POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

ENTERPRISE ICT ARCHITECTURES PROJECT

Author(s): **Carbone Emanuele (10726300)**
Carli Federico (10713446)
Ferrazzano Andrea (10703279)
Gorini Marco (10710545)
Pauselli Tommaso (10797253)

Group Number: **6**

Academic Year: 2023-2024

Contents

1 | Second Project Delivery

1.1. Introduction

In this project we are creating a database that simulates the management of a music application that allows users to listen to songs and manage playlists. The database was created in Neo4j from the relational database previously created in MySql. In addition, there are examples of queries in Neo4j of various types to possibly test the correctness of the database. Included in the project report there is the BPMN model of the procedure for uploading a song by an artist.

1.2. Schematic Representation

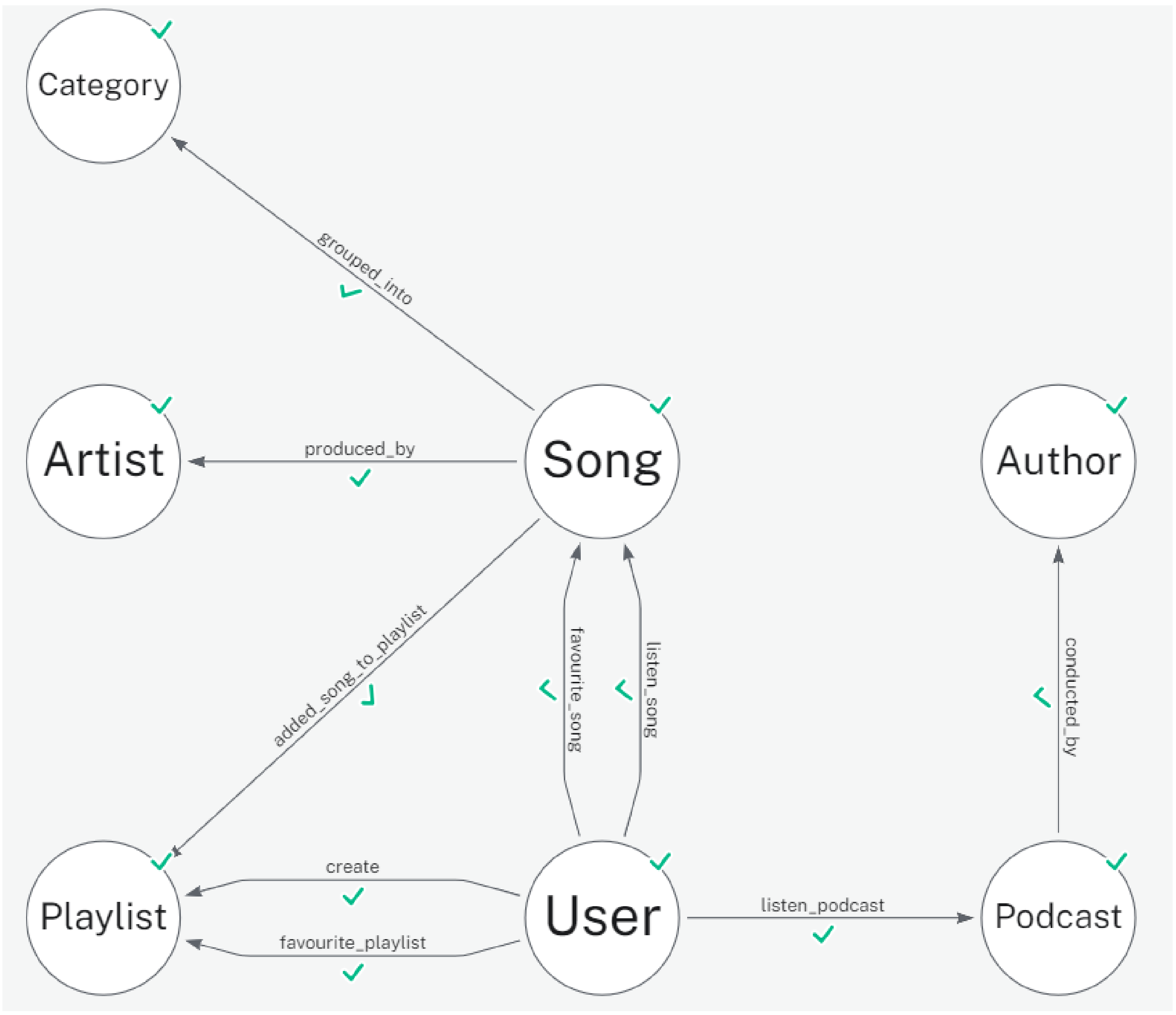


Figure 1.1: Schematic Representation

Our first task was transferring our database from SQL to Neo4j. Neo4j is a graph database management system that stores and queries data as nodes, relationships and properties. Unlike relational databases, Neo4j does not require a predefined schema and allows for flexible and expressive data modeling. Also, in Neo4j, there are no foreign keys in the same way as you would find them in a relationship database.

Thus, we started converting tables and foreign keys into nodes and relationships. In particular, every entity that we had in SQL became a node in neo4j and every bridge table became a relationship between nodes. An important exception, in our case, was listening to songs or podcasts. In SQL, we created a Listen entity, which had User's and Song's (or Podcast's) foreign keys. Neo4j, as it is a graph database, lets us handle

Listen easily by creating two relationships: one from user to song called listen_song, one from user to podcast called listen_podcast. We couldn't do this in SQL, as recording two instances of an user listening to the same song requires creating an entity or adding another key to the relationship thus creating a new table.

1.3. Detail

The model has the following nodes:

- **User**: we decided to create this node in order to represent the user's profile and his information.It has 5 attributes and we choose the mail attribute (string) as the primary key because it is a unique information.The "Username" attribute (string) means the nickname with which the user is represented in the platform. Premium is a Boolean attribute that indicates whether the user has paid for the premium subscription and R.Date (datatime) indicates the date the user registered. The password attribute (string) contains the account's password;
- **Podcast**: we think that the best way to uniquely identify (integer) it is through an auto-incremented "id" (because there could be podcasts with the same titles), and it has the attributes "title" (string), "duration"(integer, as Duration(s) in the model) and "publicationDate"(datatime);
- **Song**: it has the same attributes of Podcast and the attribute "text" (string);
- **Playlist** : the "id" attribute (integer) is the primary key and identifies the playlist, the Boolean attribute "isPublic" indicates whether the playlist is public or not. "Title" (string) indicates the name given by the creator and can't be a key because more playlists might have the same name;
- **Artist** : we think that the best way to uniquely identify it is through an auto-incremented "id" (integer), and it has the attributes "name" (string), "surname"(string), "stageName" (string, in case it could have a different name to introduce to the audience), "dateOfBirth" (datatime) and "recordLabel" (string);
- **Author**: we think that the best way to uniquely identify it is through an auto-incremented "id" (integer), and it has the attribute "name" (string), "surname" (string), and "productionLabel" (string);
- **Category**: this node represents the genre of a song. We create a node instead of considering it as an attribute in the song node because it has the "description" attribute (string) and,as seen before in the creator case, a song can belong to more categories at the same time so we would have needed a table anyway in order to represent the multivalued attribute;

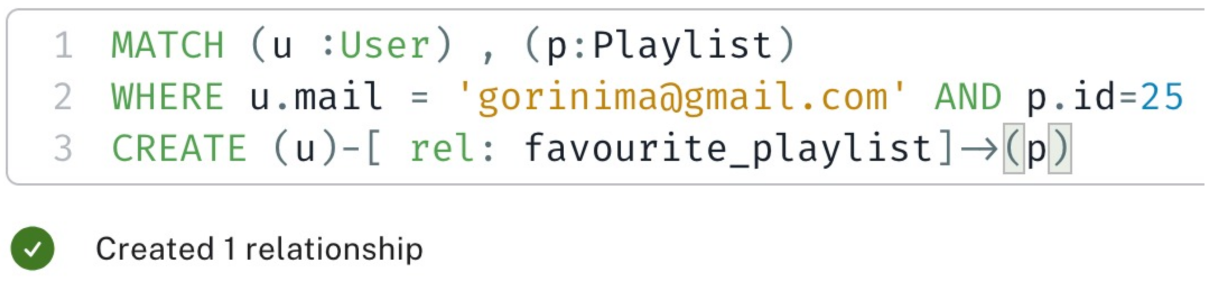
The model has the following relations:

- **create**: it's between "User" and "Playlist", because the user create the playlists.

- **listen_podcast**: it's between "User" and "Podcast", the relation has the attributes "date" (datatime) to store when the podcast was listened, "finished" (boolean) to store if the user finished to listen the podcast and "mailUser" (string) to store the user's mail.
- **listen_song**: it's between "User" and "Song", the relation has the same attributes of listen_podcast.
- **favourite_playlist**: it's between "User" and "Playlist", because the user is able to add the playlists as favourite.
- **favourite_song**: it's between "User" and "Song", because the user is able to add the songs as favourite.
- **added_song_to_playlist**: it's between "Playlist" and "Song", to understand in which playlist the song has been added, and it has the attribute "addedDate" (datatime) to understand when it has been added.
- **conducted_by**:it's between "Author" and "Podcast", to understand which author conducted the podcast.
- **produced_by**: it's between "Artist" and "Song", to understand which artist has produced the song.
- **grouped_into**: it's between "Song" and "Category", to understand in which categories the songs habe been grouped.

1.4. Neo4j Queries

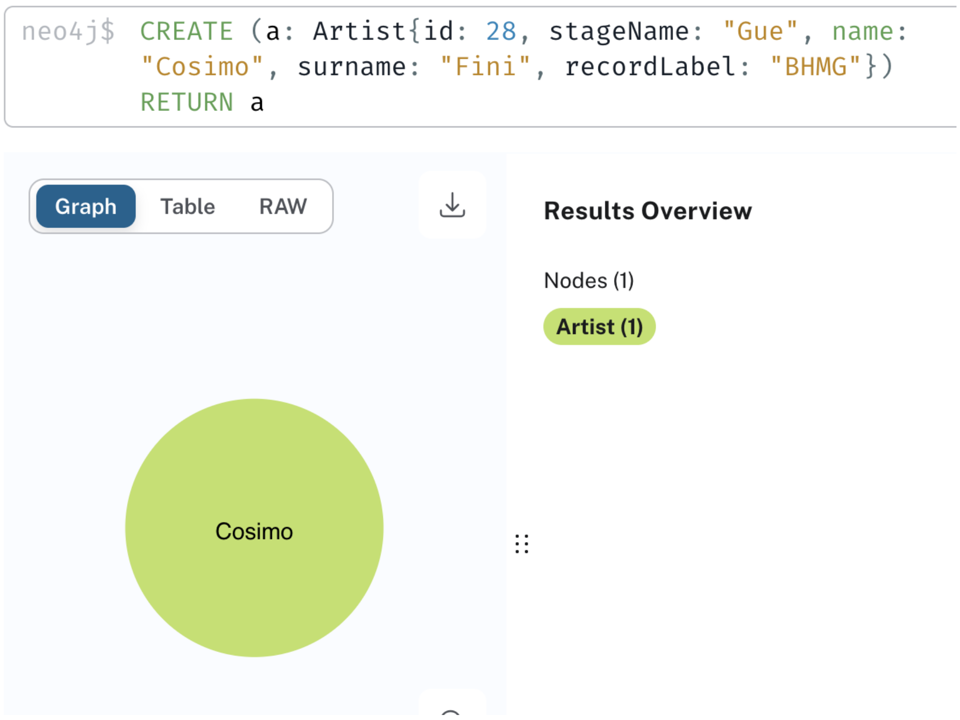
1.4.1. Creation/Update/Delete Queries



```
MATCH (u :User) , (p:Playlist)
WHERE u.mail = 'gorinima@gmail.com' AND p.id=25
CREATE (u)-[ rel: favourite_playlist]->(p)
```

Created 1 relationship

This code allows to create a relationship "favourite_playlist" between the user "gorinima@gmail.com" and the playlist "25".



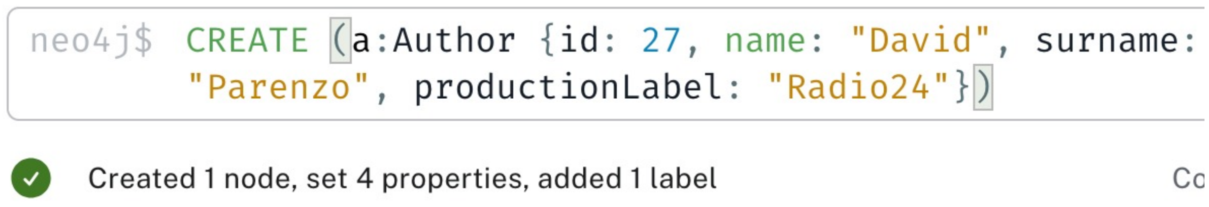
```
neo4j$ CREATE (a: Artist{id: 28, stageName: "Gue", name:
       "Cosimo", surname: "Fini", recordLabel: "BHMG"})
       RETURN a
```

Results Overview

Nodes (1)

Artist (1)

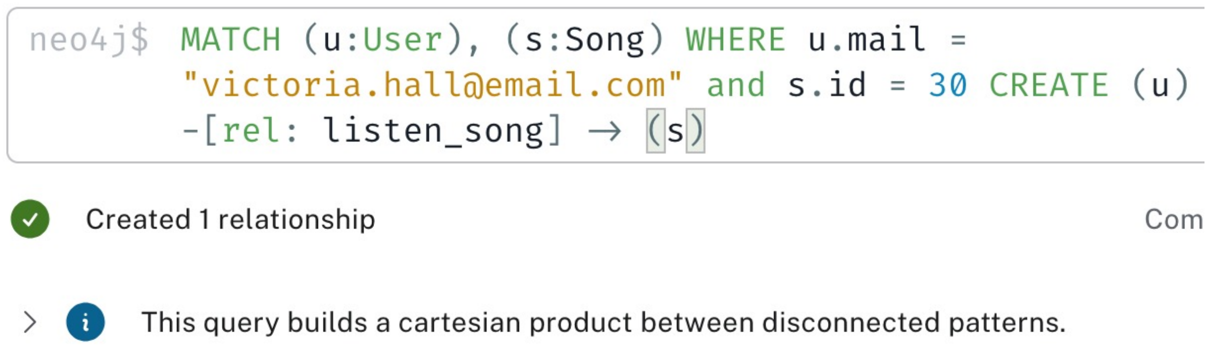
This code allows to create an entity "Artist".



```
neo4j$ CREATE (a:Author {id: 27, name: "David", surname:
       "Parenzo", productionLabel: "Radio24"})
```

Created 1 node, set 4 properties, added 1 label   Co

This code allows to create an entity "Author".

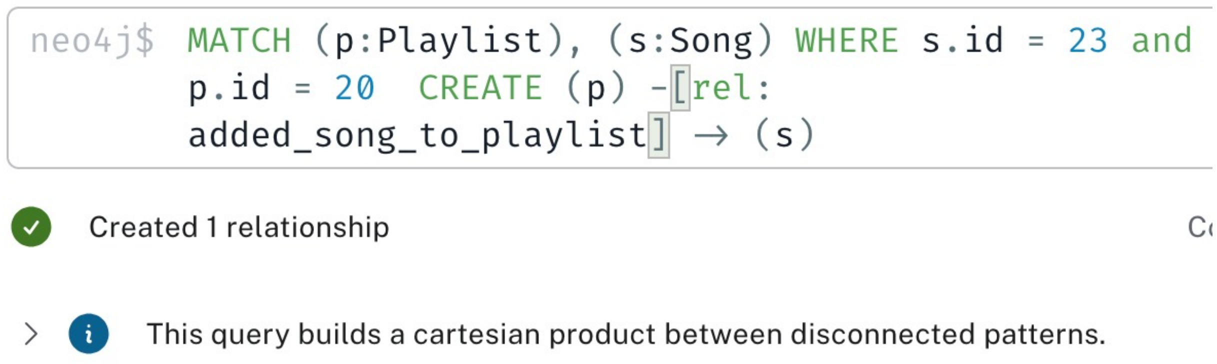


```
neo4j$ MATCH (u:User), (s:Song) WHERE u.mail =
       "victoria.hall@email.com" and s.id = 30 CREATE (u)
       -[rel: listen_song] → (s)
```

Created 1 relationship   Com

This query builds a cartesian product between disconnected patterns.

This code allows to add the relationship indicating that victoria listens to the song with id 30

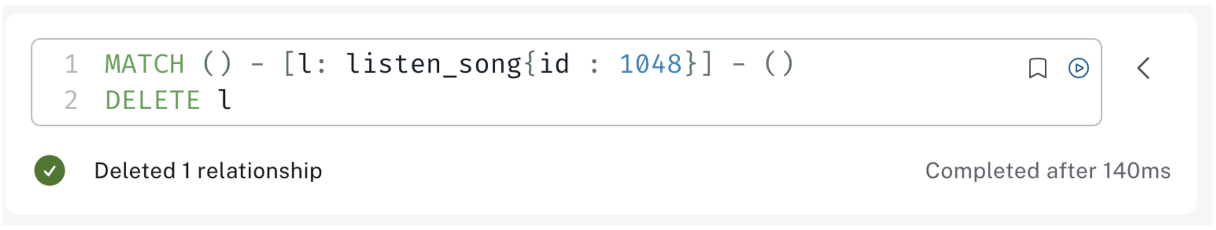


```
neo4j$ MATCH (p:Playlist), (s:Song) WHERE s.id = 23 and
       p.id = 20  CREATE (p) -[rel:
       added_song_to_playlist] → (s)
```

Created 1 relationship   Co

This query builds a cartesian product between disconnected patterns.

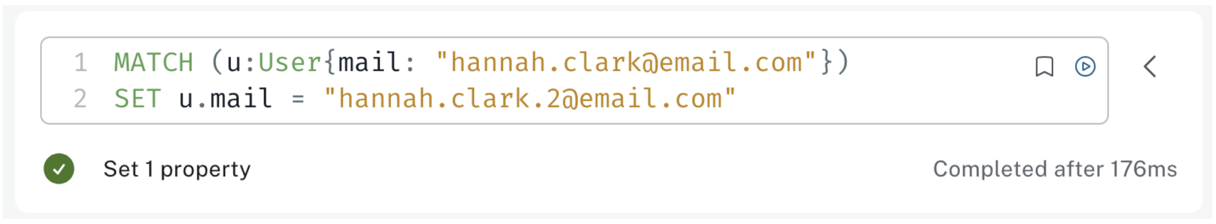
This code allows to create the relationship indicating that the song with id 23 is added to the playlist 20



```
MATCH () - [l: listen_song{id : 1048}] - ()
DELETE l
```

Deleted 1 relationship Completed after 140ms

This code allow to delete the relationship indicating every listen of the song with id 1048



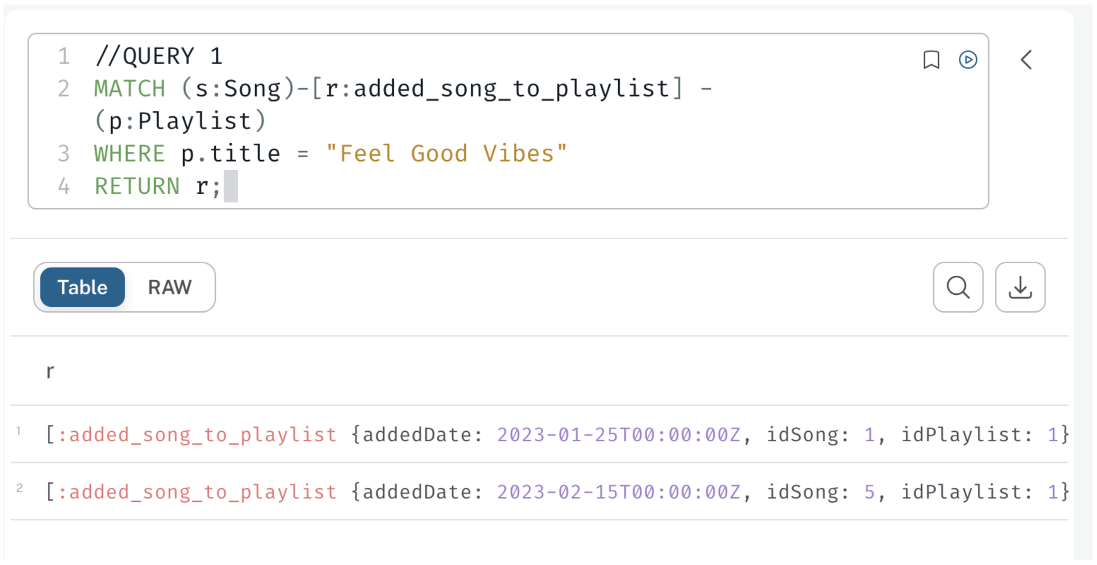
```
MATCH (u:User{mail: "hannah.clark@email.com"})
SET u.mail = "hannah.clark.2@email.com"
```

Set 1 property Completed after 176ms

This code allows to update hannah's mail.

1.4.2. Complex Queries

1) These songs feel good

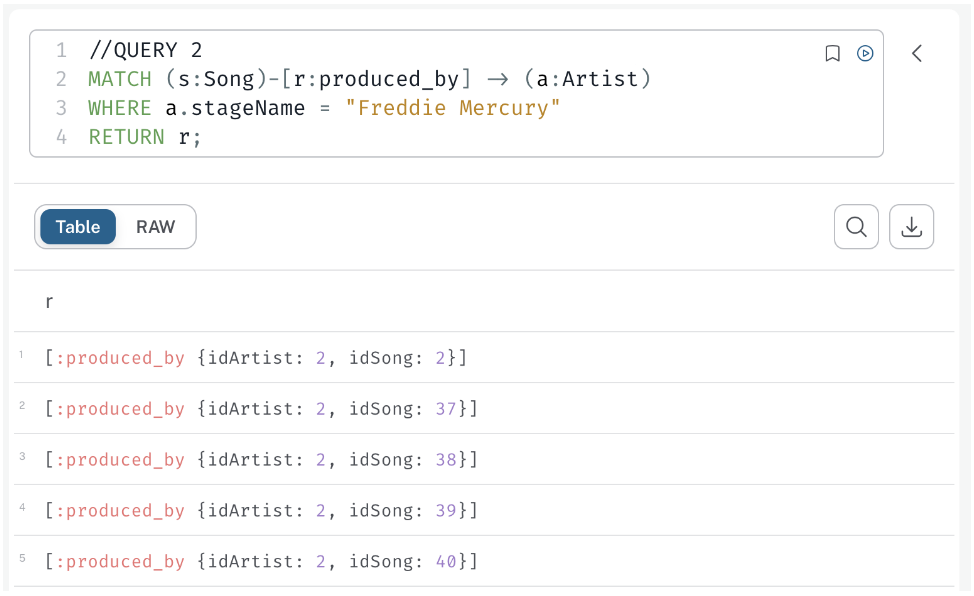


```
//QUERY 1
MATCH (s:Song)-[r:added_song_to_playlist] -
(p:Playlist)
WHERE p.title = "Feel Good Vibes"
RETURN r;
```

```
r
1 [:added_song_to_playlist {addedDate: 2023-01-25T00:00:00Z, idSong: 1, idPlaylist: 1}
2 [:added_song_to_playlist {addedDate: 2023-02-15T00:00:00Z, idSong: 5, idPlaylist: 1}
```

The code allows to find the relation between Song and Playlist and it returns the songs added to the playlist called "Feel Good Vibes".

2) What has Freddie produced?



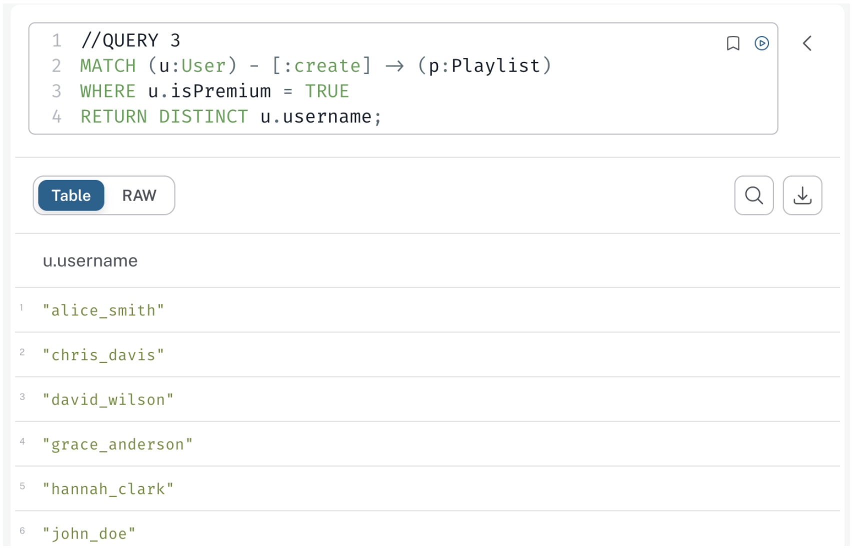
```
//QUERY 2
MATCH (s:Song)-[r:produced_by] → (a:Artist)
WHERE a.stageName = "Freddie Mercury"
RETURN r;
```

Table RAW

```
r
1 [:produced_by {idArtist: 2, idSong: 2}]
2 [:produced_by {idArtist: 2, idSong: 37}]
3 [:produced_by {idArtist: 2, idSong: 38}]
4 [:produced_by {idArtist: 2, idSong: 39}]
5 [:produced_by {idArtist: 2, idSong: 40}]
```

The code allows to find the relation between Song and Artist and it returns the songs produced by "Freddie Mercury".

3) Premium creators



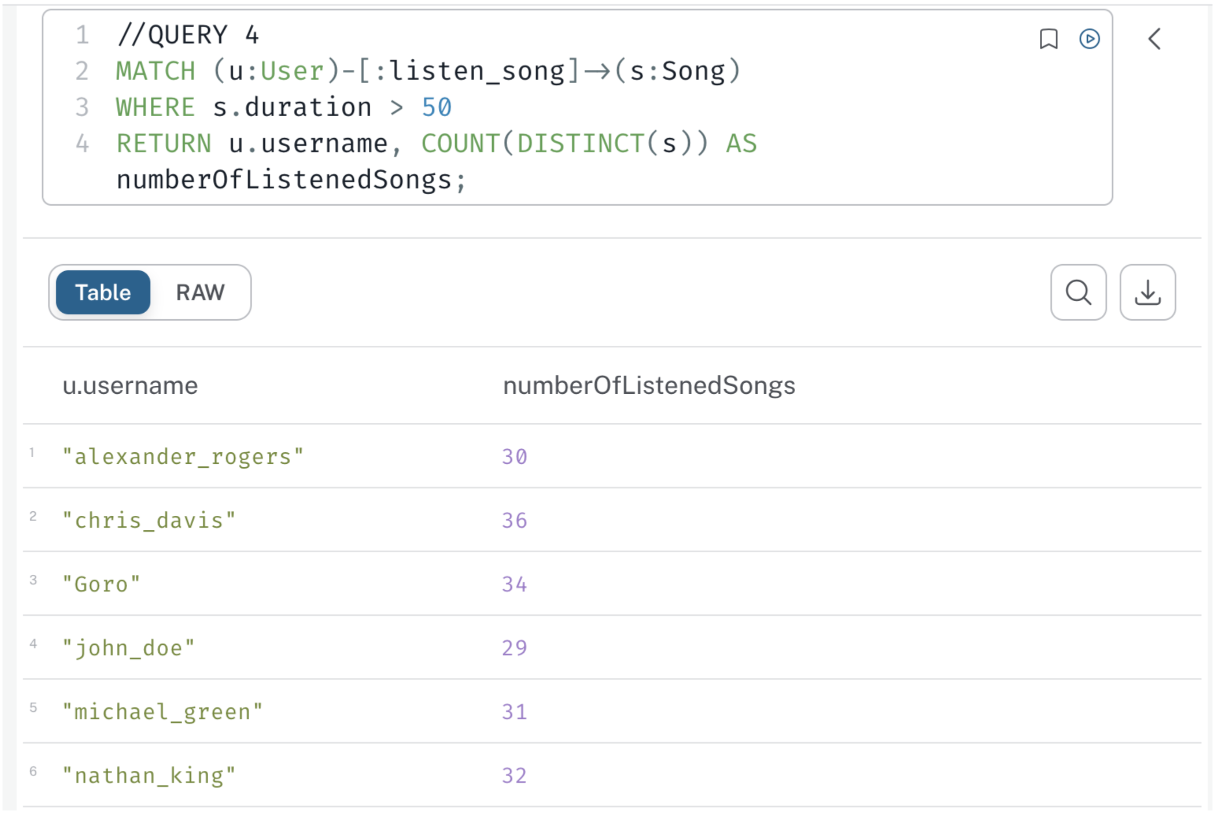
```
//QUERY 3
MATCH (u:User) - [:create] → (p:Playlist)
WHERE u.isPremium = TRUE
RETURN DISTINCT u.username;
```

Table RAW

```
u.username
1 "alice_smith"
2 "chris_davis"
3 "david_wilson"
4 "grace_anderson"
5 "hannah_clark"
6 "john_doe"
```

The code allows to find the users that created at least one playlist and that are premium, and it returns their usernames.

4) Users for long songs



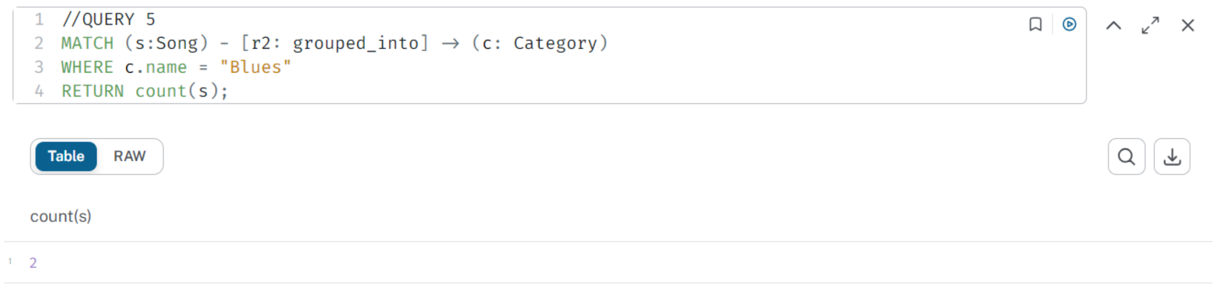
```
//QUERY 4
MATCH (u:User)-[:listen_song]→(s:Song)
WHERE s.duration > 50
RETURN u.username, COUNT(DISTINCT(s)) AS
numberOfListenedSongs;
```

Table RAW

	u.username	numberOfListenedSongs
1	"alexander_rogers"	30
2	"chris_davis"	36
3	"Goro"	34
4	"john_doe"	29
5	"michael_green"	31
6	"nathan_king"	32

The code allows to find the users that listened to songs that are longer than 50 seconds, and it returns their usernames and the number of different songs longer than 50 seconds that they've listened.

5) Blues songs



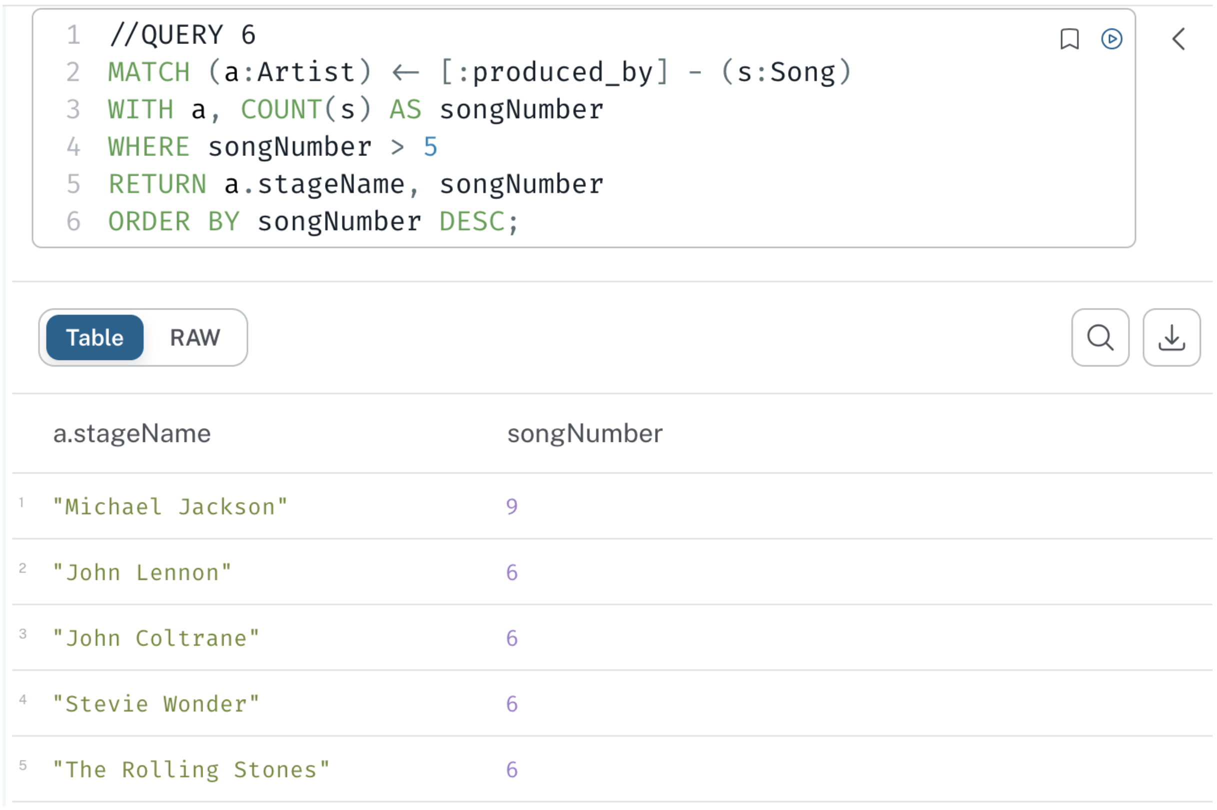
```
//QUERY 5
MATCH (s:Song) - [r2: grouped_into] → (c: Category)
WHERE c.name = "Blues"
RETURN count(s);
```

Table RAW

	count(s)
1	2

The code returns the number of songs in the "Blues" category.

6) Veteran artists

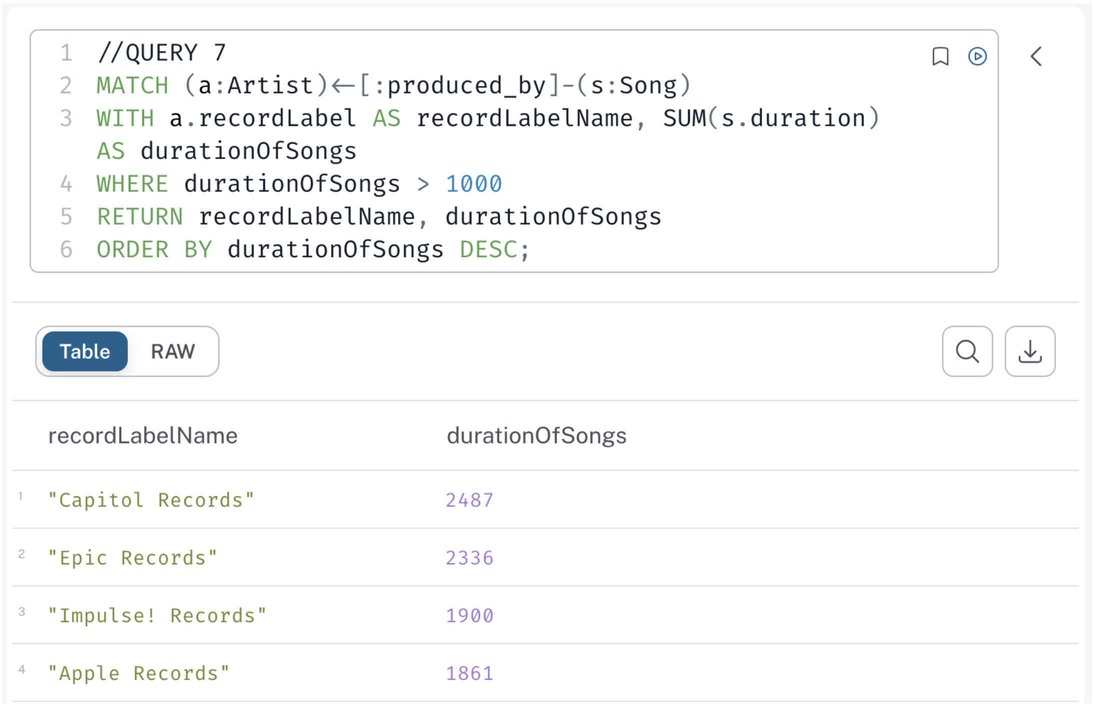


```
//QUERY 6
MATCH (a:Artist) ← [:produced_by] - (s:Song)
WITH a, COUNT(s) AS songNumber
WHERE songNumber > 5
RETURN a.stageName, songNumber
ORDER BY songNumber DESC;
```

	a.stageName	songNumber
1	"Michael Jackson"	9
2	"John Lennon"	6
3	"John Coltrane"	6
4	"Stevie Wonder"	6
5	"The Rolling Stones"	6

The code allows to find the Artists that produced more than 5 Songs, and it returns their stageName and the number of Songs produced ordering them from the one that produced more of them.

7) Strongest labels

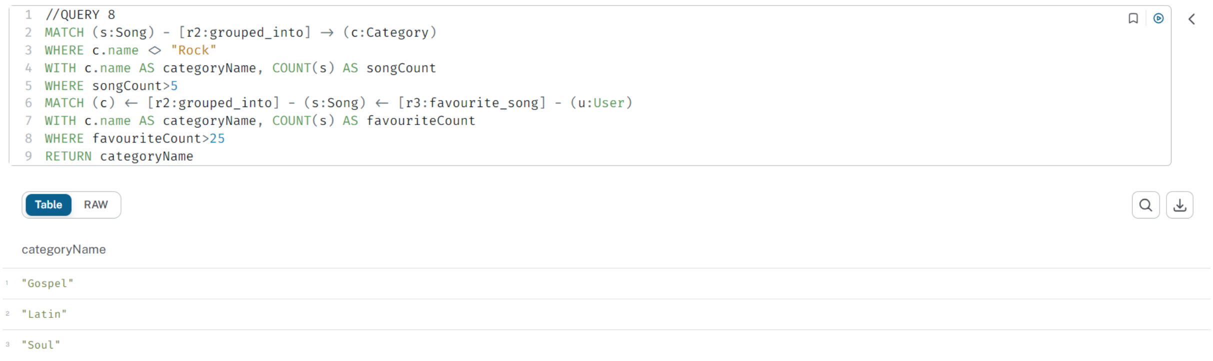


```
//QUERY 7
MATCH (a:Artist)<-[:produced_by]-(s:Song)
WITH a.recordLabel AS recordLabelName, SUM(s.duration)
AS durationOfSongs
WHERE durationOfSongs > 1000
RETURN recordLabelName, durationOfSongs
ORDER BY durationOfSongs DESC;
```

	recordLabelName	durationOfSongs
1	"Capitol Records"	2487
2	"Epic Records"	2336
3	"Impulse! Records"	1900
4	"Apple Records"	1861

The code allows to find record labels whose artist have produced songs that, summed together, exceed 1000 seconds of duration. The output includes the record label name and the total duration of the produced songs, sorted by duration.

8) Rock haters

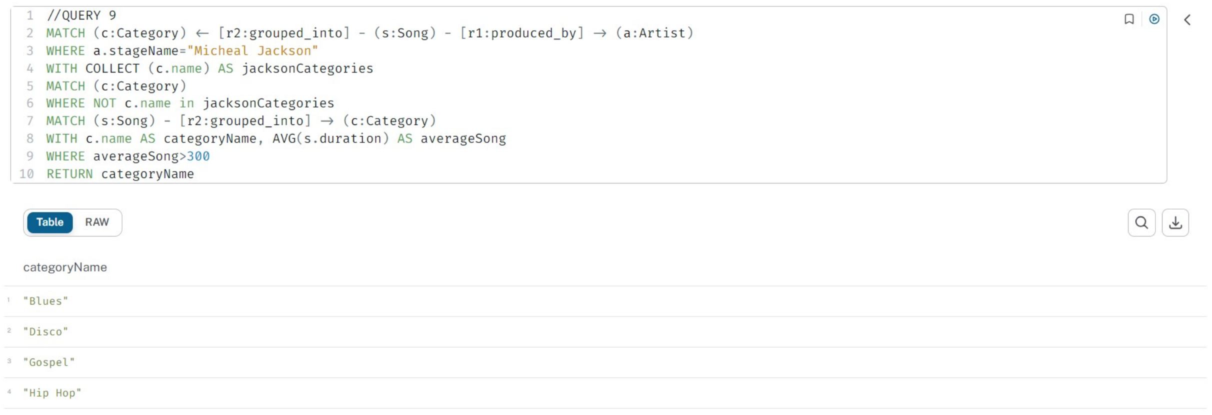


```
//QUERY 8
MATCH (s:Song) - [r2:grouped_into] -> (c:Category)
WHERE c.name <> "Rock"
WITH c.name AS categoryName, COUNT(s) AS songCount
WHERE songCount>5
MATCH (c) <- [r2:grouped_into] - (s:Song) <- [r3:favourite_song] - (u:User)
WITH c.name AS categoryName, COUNT(s) AS favouriteCount
WHERE favouriteCount>25
RETURN categoryName
```

	categoryName
1	"Gospel"
2	"Latin"
3	"Soul"

The code allows to find the categories different from "rock" with more than 5 songs and which have more than 25 songs picked as favourite, and it returns their categories.

9) No jacksoned categories

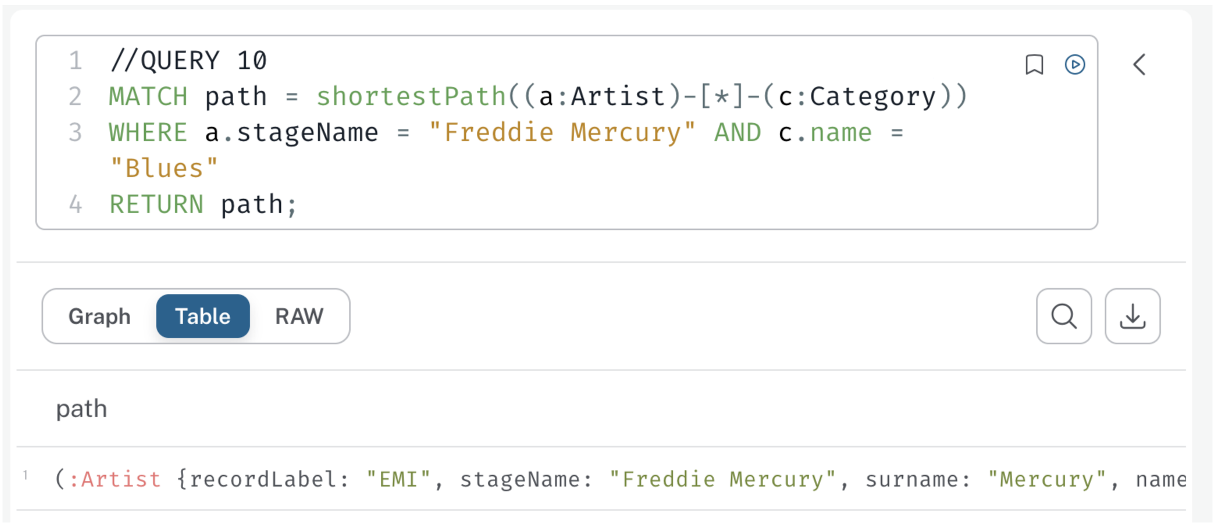


```
//QUERY 9
MATCH (c:Category) <- [r2:grouped_into] - (s:Song) - [r1:produced_by] -> (a:Artist)
WHERE a.stageName="Micheal Jackson"
WITH COLLECT (c.name) AS jacksonCategories
MATCH (c:Category)
WHERE NOT c.name in jacksonCategories
MATCH (s:Song) - [r2:grouped_into] -> (c:Category)
WITH c.name AS categoryName, AVG(s.duration) AS averageSong
WHERE averageSong>300
RETURN categoryName
```

	categoryName
1	"Blues"
2	"Disco"
3	"Gospel"
4	"Hip Hop"

The code allows to find the songs' categories not produced by "Micheal Jackson" that have an average duration higher than 300 seconds, and it returns the categories' name.

10) Freddie's Blues

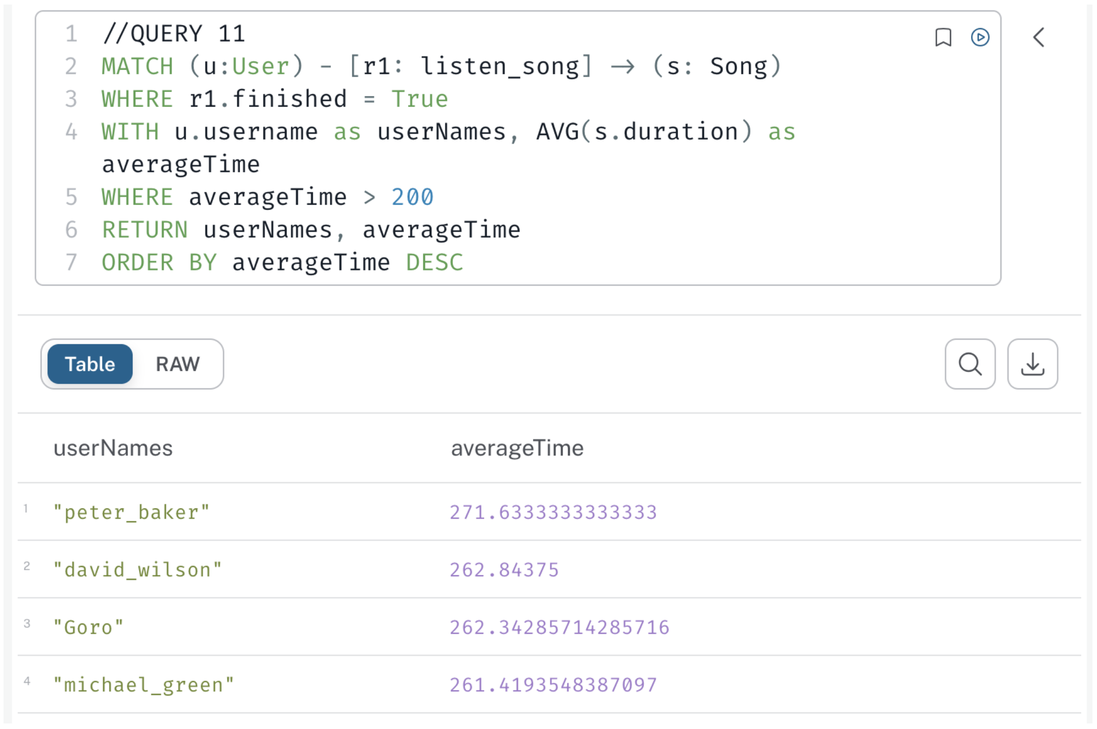


```
//QUERY 10
MATCH path = shortestPath((a:Artist)-[*]-(c:Category))
WHERE a.stageName = "Freddie Mercury" AND c.name =
"Blues"
RETURN path;
```

```
path
1 (:Artist {recordLabel: "EMI", stageName: "Freddie Mercury", surname: "Mercury", name
```

The code allows to find the shortest path between the Artist "Freddie Mercury" and the Category "Blues".

11) Long listeners

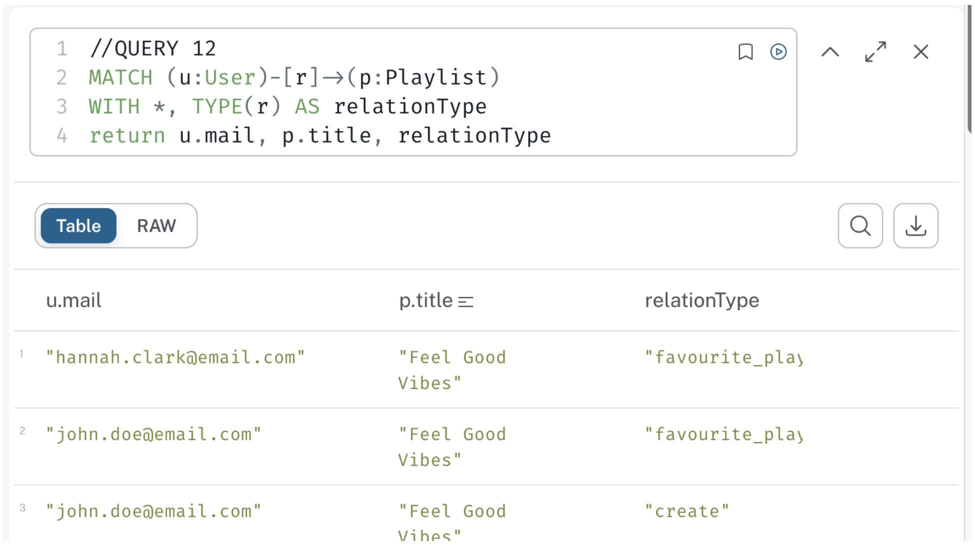


```
//QUERY 11
MATCH (u:User) - [r1: listen_song] → (s: Song)
WHERE r1.finished = True
WITH u.username as userNames, AVG(s.duration) as
averageTime
WHERE averageTime > 200
RETURN userNames, averageTime
ORDER BY averageTime DESC
```

	userNames	averageTime
1	"peter_baker"	271.6333333333333
2	"david_wilson"	262.84375
3	"Goro"	262.34285714285716
4	"michael_green"	261.4193548387097

The code allows to find the users which have an average length of listened to songs greater than 200. The output includes the username of those users and the average duration of the songs they have listened.

12) Relation finder

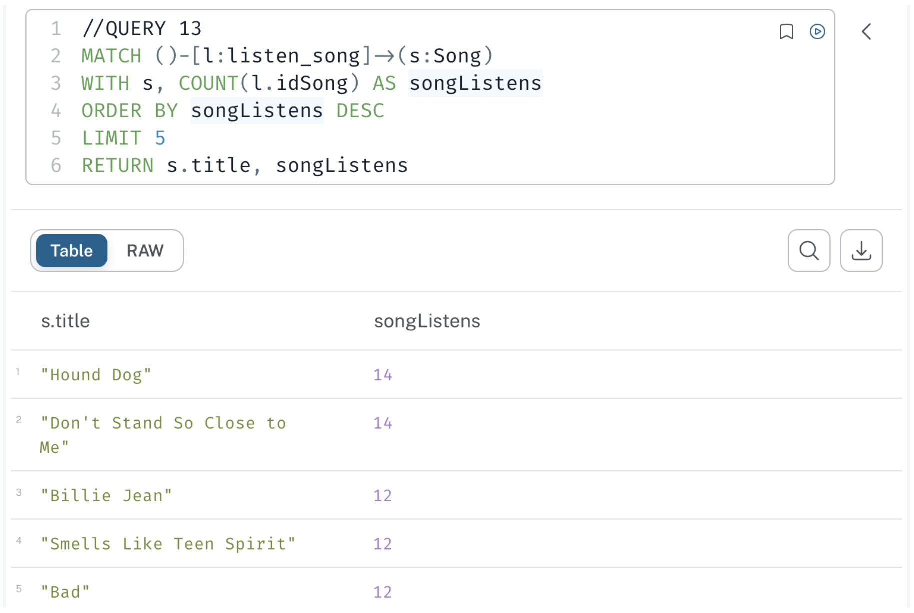


```
//QUERY 12
MATCH (u:User)-[r]→(p:Playlist)
WITH *, TYPE(r) AS relationType
return u.mail, p.title, relationType
```

	u.mail	p.title	relationType
1	"hannah.clark@email.com"	"Feel Good Vibes"	"favourite_play
2	"john.doe@email.com"	"Feel Good Vibes"	"favourite_play
3	"john.doe@email.com"	"Feel Good Vibes"	"create"

The code allows to find the username, the playlist title and the their relations type.

13) Top 5 songs

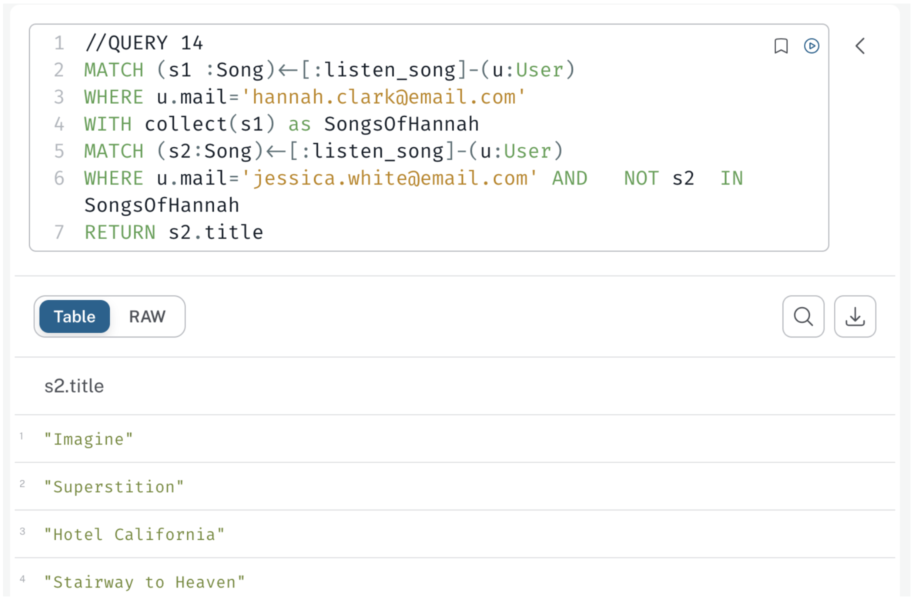


```
//QUERY 13
MATCH ()-[l:listen_song]→(s:Song)
WITH s, COUNT(l.idSong) AS songListens
ORDER BY songListens DESC
LIMIT 5
RETURN s.title, songListens
```

	s.title	songListens
1	"Hound Dog"	14
2	"Don't Stand So Close to Me"	14
3	"Billie Jean"	12
4	"Smells Like Teen Spirit"	12
5	"Bad"	12

The code allows to find the top 5 most listened to songs and how many times they have been listened, and it returns their title and how many times they have been listened sorted by listens.

14) What has Jessica listened to?



```
//QUERY 14
MATCH (s1 :Song)←[:listen_song]-(u:User)
WHERE u.mail='hannah.clark@email.com'
WITH collect(s1) as SongsOfHannah
MATCH (s2:Song)←[:listen_song]-(u:User)
WHERE u.mail='jessica.white@email.com' AND   NOT s2  IN
SongsOfHannah
RETURN s2.title
```

	s2.title
1	"Imagine"
2	"Superstition"
3	"Hotel California"
4	"Stairway to Heaven"

The code allows to find the songs listened by the user with the mail "jessica.white@email.com" and that are not listened by the user with the mail "hannah.clark@email.com", and it returns the title of those songs.

1.5. BPMN

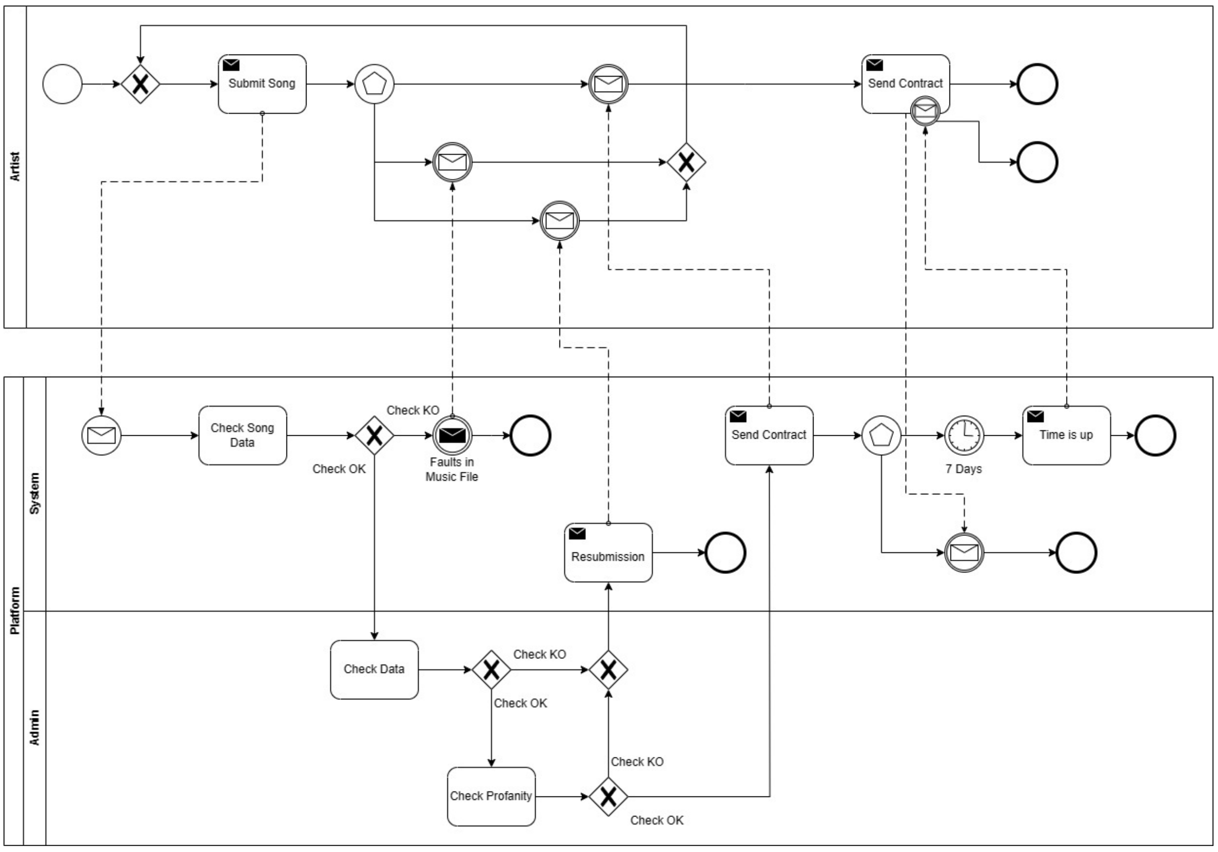


Figure 1.2: BPMN

BPMN stands for Business Process Model and Notation. It is a standard graphical notation for business process modeling and communication. We modeled the submission process of a song. The process starts with an artist sending a song's data, the system then performs some checks, also involving administrators, and asks for resubmission if it finds errors. If the song is approved, a contract is sent to the artist, who has to sign it and send it back.
We made the following assumptions:

- **If an error is found during the checks**: A. On the system's side, after notifing the artist, the process ends. It would then start again when it receives a new submission. B. On the artist's side, we have decided to model a loop. To do this we used an event-based gateway: if a resubmission message arrives from the system (either due to system or administrator check), the artist will loop back to sending the song data; if the contract arrives, then the artist will have to complete the next task, that is signing the contract and sending it back to the system.

- **Administrators check**: We have decided to split in two the checks administrators perform. This way, if the first check returns an error, the second check is not performed.
- **Contract**: When the system sends the contract to the artist, the artist will have to sign it and send it back. If seven days pass without a response, the system will notify the artist and terminate the process. To model this on the artist's side we used an interrupting message, sent from the system and received through a boundary catch.