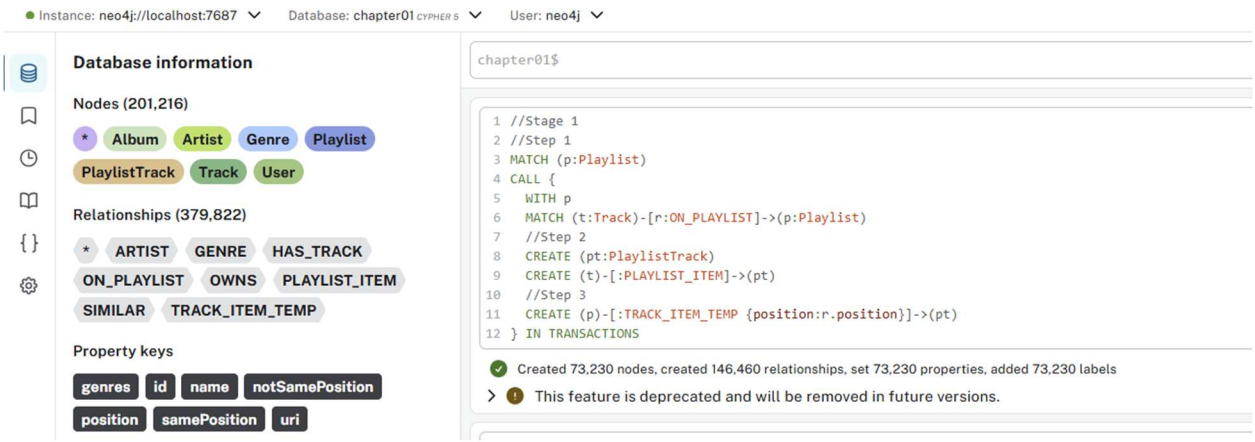
008-refactor-playlist-linked-1

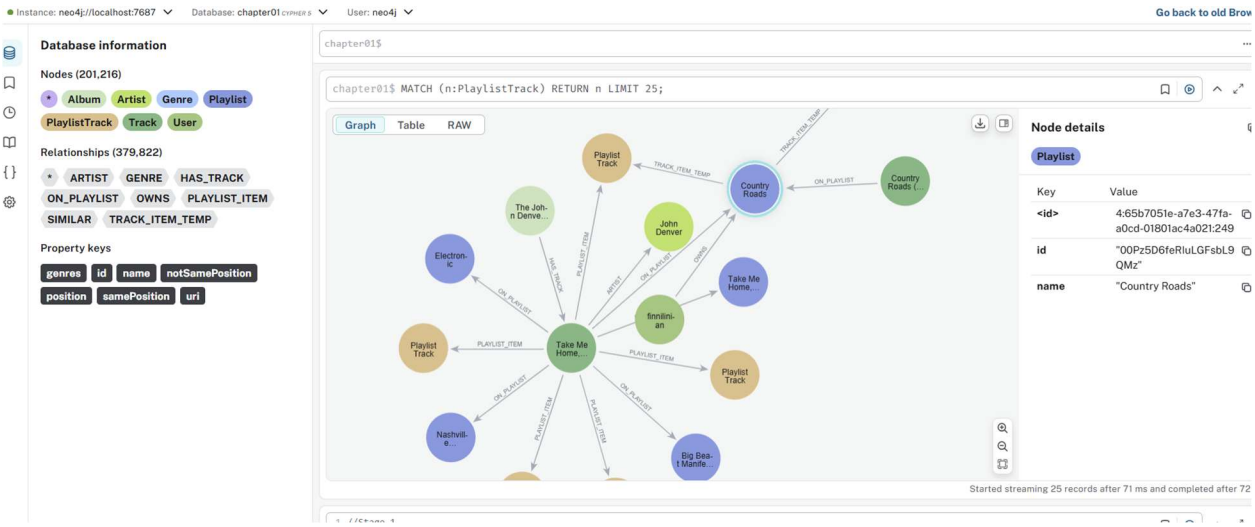
chapter01$

//Stage 1
//Step 1
MATCH (p:Playlist)

//The CALL { } subquery runs for each playlist (p)
//Finds all tracks (t) that have an ON_PLAYLIST relationship to the current playlist
//The relationship r contains the position property

```
CALL {
 WITH p
 MATCH (t:Track)-[r:ON_PLAYLIST]->(p:Playlist)

//Creates a new PlaylistTrack node for each track-playlist combination
//Establishes a relationship from the track to this junction node
//Step 2
 CREATE (pt:PlaylistTrack)
 CREATE (t)-[:PLAYLIST_ITEM]->(pt)

//Creates a relationship from the playlist to the junction node
//Preserves the position data by copying it from the original ON_PLAYLIST relationship
 //Step 3
 CREATE (p)-[:TRACK_ITEM_TEMP {position:r.position}]->(pt)

//Executes the operations in batched transactions
//Prevents memory issues when processing large datasets
//Neo4j automatically manages transaction boundaries
} IN TRANSACTIONS
```





Query Overview

The query is creating a **junction node** pattern to replace a direct relationship between Playlists and Tracks.

Original Structure (before):

(Track)-[:ON_PLAYLIST {position: X}]->(Playlist)

New Structure (after):

(Track)-[:PLAYLIST_ITEM]->(PlaylistTrack)<-[:TRACK_ITEM_TEMP {position: X}]-(Playlist)

Why This Pattern is Used

This transformation is common when you need to:

1. **Add metadata** to playlist-track relationships
2. **Normalize the data model** for many-to-many relationships
3. **Prepare for additional relationships** to the junction node
4. **Migrate to a more flexible schema**

009-refactor-playlist-linked-2

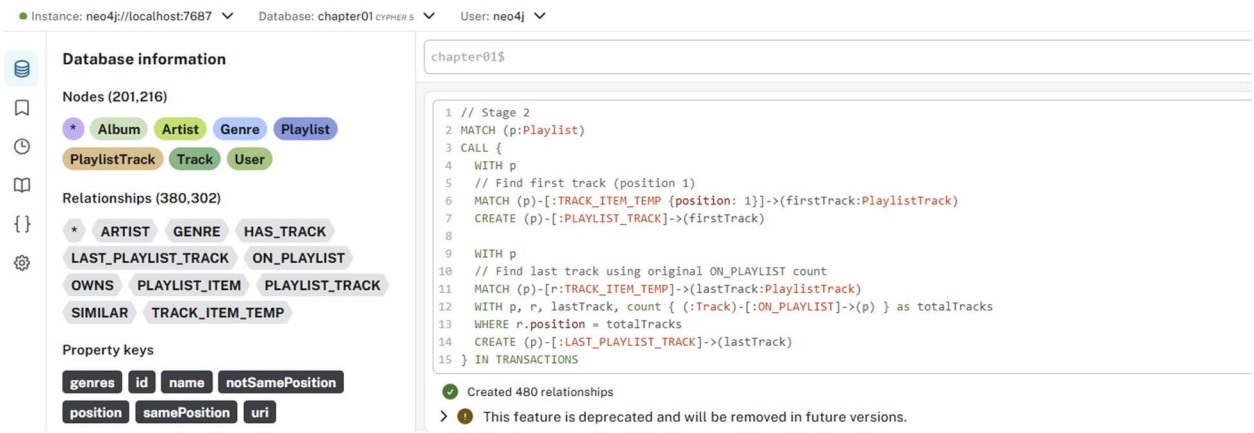
Don't execute original:

```
//Stage 2
MATCH (p:Playlist)
CALL {
 WITH p
```

```
  MATCH (p)-[:TRACK_ITEM_TEMP {position:1}]->(firstTrack:PlaylistTrack)
  CREATE (p)-[:PLAYLIST_TRACK]->(firstTrack)
  WITH p
  MATCH (p)-[r:TRACK_ITEM_TEMP]->(lastTrack:PlaylistTrack)
   WHERE r.position = COUNT {()-[:ON_PLAYLIST]->(p)}
  CREATE (p)-[:LAST_PLAYLIST_TRACK]->(lastTrack)
 } IN TRANSACTION
```

chapter01$

```
 // Stage 2
 MATCH (p:Playlist)
 CALL {
  WITH p
  // Find first track (position 1)
  MATCH (p)-[:TRACK_ITEM_TEMP {position: 1}]->(firstTrack:PlaylistTrack)
  CREATE (p)-[:PLAYLIST_TRACK]->(firstTrack)

  WITH p
  // Find last track using original ON_PLAYLIST count
  MATCH (p)-[r:TRACK_ITEM_TEMP]->(lastTrack:PlaylistTrack)
  WITH p, r, lastTrack, count { (:Track)-[:ON_PLAYLIST]->(p) } as totalTracks
  WHERE r.position = totalTracks
  CREATE (p)-[:LAST_PLAYLIST_TRACK]->(lastTrack)
 } IN TRANSACTIONS
```



If you are not sure that you ran the query twice, you can use below queries to verify that:

```
MATCH (p:Playlist)
WHERE size([(p)-[:PLAYLIST_TRACK]->() | 1]) > 1
  OR size([(p)-[:LAST_PLAYLIST_TRACK]->() | 1]) > 1
RETURN count(p) as playlistsWithDuplicates;
```

Interpret Results:

- *If playlistsWithDuplicates = 0 → Query ran once (or relationships don't exist)*
- *If playlistsWithDuplicates > 0 → Query ran multiple times*

If duplicates exist, see details:

```
MATCH (p:Playlist)
WITH p,
   size([(p)-[:PLAYLIST_TRACK]->() | 1]) as ptCount,
   size([(p)-[:LAST_PLAYLIST_TRACK]->() | 1]) as lptCount
WHERE ptCount > 1 OR lptCount > 1
RETURN p.name as playlistName, ptCount, lptCount
ORDER BY ptCount DESC, lptCount DESC;
```

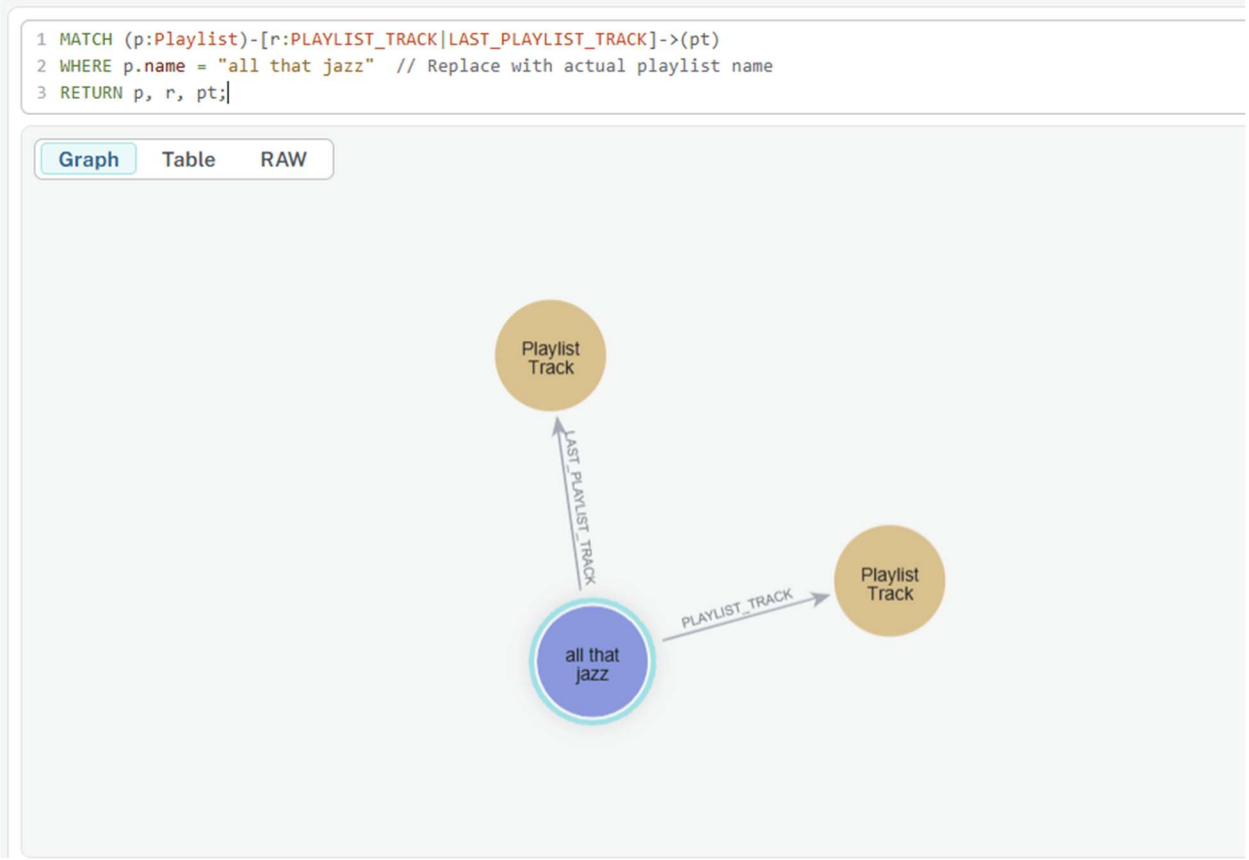
Visual Graph Inspection (Optional):

You can also visually check a specific playlist:

```
// Pick a playlist and view its relationships
MATCH (p:Playlist)-[r:PLAYLIST_TRACK|LAST_PLAYLIST_TRACK]->(pt)
WHERE p.name = "all that jazz"  // Replace with actual playlist name
RETURN p, r, pt;
```

Look for:

- Multiple PLAYLIST_TRACK arrows from the same playlist
- Multiple LAST_PLAYLIST_TRACK arrows from the same playlist



If Duplicates Found

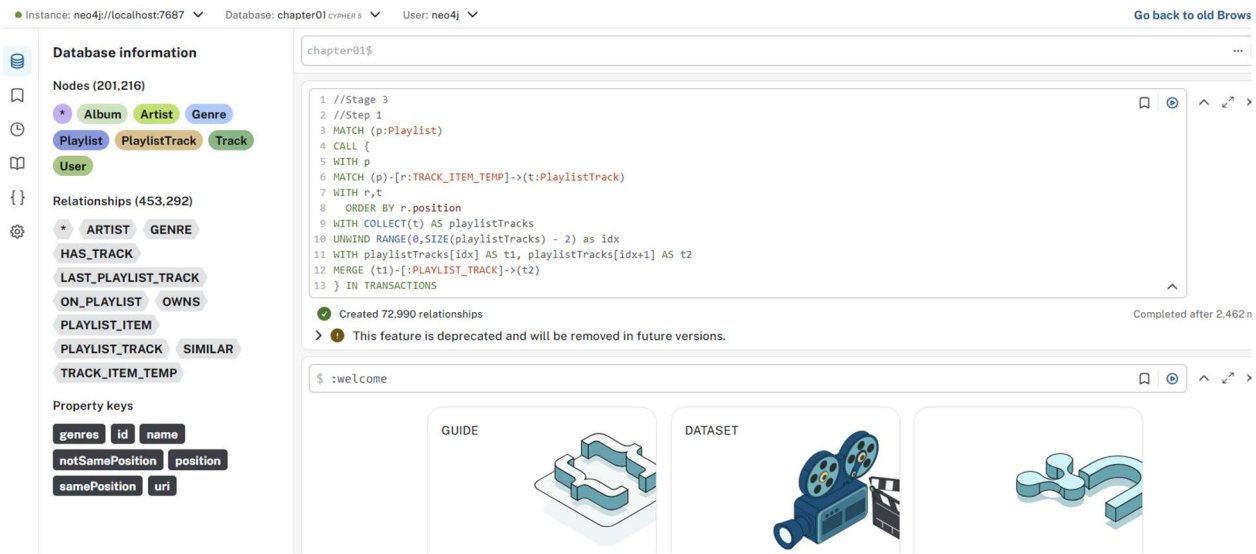
// Remove duplicates in one go
MATCH (p:Playlist)-[r:PLAYLIST_TRACK]->(pt)
WITH p, pt, collect(r) as rels
WHERE size(rels) > 1
UNWIND tail(rels) as extraRel
DELETE extraRel;

MATCH (p:Playlist)-[r:LAST_PLAYLIST_TRACK]->(pt)
WITH p, pt, collect(r) as rels
WHERE size(rels) > 1
UNWIND tail(rels) as extraRel
DELETE extraRel;

010-refactor-playlist-linked-3

chapter01$

```
//Stage 3
//Step 1
MATCH (p:Playlist)
CALL {
WITH p
MATCH (p)-[r:TRACK_ITEM_TEMP]->(t:PlaylistTrack)
WITH r,t
 ORDER BY r.position
WITH COLLECT(t) AS playlistTracks
UNWIND RANGE(0,SIZE(playlistTracks) - 2) as idx
WITH playlistTracks[idx] AS t1, playlistTracks[idx+1] AS t2
MERGE (t1)-[:PLAYLIST_TRACK]->(t2)
} IN TRANSACTIONS
```



Issues with above query:

1. **Missing** OF 10000 **in** IN TRANSACTIONS - This specifies the batch size
2. **No cleanup of temporary relationships** - You're creating new relationships but not removing the old ones
3. **Potential performance issues** with large playlists due to the UNWIND approach

chapter01$

```
// Count how many TRACK_ITEM_TEMP relationships exist
MATCH ()-[r:TRACK_ITEM_TEMP]->()
RETURN COUNT(r) AS tempRelationshipsToDelete;
```

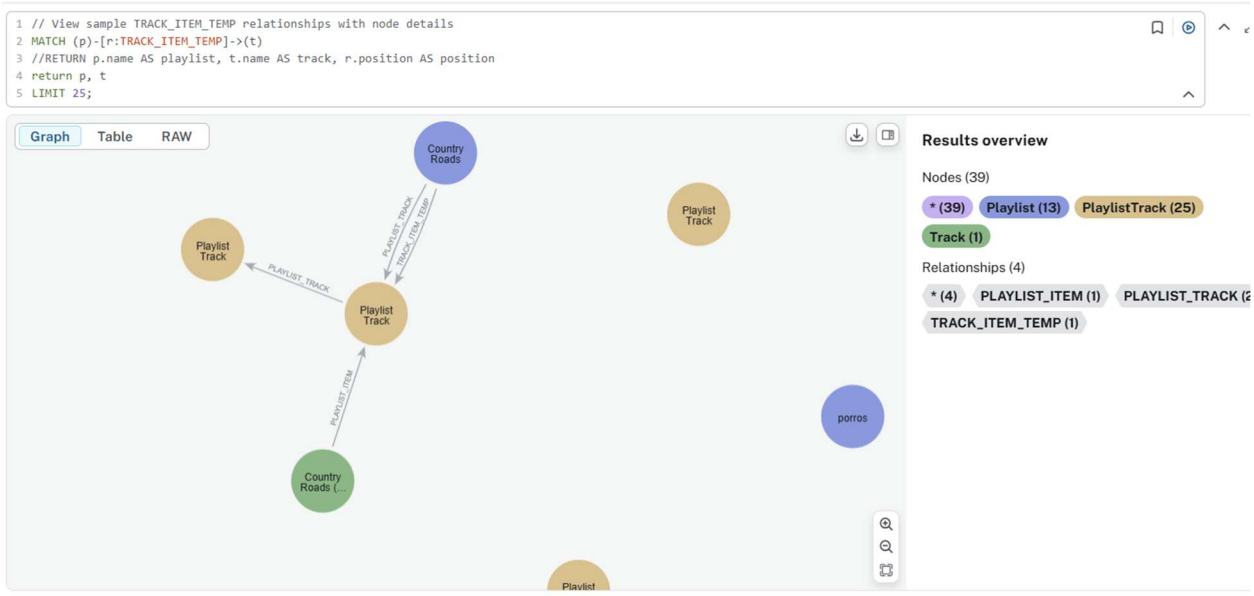
```
1 // Count how many TRACK_ITEM_TEMP relationships exist
2 MATCH ()-[r:TRACK_ITEM_TEMP]->()
3 RETURN COUNT(r) AS tempRelationshipsToDelete;
```

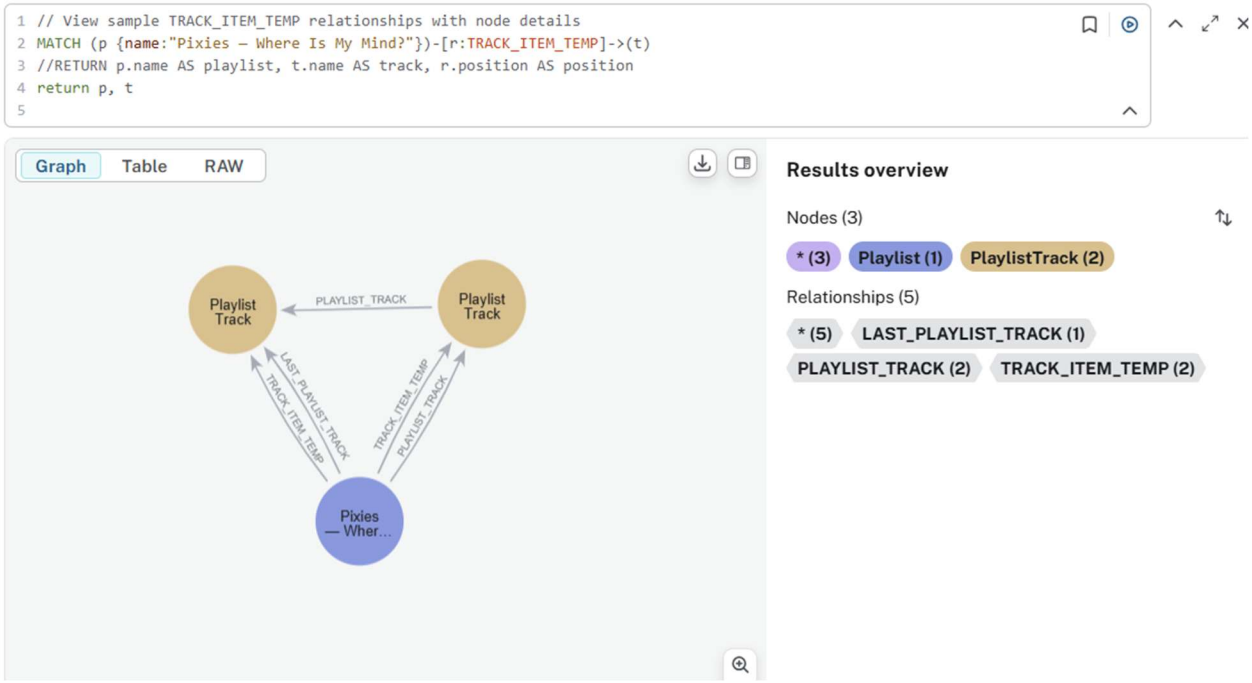
Table RAW

tempRelationships
1 73230

chapter01$

```
// View sample TRACK_ITEM_TEMP relationships with node details
MATCH (p)-[r:TRACK_ITEM_TEMP]->(t)
//RETURN p.name AS playlist, t.name AS track, r.position AS position
return p, t
LIMIT 25;
```





++++++++++

```
// Step 2a - First verify what will be deleted
MATCH (p)-[r:TRACK_ITEM_TEMP]->(t)
RETURN
  p.name AS playlistName,
  COUNT(r) AS trackCount,
  COLLECT(t.name)[0..5] AS sampleTracks
LIMIT 10;

// Step 2b - Then proceed with deletion (if correct)
MATCH ()-[r:TRACK_ITEM_TEMP]->()
DELETE r
RETURN "Deleted " + COUNT(r) + " TRACK_ITEM_TEMP relationships";
```

++++++++++++

```
// Safe approach that shows what was deleted
MATCH ()-[r:TRACK_ITEM_TEMP]->()
WITH COLLECT(r) AS relationshipsToDelete
FOREACH (r IN relationshipsToDelete | DELETE r)
RETURN SIZE(relationshipsToDelete) AS deletedRelationshipCount;
```

++++++++++++++++

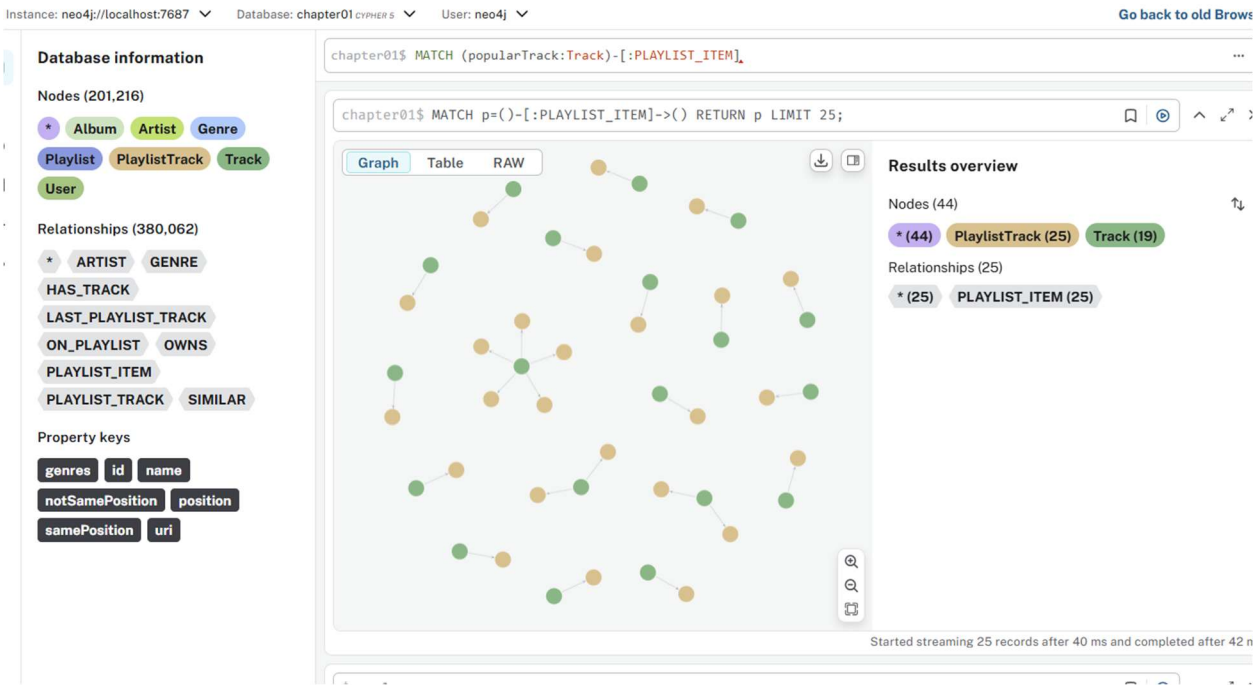
011-refactor-playlist-linked-4

Don’t execute original querry:

```
//Stage 3
//Step 2
MATCH ()-[r:TRACK_ITEM_TEMP]-()
CALL(r) {
  DELETE r
}
IN TRANSACTIONS
```

chapter01$

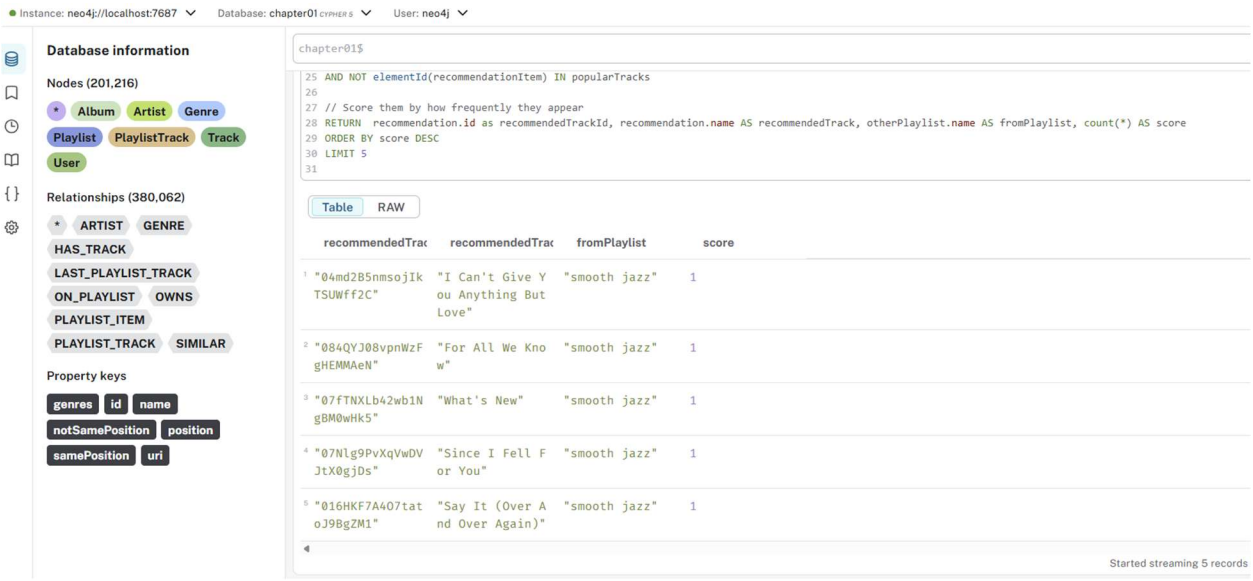
```
MATCH ()-[r:TRACK_ITEM_TEMP]->()
WITH COLLECT(r) AS relationshipsToDelete
FOREACH (r IN relationshipsToDelete | DELETE r)
RETURN SIZE(relationshipsToDelete) AS deletedRelationshipCount;
```



012-recommendation-2

```
chapter01$
        //Find popular tracks
        MATCH (popularTrack:Track)-[:PLAYLIST_ITEM]->()
        WITH popularTrack, count(*) as playlistCount
        ORDER BY playlistCount DESC
        LIMIT 10
        WITH collect(elementId(popularTrack)) as popularTracks
        // For a given Playlist
        MATCH (playlist:Playlist) WHERE playlist.name = "all that jazz"
        // Find the last track
        MATCH (playlist)-[:LAST_PLAYLIST_TRACK]->(lastTrackItem)

        // Get the previous tracks
        WITH playlist, lastTrackItem, popularTracks, COLLECT {
        MATCH (playlist) (()-[:PLAYLIST_TRACK]->()) {1,100} (previousTrackItem)
        WHERE previousTrackItem <> lastTrackItem
        RETURN elementId(previousTrackItem)
        } AS previousTrackItems

        // Find other playlists that have the same the last track
```

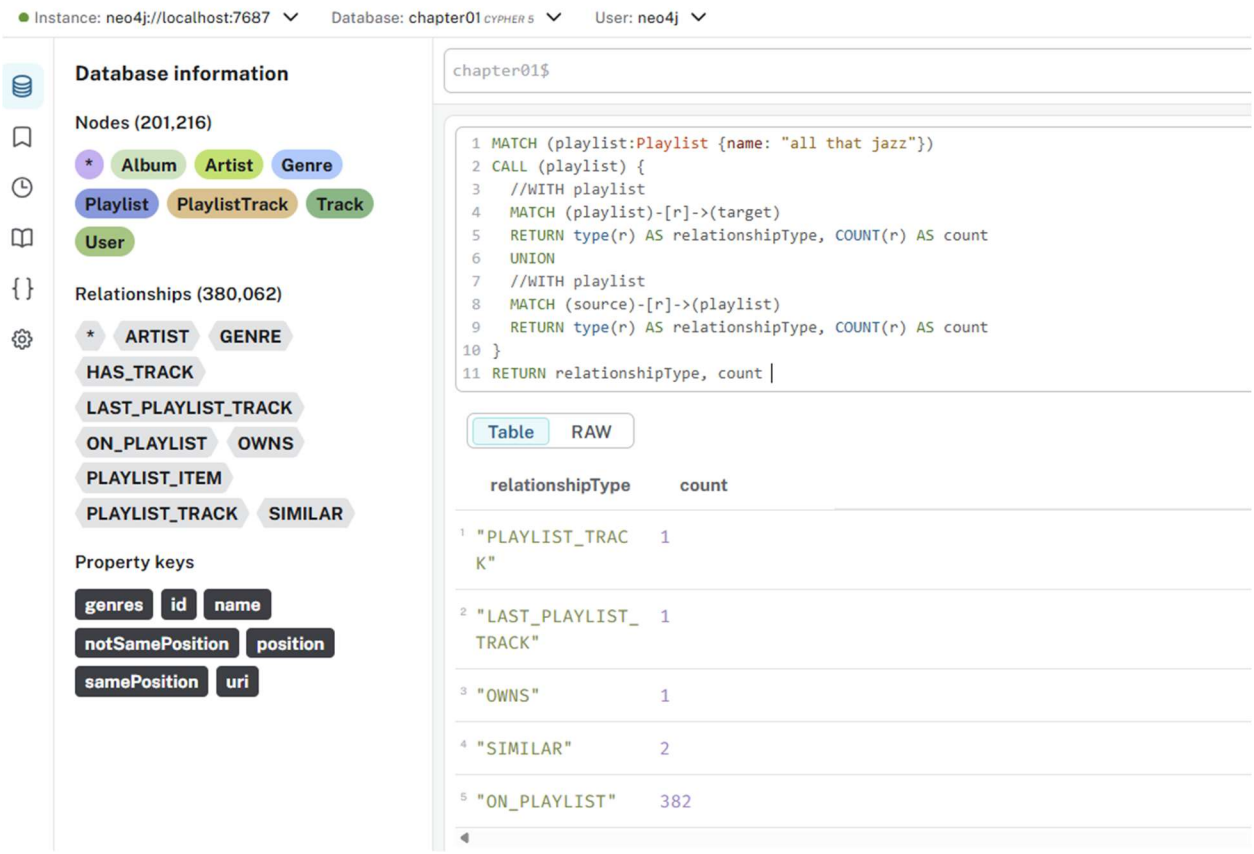
```
MATCH (playlist)-[:SIMILAR]-(otherPlaylist:Playlist)-[:LAST_PLAYLIST_TRACK]-
>(otherLastTrack)<-[:PLAYLIST_ITEM]-(:Track)-[:PLAYLIST_ITEM]->()<-
[:LAST_PLAYLIST_TRACK]-(playlist)

// Find other tracks which are not in the given playlist
MATCH (otherPlaylist) (()-[:PLAYLIST_TRACK]->()) {1,100} (recommendationItem)<-
[:PLAYLIST_ITEM]-(recommendation)
WHERE NOT elementId(recommendationItem) IN previousTrackItems
AND NOT elementId(recommendationItem) IN popularTracks

// Score them by how frequently they appear
RETURN  recommendation.id as recommendedTrackId, recommendation.name AS
recommendedTrack, otherPlaylist.name AS fromPlaylist, count(*) AS score
ORDER BY score DESC
LIMIT 5
```



Let's debug if relationships are missing:

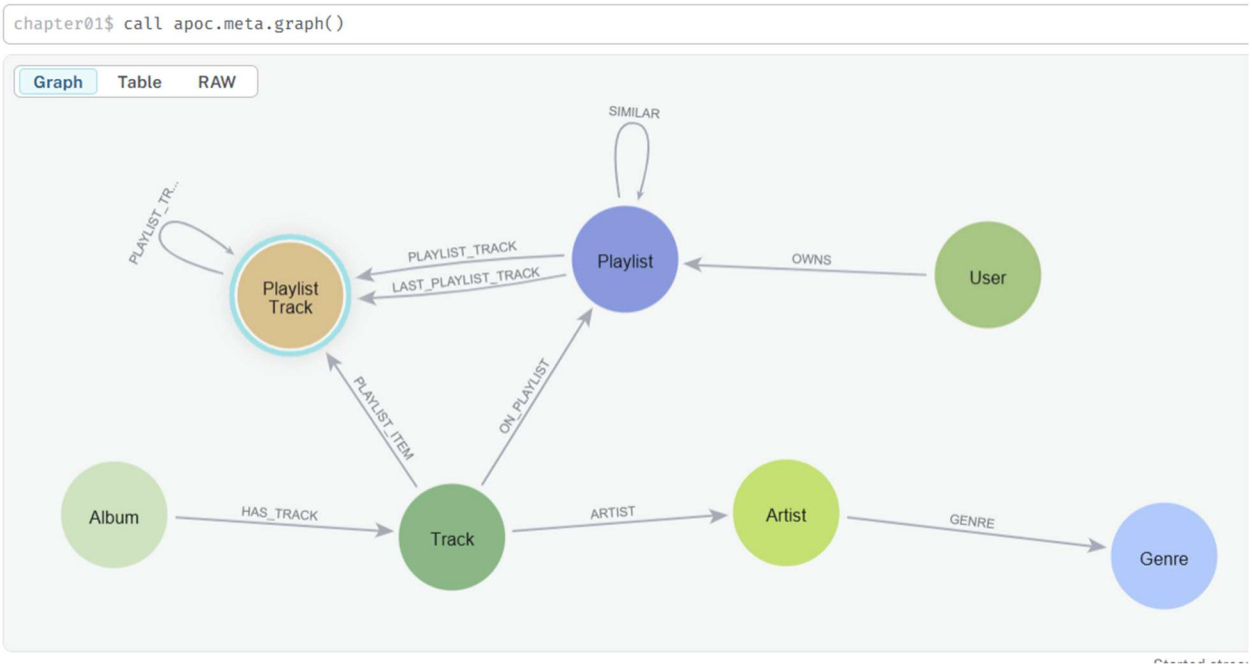
```
chapter01$
        MATCH (playlist:Playlist {name: "all that jazz"})
        CALL (playlist) {
         //WITH playlist
         MATCH (playlist)-[r]->(target)
         RETURN type(r) AS relationshipType, COUNT(r) AS count
         UNION
```

```
  //WITH playlist
  MATCH (source)-[r]->(playlist)
  RETURN type(r) AS relationshipType, COUNT(r) AS count
}
RETURN relationshipType, count
```



Let's see the structure of the Graph: There are a couple of ways to do this depending on whether you have APOC installed or not.

```
CALL apoc.meta.graph()
```



CALL apoc.meta.schema()

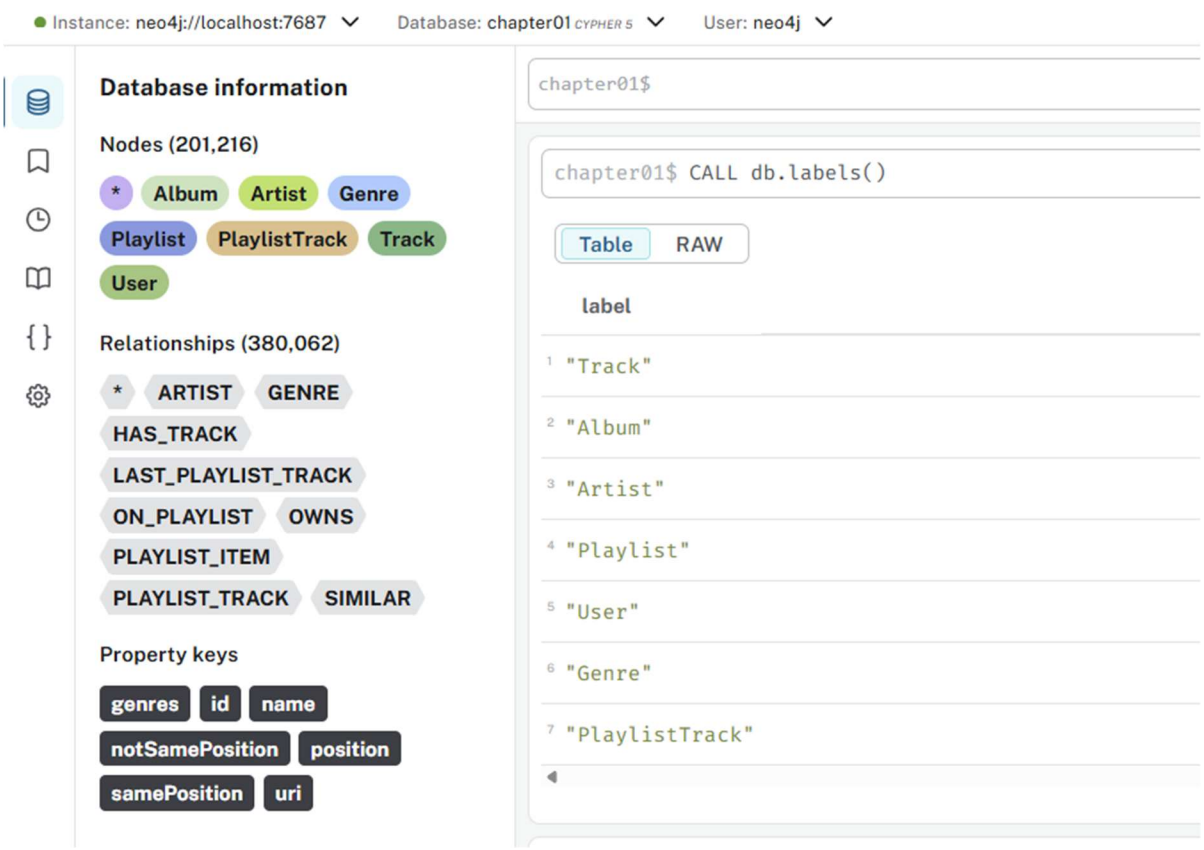
Returns a map with all **labels, relationships, and property types**. Great for structural analysis (not visual).

Without APOC – Using Built-in Neo4j Procedures

If APOC is **not installed**, you can still get basic structure info.

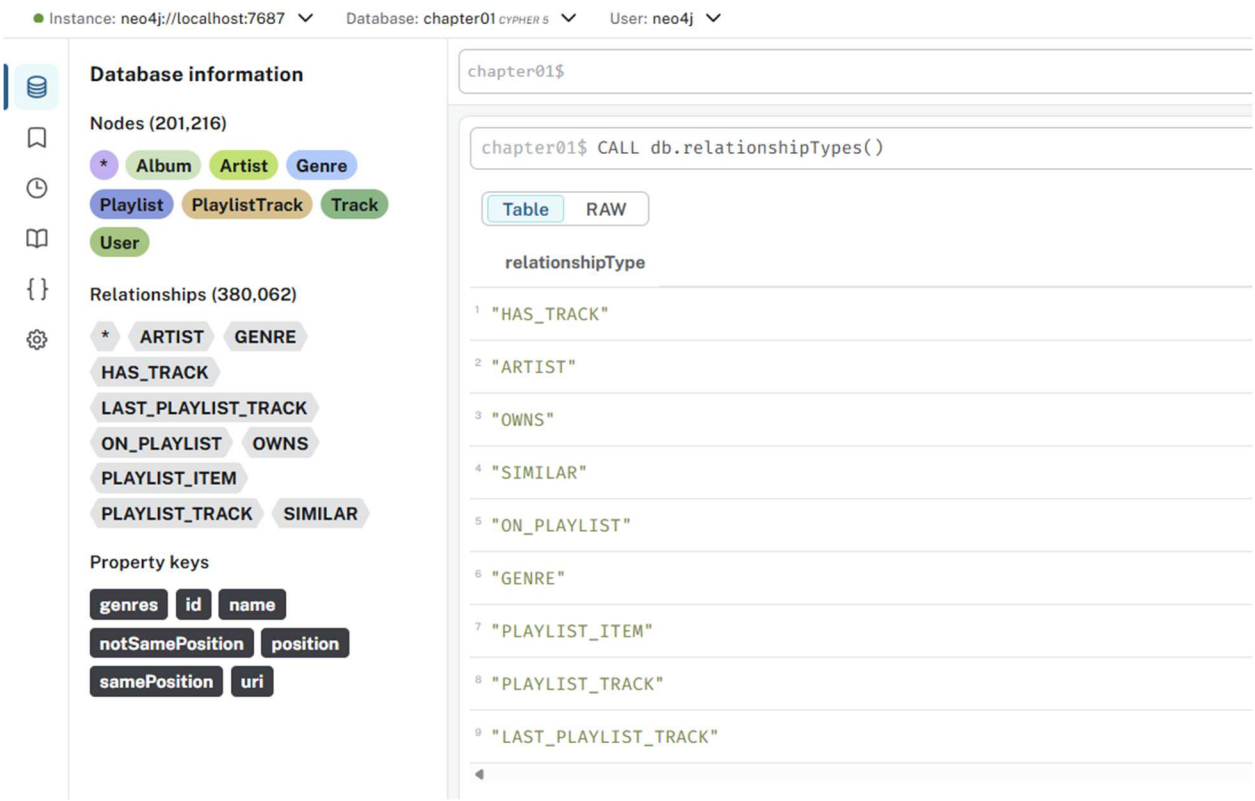
Show all node labels:

CALL db.labels()



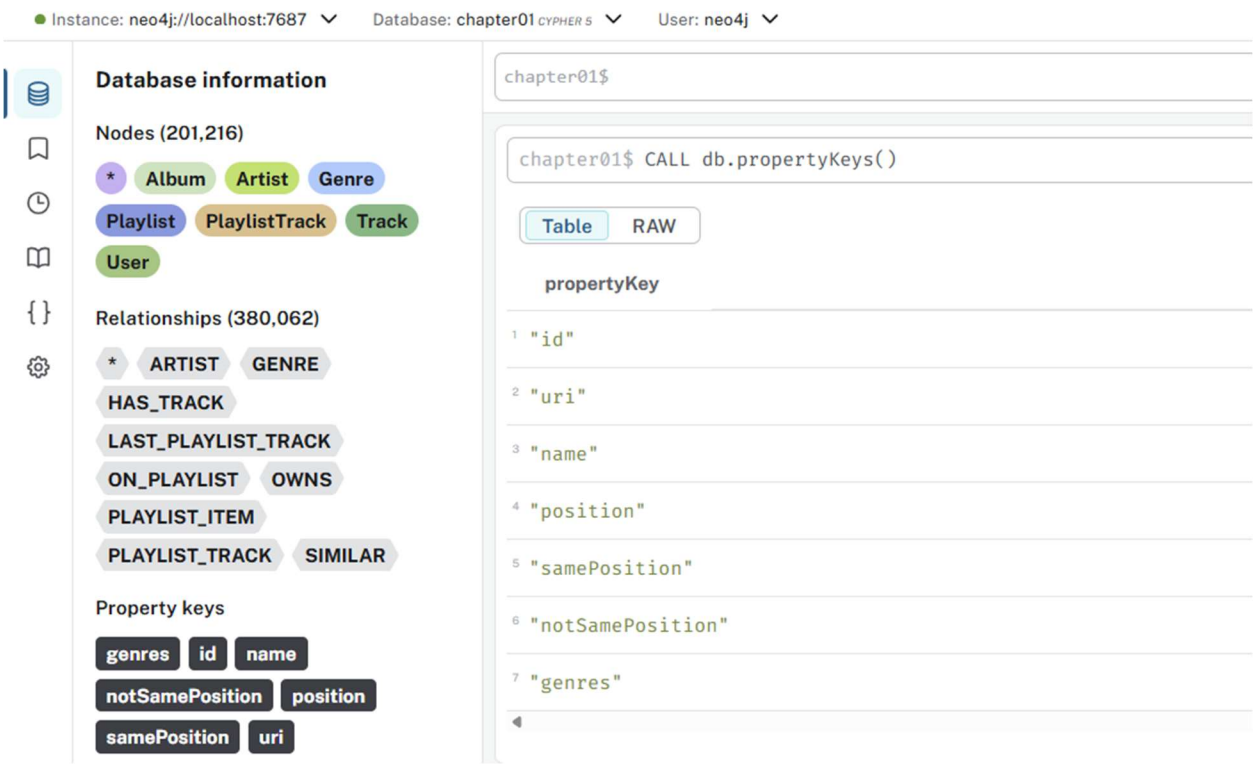
Show all relationship types:

CALL db.relationshipTypes()



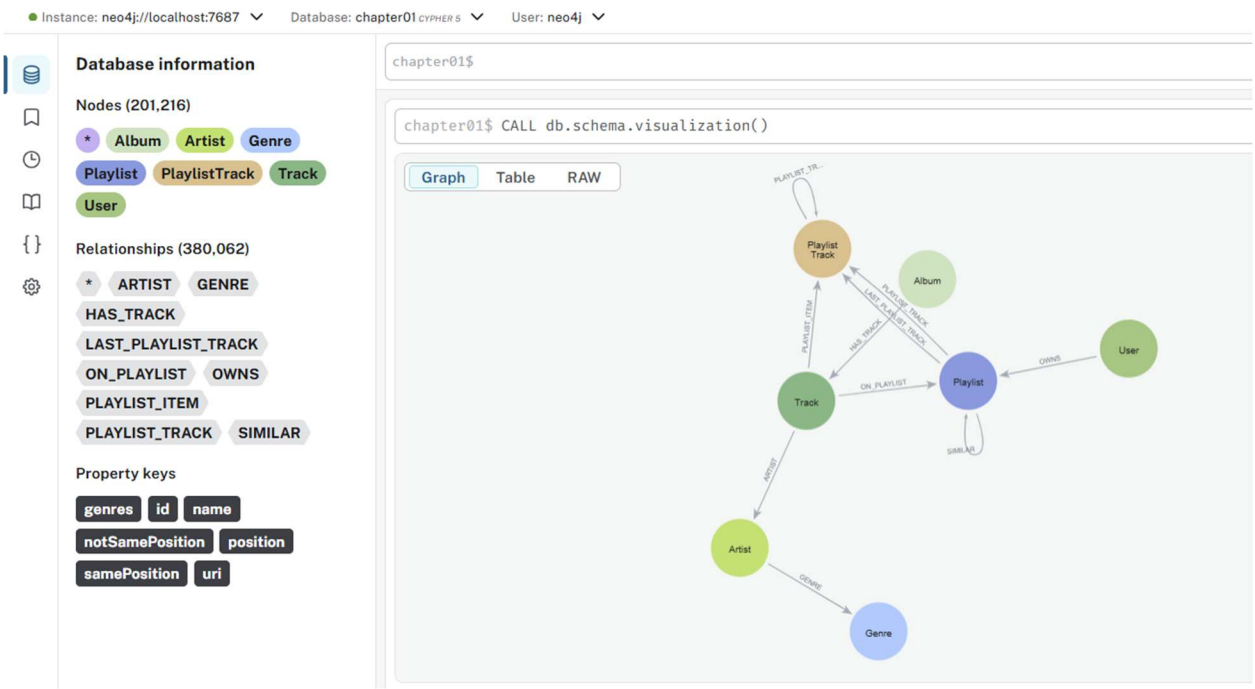
Show all property keys:

```
CALL db.propertyKeys()
```



Show schema overview

CALL db.schema.visualization()



Check if the playlist has any tracks at all:

chapter01$

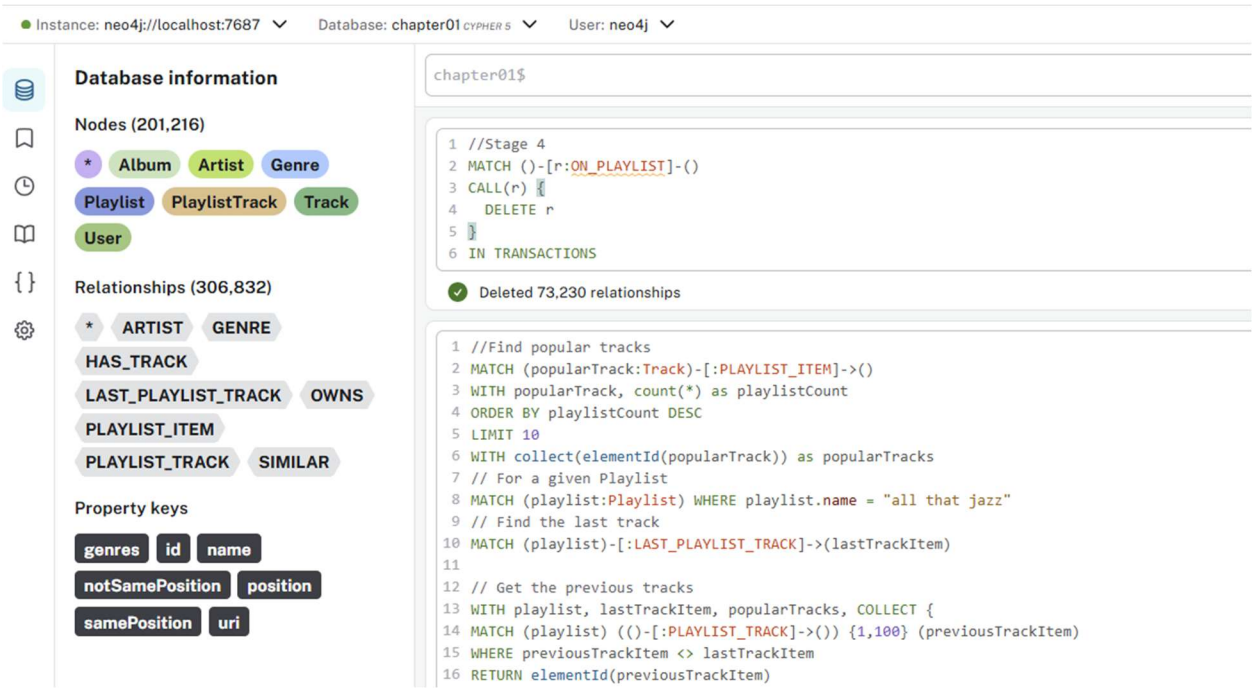
```
MATCH (t)-[:ON_PLAYLIST]->(playlist:Playlist {name: "all that jazz", id:
"0kI1NEhdo4bJemAqZn5xJd"})
MATCH (playlist)-[:LAST_PLAYLIST_TRACK]->(lastTrackItem)
WITH count(t) as cnt, playlist.name as pname , lastTrackItem.name as lst
RETURN pname, cnt,lst
```

chapter01$

```
MATCH (playlist:Playlist {name: "all that jazz"})
//OPTIONAL MATCH (playlist)-[r:PLAYLIST_TRACK]->(PlaylistTrack)
OPTIONAL MATCH (t)-[:ON_PLAYLIST]-> (playlist)-[r:LAST_PLAYLIST_TRACK]-
>(LastPlaylistTrack)<-[:PLAYLIST_ITEM]-(t2)
RETURN count(*) as lst, playlist.name, t2.name , t2.id, t2.uri
```

013-refactor-playlist-linked-5

```
chapter01$
        //Stage 4
        MATCH ()-[r:ON_PLAYLIST]-()
        CALL(r) {
          DELETE r
        }
        IN TRANSACTIONS
```



Simple DELETE (Recommended)

```
//Stage 4 - Clean up ON_PLAYLIST relationships
MATCH ()-[r:ON_PLAYLIST]-()
DELETE r
```

Batched DELETE for Large Datasets

```
//Stage 4 - Batched cleanup of ON_PLAYLIST relationships
MATCH ()-[r:ON_PLAYLIST]-()
CALL {
  WITH r
  DELETE r
} IN TRANSACTIONS OF 10000 ROWS
```

Using apoc.periodic.iterate (For Very Large Datasets)

```
//Stage 4 - Efficient batch deletion for very large datasets
CALL apoc.periodic.iterate(
  "MATCH ()-[r:ON_PLAYLIST]-() RETURN r",
  "DELETE r",
  {batchSize: 10000, parallel: false}
)
```