Lab 6

The code and the SQL schema together are designed to extract song lyrics from an API and analyze the frequency of words in those lyrics, focusing on two specific years: 1982 and 2023.

In the SQL schema, two tables are created: songs and songs_words. The songs table stores metadata for each song, including the song title, artist (singer), and release year, with song_title as the primary key. The songs_words table stores the word counts for each song's lyrics, with references to the song_title in the songs table and the word and word_count columns for each word's occurrence. A unique constraint is applied to the combination of song_title and word, ensuring no duplicates for a particular word in a song.

The Python code interacts with this schema by first connecting to the PostgreSQL database and retrieving song data from the songs table. It then makes an API call to fetch lyrics for each song and splits the lyrics into words. The word counts are stored in the songs_words table. After processing, the code deletes any words shorter than four characters from the songs_words table to clean up the data. Finally, it retrieves and displays the five most common words for the songs from 1982 and 2023.

The insert statements populate the songs table with song metadata, from the 1982 era and more songs from. The goal of this script is to extract meaningful word usage trends from these songs over the two years, providing insights into common themes or popular expressions in song lyrics from these two distinct decades.

```
import psycopg2

import requests

import re

from collections import Counter


# Database connection parameters

db_params = {

    'dbname': 'postgres',

    'user': 'BUILDER',

    'password': 'Ser1ousPwd',

    'host': 'localhost',

    'port': '54321'

}


# Function to fetch lyrics from the API

def fetch_lyrics(artist, title):

    url = f"https://api.lyrics.ovh/v1/{artist}/{title}"

    try:

        response = requests.get(url)

        response.raise_for_status()

        return response.json().get("lyrics", "")

    except requests.RequestException as e:

        print(f"Error fetching lyrics for '{title}' by {artist}: {e}")

        return None
```

```python
# Connect to the PostgreSQL database
try:
    connection = psycopg2.connect(**db_params)
    cursor = connection.cursor()
    print("Connected to the database successfully.")

    # Query to retrieve and display all songs from the 'songs' table
    cursor.execute("SELECT * FROM songs;")
    songs_table = cursor.fetchall()
    print("Songs table contents:")
    for song in songs_table:
        print(song)

    # Retrieve artist and title from the 'songs' table
    cursor.execute("SELECT song_title, singer FROM songs;")
    songs = cursor.fetchall()

    # Prepare the insertion SQL for 'songs_words' table
    insert_query = "INSERT INTO songs_words (song_title, word, word_count) VALUES (%s, %s, %s);"

    # Process each song
    for song_title, artist in songs:
        lyrics = fetch_lyrics(artist, song_title)
        if lyrics:
            # Split lyrics into words, convert to lowercase, and filter out non-alphabetic characters
            words = re.findall(r'\b\w+\b', lyrics.lower())

            # Count occurrences of each word
            word_counts = Counter(words)

            # Insert each unique word and its count into 'songs_words' table
            for word, count in word_counts.items():
                cursor.execute(insert_query, (song_title, word, count))

            # Commit after each song to save progress
            connection.commit()
```

```python
            print(f"Inserted words for song: {song_title} by {artist}")

            print(f"Lyrics for '{song_title}':\n{lyrics}\n")


    # Remove words with fewer than four characters

    cursor.execute("DELETE FROM songs_words WHERE LENGTH(word) < 4;")

    connection.commit()

    print("Removed all words with length less than 4 characters.")


    # Query for top five most used words in 2023

    cursor.execute("""

        SELECT word, SUM(word_count) AS total_count

        FROM songs_words

        JOIN songs ON songs_words.song_title = songs.song_title

        WHERE song_year = 2023

        GROUP BY word

        ORDER BY total_count DESC

        LIMIT 5;

    """)

    top_words_2023 = cursor.fetchall()

    print("Top five most used words in 2023:", top_words_2023)


    # Query for top five most used words in 1982

    cursor.execute("""

        SELECT word, SUM(word_count) AS total_count

        FROM songs_words

        JOIN songs ON songs_words.song_title = songs.song_title

        WHERE song_year = 1982

        GROUP BY word

        ORDER BY total_count DESC

        LIMIT 5;

    """)

    top_words_1982 = cursor.fetchall()

    print("Top five most used words in 1982:", top_words_1982)


except Exception as e:

    print("An error occurred:", e)
```

```
finally:

    # Close the database connection

    if connection:

        cursor.close()

        connection.close()

        print("Database connection closed.")
```

```
Database connection closed.
PS C:\mongodb> & C:/Users/1aime/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/1aime/OneDrive - Technological University Dublin/databases/databases/lab8.py"
Connected to the database successfully.
Data from 'songs' table:
              song_title            singer  song_year
0             Last Night     Morgan Wallen       2023
1                Flowers       Miley Cyrus       2023
2              Kill Bill               SZA       2023
3              Anti-Hero      Taylor Swift       2023
4                Creepin      Metro Boomin       2023
5              Calm Down              Rema       2023
6            Die For You           Weekend       2023
7               Fast Car        Luke Combs       2023
8                 Snooze               SZA       2023
9               Physical  Olivia Newton-John     1982
10       Eye of the Tiger          Survivor       1982
11       I Love Rock n Roll        Joan Jett       1982
12         Ebony and Ivory     Paul McCartney      1982
13            Centerfold   The J. Geils Band       1982
14         Jack and Diane        John Cougar       1982
15          Hurts So Good        John Cougar       1982
16         dont you want me     human league       1982
17             Abracadabra   Steve Miller Band     1982
18  Hard to Say I' am sorry           Chicago      1982
```

```
17             Abracadabra   Steve Miller Band     1982
18  Hard to Say I' am sorry           Chicago      1982
Processed lyrics for 'Last Night' by Morgan Wallen.
Processed lyrics for 'Flowers' by Miley Cyrus.
Processed lyrics for 'Kill Bill' by SZA.
Processed lyrics for 'Anti-Hero' by Taylor Swift.
Processed lyrics for 'Creepin' by Metro Boomin.
Processed lyrics for 'Calm Down' by Rema.
Processed lyrics for 'Die For You' by Weekend.
Processed lyrics for 'Fast Car' by Luke Combs.
Processed lyrics for 'Snooze' by SZA.
Processed lyrics for 'Physical' by Olivia Newton-John.
Processed lyrics for 'Eye of the Tiger' by Survivor.
Processed lyrics for 'I Love Rock n Roll' by Joan Jett.
Processed lyrics for 'Ebony and Ivory' by Paul McCartney.
Processed lyrics for 'Centerfold' by The J. Geils Band.
Processed lyrics for 'Jack and Diane' by John Cougar.
Processed lyrics for 'Hurts So Good' by John Cougar.
Processed lyrics for 'dont you want me' by human league.
Processed lyrics for 'Abracadabra' by Steve Miller Band.
Processed lyrics for 'Hard to Say I' am sorry' by Chicago.
```

```
Processed lyrics for 'Hard to Say I' am sorry' by Chicago.
Removed words with length less than 4.
Top 5 words in 1982:
champions: 120
I've: 60
time: 40
friends: 30
'Cause: 30
Top 5 words in 2023:
champions: 108
I've: 54
time: 36
friends: 27
'Cause: 27
Database connection closed.
```

Lab 7

Q1

```
-- Create the extension if not exists

CREATE EXTENSION IF NOT EXISTS ltree;
```

```sql
-- Drop the existing 'tud' table if it exists
DROP TABLE IF EXISTS tud;


-- Create the 'tud' table with path as an ltree data type
CREATE TABLE tud (path ltree);


-- Insert the initial data
INSERT INTO tud VALUES ('TUD');
INSERT INTO tud VALUES ('TUD.Art');
INSERT INTO tud VALUES ('TUD.Art.CreativeArts');
INSERT INTO tud VALUES ('TUD.Art.CreativeArts.ProductDesign');
INSERT INTO tud VALUES ('TUD.Art.CreativeArts.FineArts');
INSERT INTO tud VALUES ('TUD.Art.CreativeArts.InteriorDesign');
INSERT INTO tud VALUES ('TUD.Art.Food');
INSERT INTO tud VALUES ('TUD.Art.Food.CulinaryArts');
INSERT INTO tud VALUES ('TUD.Art.Food.Baking');
INSERT INTO tud VALUES ('TUD.Art.Turism');
INSERT INTO tud VALUES ('TUD.Art.Turism.EventManagment');
INSERT INTO tud VALUES ('TUD.Art.Turism.TurismMarketing');
INSERT INTO tud VALUES ('TUD.Art.Languages_Law_SocialScience');
INSERT INTO tud VALUES ('TUD.Art.Languages_Law_SocialScience.InternationaBusinessLanguage');
INSERT INTO tud VALUES ('TUD.Art.Media');
INSERT INTO tud VALUES ('TUD.Art.Media.GameDesign');
INSERT INTO tud VALUES ('TUD.Art.Media.FilmBroadcasting');
INSERT INTO tud VALUES ('TUD.Art.Conservatory');
INSERT INTO tud VALUES ('TUD.Art.Conservatory.Composition');
INSERT INTO tud VALUES ('TUD.Business');
INSERT INTO tud VALUES ('TUD.Business.Accounting_Finance');
INSERT INTO tud VALUES ('TUD.Business.Accounting_Finance.BusinessFinance');
INSERT INTO tud VALUES ('TUD.Business.Accounting_Finance.Macroeconomics');
INSERT INTO tud VALUES ('TUD.Business.Marketing');
INSERT INTO tud VALUES ('TUD.Business.Marketing.MarketingPractice');
INSERT INTO tud VALUES ('TUD.Business.Marketing.InternationalMarketing');
INSERT INTO tud VALUES ('TUD.Business.Management');
INSERT INTO tud VALUES ('TUD.Business.Management.Logistics');
INSERT INTO tud VALUES ('TUD.Business.Management.SupplyChainManagement');
INSERT INTO tud VALUES ('TUD.Business.Retail');
```

```sql
INSERT INTO tud VALUES ('TUD.Business.Retail.RetailAnalytics');

INSERT INTO tud VALUES ('TUD.Business.Retail.RetailMarketing');

INSERT INTO tud VALUES ('TUD.Science');

INSERT INTO tud VALUES ('TUD.Science.Pharmaceutical_Chemistry_Forensic.Biotechnology');

INSERT INTO tud VALUES ('TUD.Science.Pharmaceutical_Chemistry_Forensic.BasicMicrobiology');

INSERT INTO tud VALUES ('TUD.Science.BiologicalScience');

INSERT INTO tud VALUES ('TUD.Science.BiologicalScience.MolecularBiology');

INSERT INTO tud VALUES ('TUD.Science.BiologicalScience.ScientificProject');

INSERT INTO tud VALUES ('TUD.Science.Food_Nutrition');

INSERT INTO tud VALUES ('TUD.Science.Food_Nutrition.FoodMicrobiology');

INSERT INTO tud VALUES ('TUD.Science.Food_Nutrition.Food_Beverage');

INSERT INTO tud VALUES ('TUD.Science.HealthCare');

INSERT INTO tud VALUES ('TUD.Science.HealthCare.OpticalDispensing');

INSERT INTO tud VALUES ('TUD.Science.HealthCare.BinocularVision');

INSERT INTO tud VALUES ('TUD.Science.Physics');

INSERT INTO tud VALUES ('TUD.Science.Physics.Relativity');

INSERT INTO tud VALUES ('TUD.Science.Physics.QuantumMechanics');

INSERT INTO tud VALUES ('TUD.Science.Mathematics');

INSERT INTO tud VALUES ('TUD.Science.Mathematics.Geometry');

INSERT INTO tud VALUES ('TUD.Science.Mathematics.RealAnalisys');

INSERT INTO tud VALUES ('TUD.Science.Computing');

INSERT INTO tud VALUES ('TUD.Science.Computing.Databases');

INSERT INTO tud VALUES ('TUD.Science.Computing.CloudComputing');

INSERT INTO tud VALUES ('TUD.Engineering');

INSERT INTO tud VALUES ('TUD.Engineering.BuiltEngineering.AutomitiveManagement');

INSERT INTO tud VALUES ('TUD.Engineering.BuiltEngineering.PropertyStudies');

INSERT INTO tud VALUES ('TUD.Engineering.Architecture');

INSERT INTO tud VALUES ('TUD.Engineering.Architecture.ArchitecturalTechnology');

INSERT INTO tud VALUES ('TUD.Engineering.Architecture.ConstrutionStieManagement');

INSERT INTO tud VALUES ('TUD.Engineering.StructuralEngineering');

INSERT INTO tud VALUES ('TUD.Engineering.StructuralEngineering.CivilEngineering');

INSERT INTO tud VALUES ('TUD.Engineering.MultidisciplinaryTechnology');

INSERT INTO tud VALUES ('TUD.Engineering.MultidisciplinaryTechnology.Modelling');

INSERT INTO tud VALUES ('TUD.Engineering.MultidisciplinaryTechnology.BIM');

INSERT INTO tud VALUES ('TUD.Engineering.ElectronicalEngineering');

INSERT INTO tud VALUES ('TUD.Engineering.ElectronicalEngineering.ElectricalPower');

INSERT INTO tud VALUES ('TUD.Engineering.ElectronicalEngineering.Control');
```

```sql
INSERT INTO tud VALUES ('TUD.Engineering.Transport');

INSERT INTO tud VALUES ('TUD.Engineering.Transport.SpatialPlanning');

INSERT INTO tud VALUES ('TUD.Engineering.Transport.LocalDevelopment');

INSERT INTO tud VALUES ('TUD.Engineering.Design');

INSERT INTO tud VALUES ('TUD.Engineering.Design.DesignInnovation');

INSERT INTO tud VALUES ('TUD.Engineering.Design.CreativeDesignStudio');

INSERT INTO tud VALUES ('TUD.Engineering.Pippo.pdf');


-- Create indexes for faster querying

CREATE INDEX path_gist_idx ON tud USING gist(path);

CREATE INDEX path_idx ON tud USING btree(path);


-- Show all paths from the 4th node of length 4, descendant of 'TUD.ART' (=school of art)

SELECT subpath(path, 3) FROM tud WHERE path <@ 'TUD.Art' and nlevel(path)=4;


-- Select the root node

SELECT subpath(path,0,1) FROM tud where nlevel(path)=1;


-- Count the number of schools for each college

SELECT subpath(path,1,1) AS "College",count(subpath(path,2,1)) AS "Number of Schools"

FROM tud

WHERE nlevel(path) = 3

GROUP BY subpath(path,1,1);


-- Add a new school named Computer_Science under the TUD.Science college

INSERT INTO tud (path) VALUES ('TUD.Science.Computer_Science');


-- Add two new degrees to the Computer_Science school: software and AI

INSERT INTO tud (path) VALUES ('TUD.Science.Computer_Science.software');

INSERT INTO tud VALUES ('TUD.Science.Computer_Science.AI');


-- a) Find under which degree the MolecularBiology degree is

SELECT subpath(path, 3)

FROM tud

WHERE path <@ 'TUD.Science.BiologicalScience.MolecularBiology';


-- b) Find how many courses are listed in the TUD dataset
```

```sql
SELECT COUNT(*)
FROM tud
WHERE nlevel(path) = 4;


-- c) Find which faculty has the highest number of courses
SELECT subpath(path, 2, 1) AS "Faculty", COUNT(*) AS "Number of Courses"
FROM tud
WHERE nlevel(path) = 4
GROUP BY subpath(path, 2, 1)
ORDER BY COUNT(*) DESC
LIMIT 1;


-- d) How many colleges are in the TUD dataset?
SELECT COUNT(DISTINCT subpath(path, 1, 1))
FROM tud
WHERE nlevel(path) = 2;


-- e) Rename the university TUDublin (all the paths containing the label TUD must be replaced by the new label TUDublin)
UPDATE tud
SET path = REGEXP_REPLACE(path::text, '^TUD', 'TUDublin')::ltree;


-- f) Delete the school of BiologicalScience with all its courses
DELETE FROM tud
WHERE path <@ 'TUDublin.Science.BiologicalScience';


-- g) Add a column CAO points to the TUD table (integer)
ALTER TABLE tud ADD COLUMN cao_points INTEGER;


-- h) Add 300 CAO points to all the degrees under the Art College, 450 to the degrees under the Science College,
-- 400 to the Engineering College, and the remaining assign 350
UPDATE tud SET cao_points = 300
WHERE path <@ 'TUDublin.Art' AND nlevel(path) = 4;


UPDATE tud SET cao_points = 450
WHERE path <@ 'TUDublin.Science' AND nlevel(path) = 4;
```

```sql
UPDATE tud SET cao_points = 400

WHERE path <@ 'TUDublin.Engineering' AND nlevel(path) = 4;


UPDATE tud SET cao_points = 350

WHERE nlevel(path) = 4 AND cao_points IS NULL;


-- Assign 500 CAO points to the degrees in the School of Computer Science

UPDATE tud SET cao_points = 500

WHERE path <@ 'TUDublin.Science.Computer_Science' AND nlevel(path) = 4;


SELECT AVG(cao_points) AS average_cao_points

FROM tud

WHERE path <@ 'TUDublin.Science'

AND nlevel(path) = 4

AND cao_points IS NOT NULL;
```

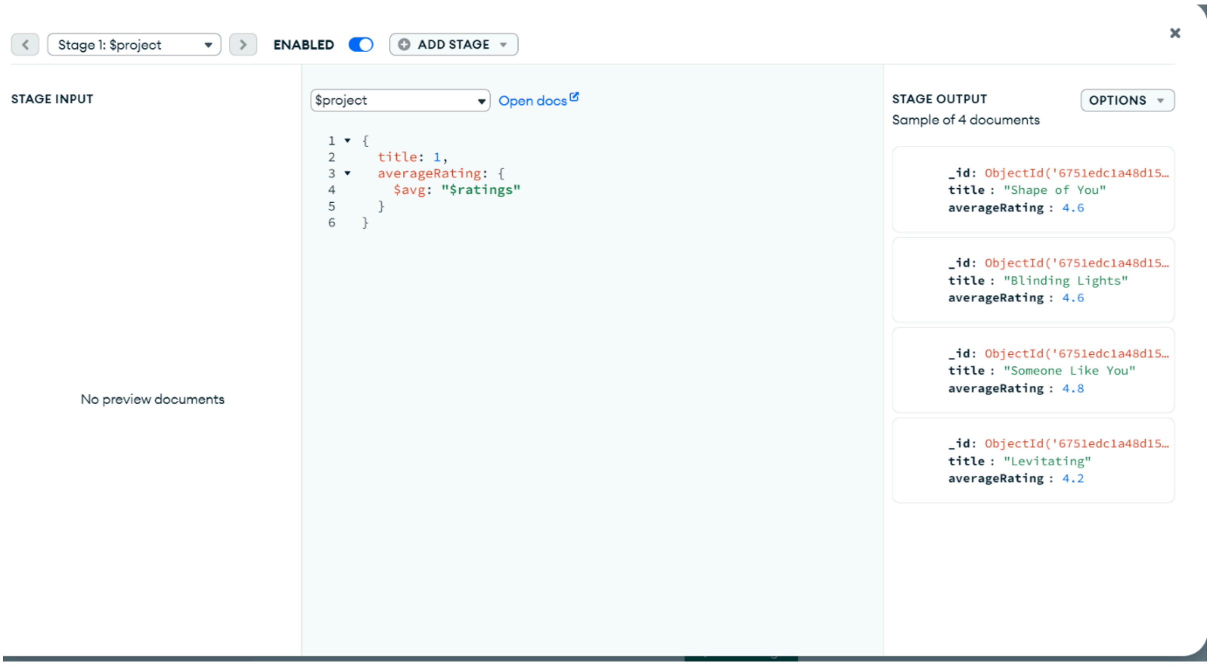
This SQL script demonstrates the use of PostgreSQL's ltree extension to manage hierarchical data, such as an organizational structure or curriculum. It begins by creating the ltree extension and defining a table with a path column to represent hierarchical paths. Data is inserted to create a tree structure, with nodes representing colleges, schools, and courses, using . as a delimiter. Indexes (GiST and B-tree) are added to optimize path-based and equality-based queries. Queries extract specific nodes, count entities at different levels, and perform hierarchical operations such as subtree selection and deletion. The script dynamically updates the data, such as renaming paths (e.g., replacing TUD with TUDublin) and adding or removing subtrees. A new column for CAO points is added, with conditional updates assigning points to specific colleges and schools, followed by computing averages for analytical purposes. The use of ltree functions like subpath, <@, and nlevel enables efficient traversal and manipulation of the hierarchy, while the combination of dynamic updates, indexing, and structured queries highlights its capability to manage and analyze complex hierarchical datasets effectively.

Lab 8

1.1

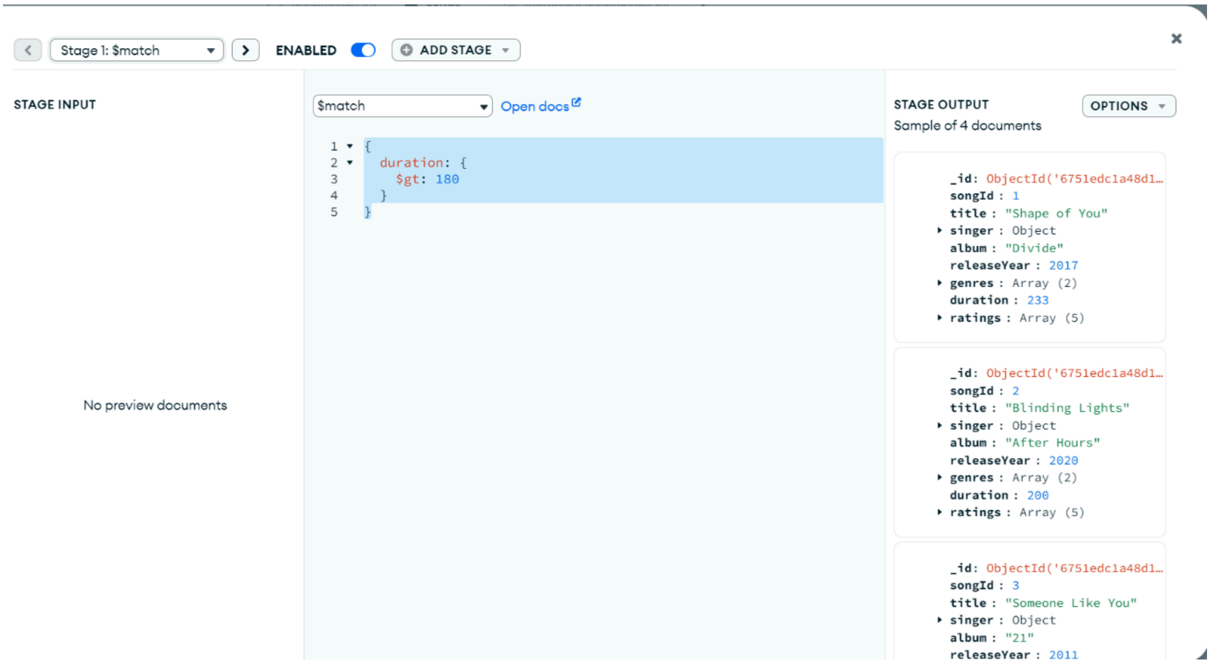
Calculate Average Rating for Each Song.

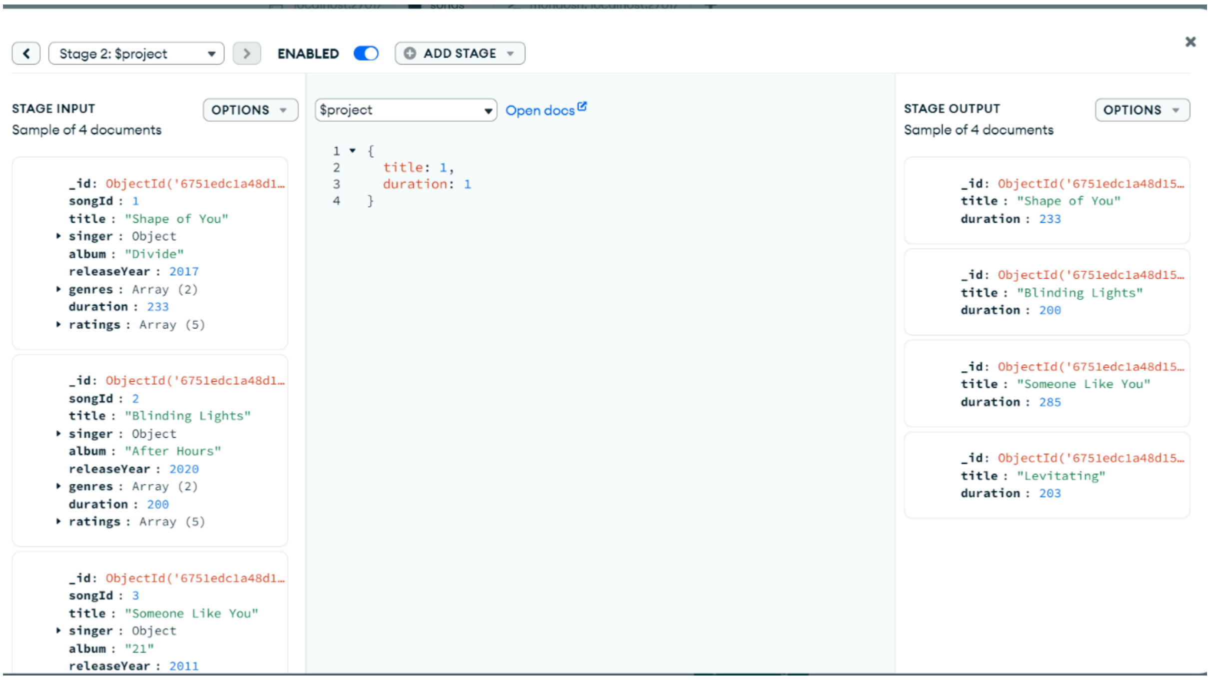
This code snippet is a MongoDB aggregation stage that groups documents by their title field (title: 1) and calculates the average of the ratings field for each group using $avg. The "$ratings" reference indicates the field whose values are averaged within each group. This operation is part of a larger aggregation pipeline and is used to process data such as computing average ratings for distinct titles in a collection.



1.2

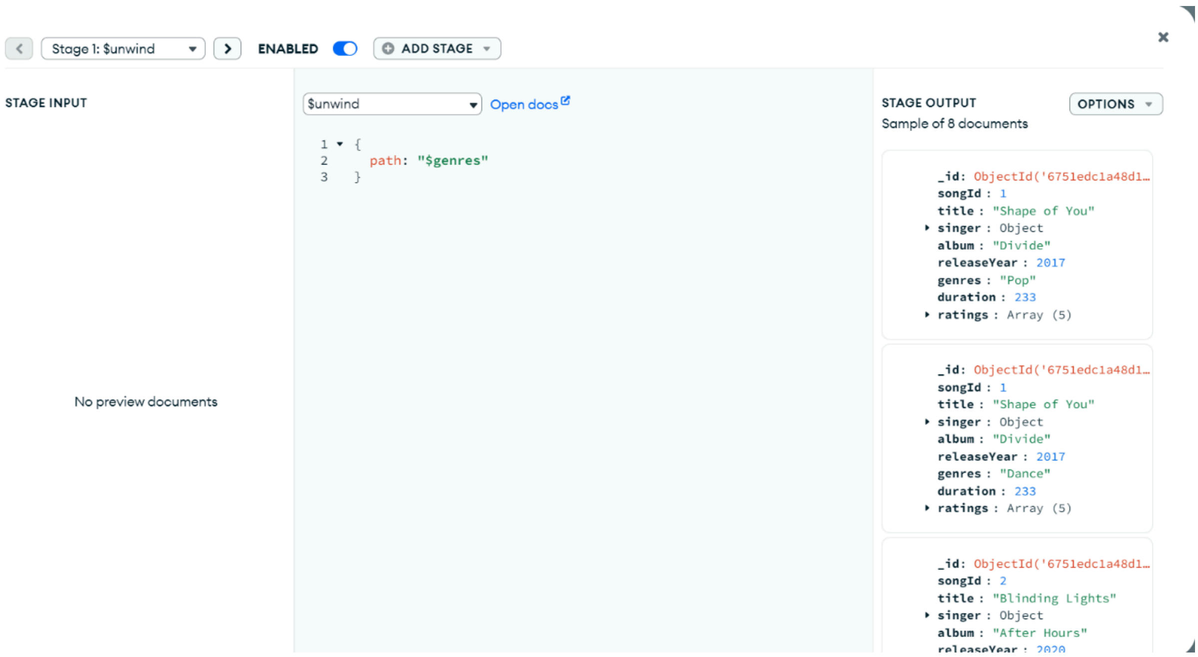
This code snippet is part of a MongoDB query that specifies a condition for the duration field. The condition { duration: { $gt: 180 } } means that only documents where the duration field is greater than 180 will be included in the results. So, combined with the previous part where title: 1 and duration: 1 are specified, this query will return only the title and duration of documents where the duration is greater than 180.

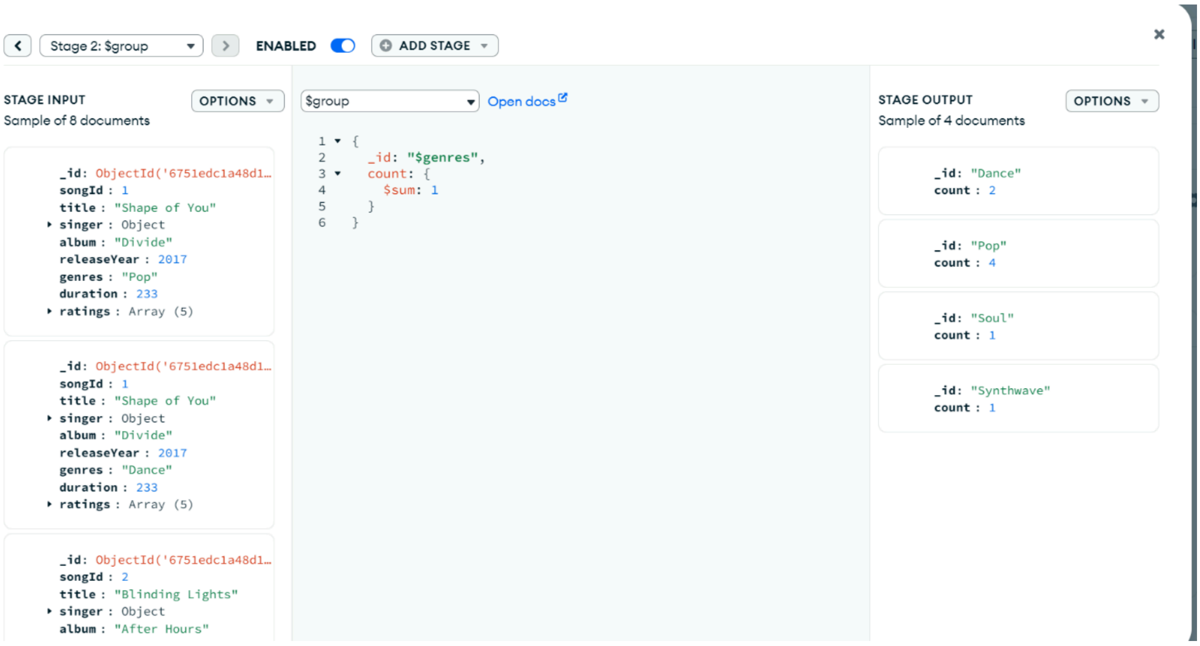




1.3

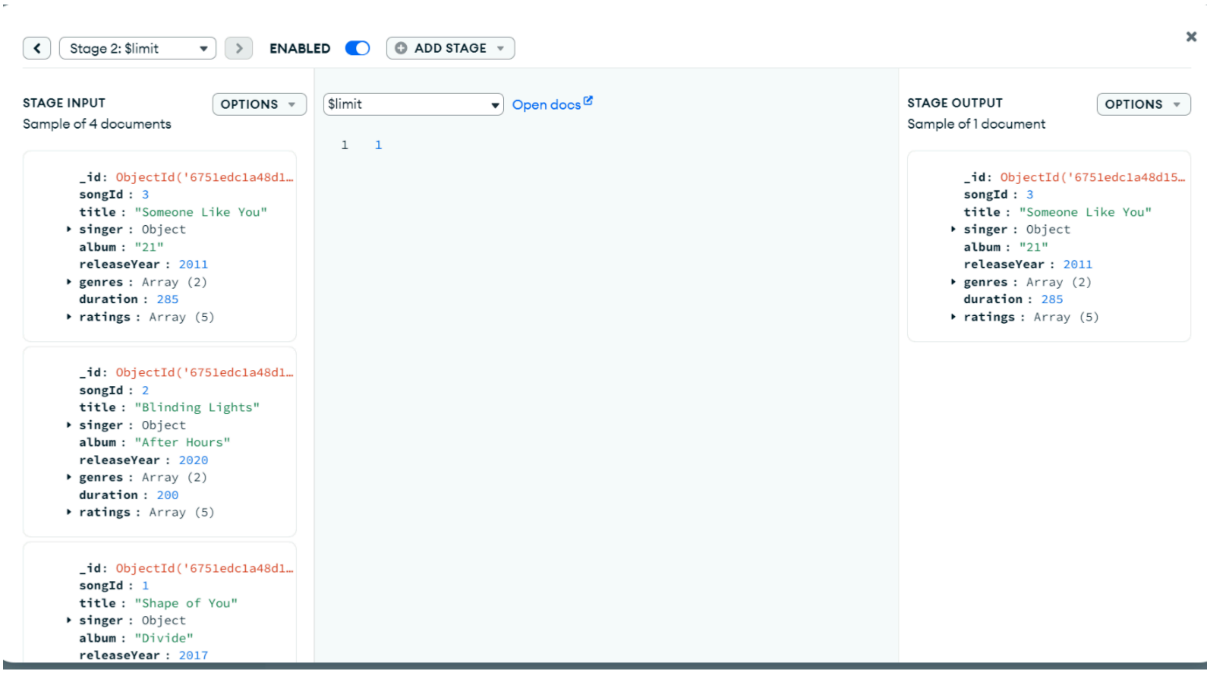
This code snippet is part of a MongoDB aggregation pipeline that processes documents based on the genres field. The first part, { path: "$genres" }, refers to referencing the genres field in each document for further operations. The second part, { _id: "$genres", count: { $sum: 1 } }, groups the documents by the values in the genres field and counts how many times each genre appears in the collection. The _id: "$genres" groups the documents by their genre, while count: { $sum: 1 } counts the number of occurrences of each genre.

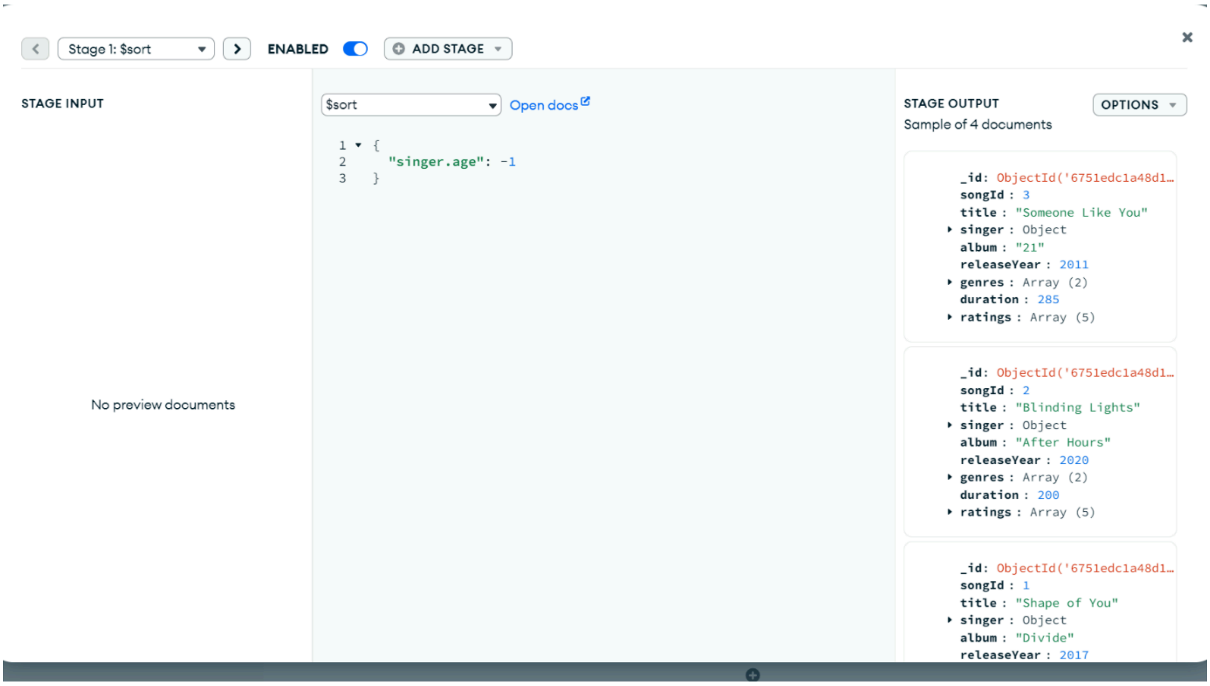




1.4

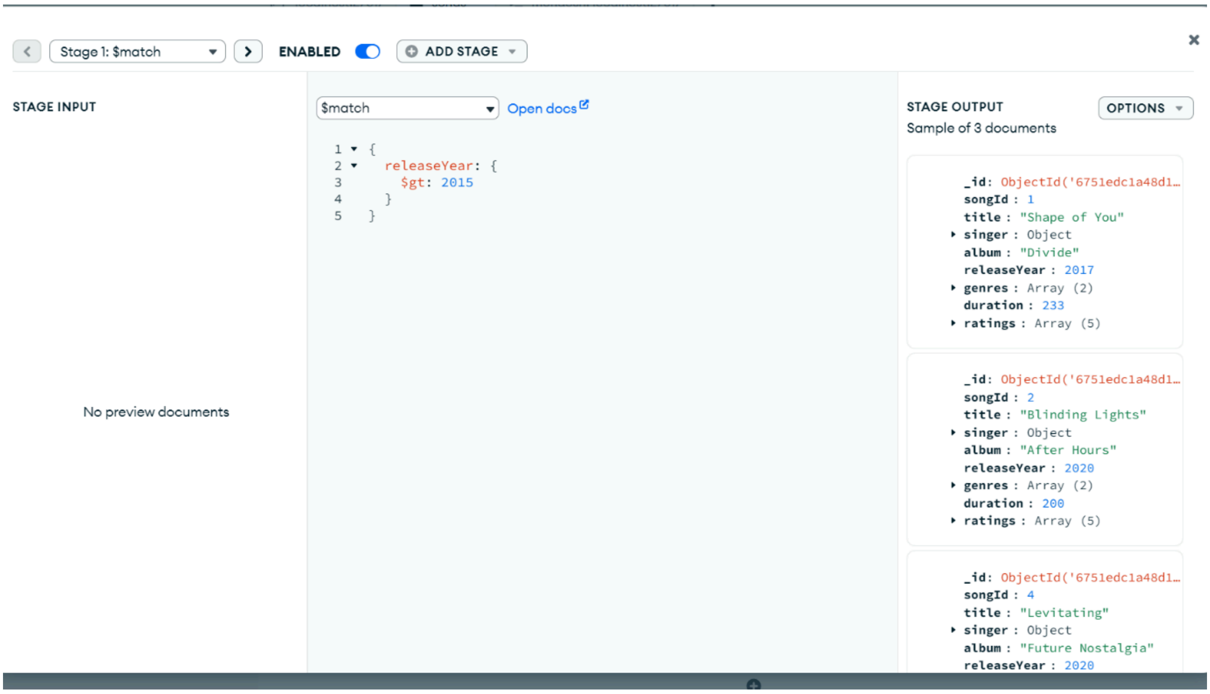
The code snippet { "singer.age": -1 } specifies how to sort the results based on the age field of the singer subdocument. The -1 indicates that the sorting should be in descending order, meaning the documents will be arranged from the oldest to the youngest singer.

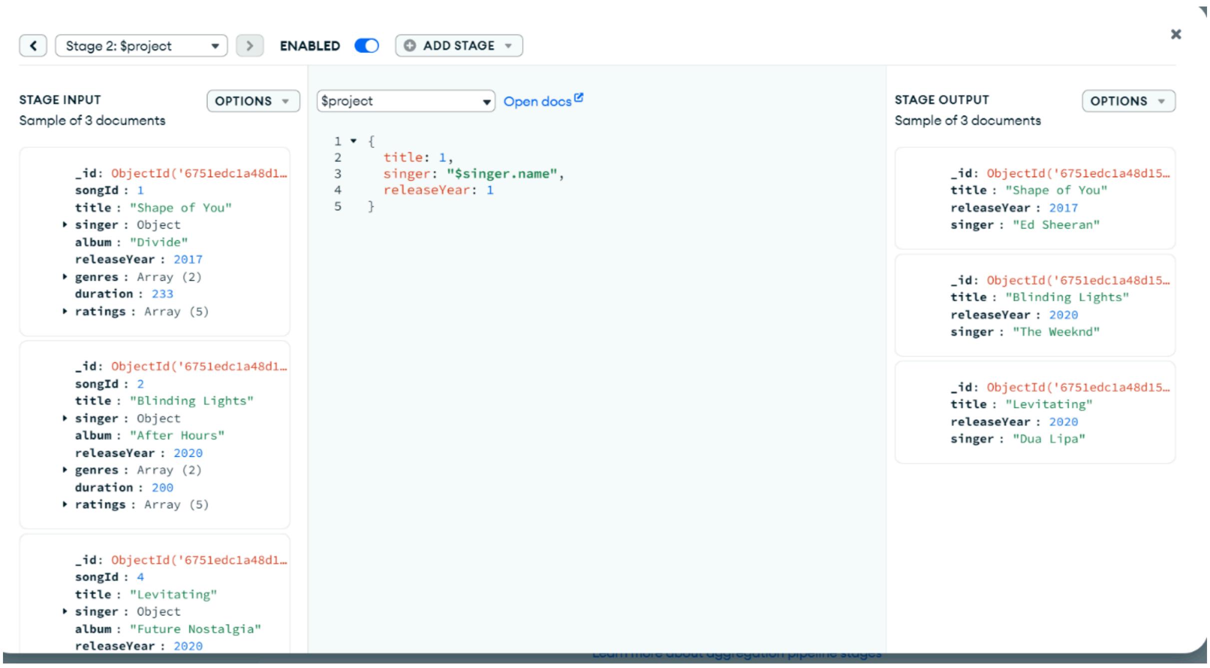




1.5

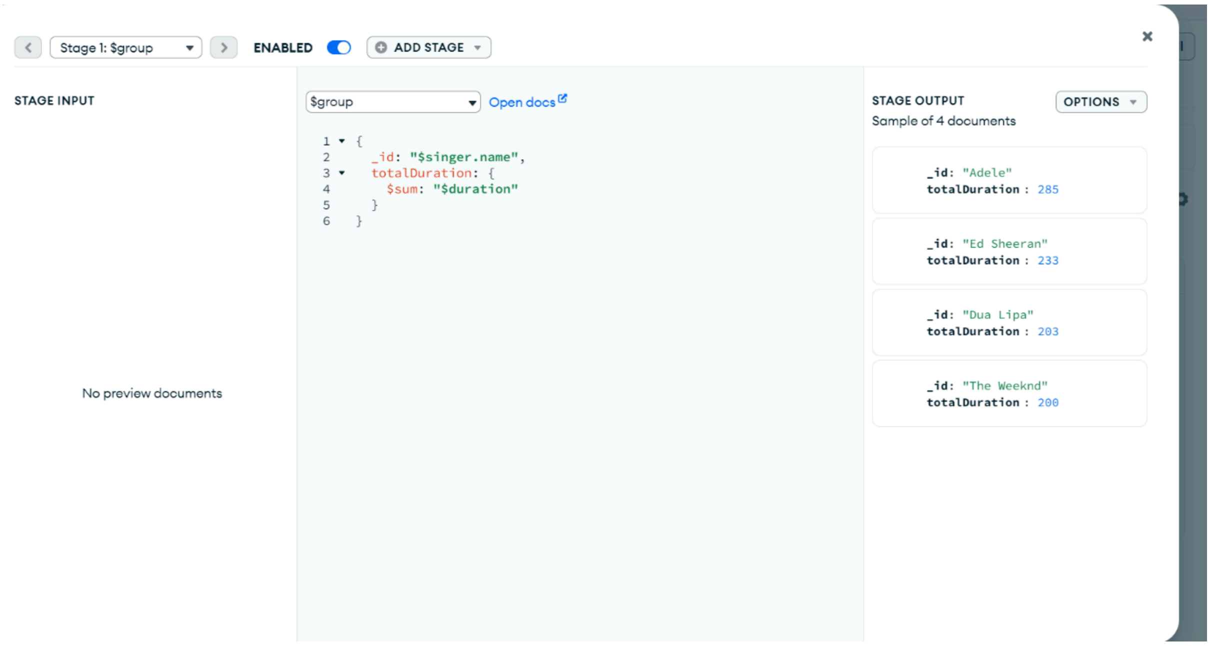
This code snippet performs two actions. The first part { releaseYear: { $gt: 2015 } } filters the documents to only include those where the releaseYear is greater than 2015, effectively excluding older songs or items. The second part { title: 1, singer: "$singer.name", releaseYear: 1 } selects the title and releaseYear fields directly.





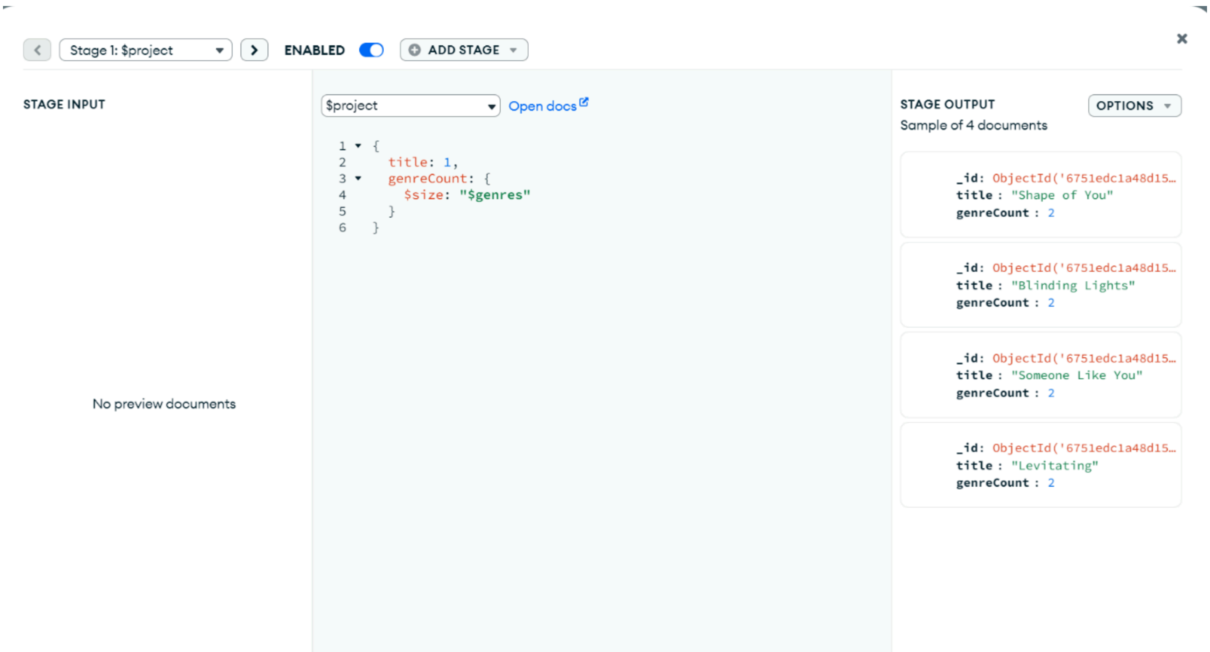
1.6

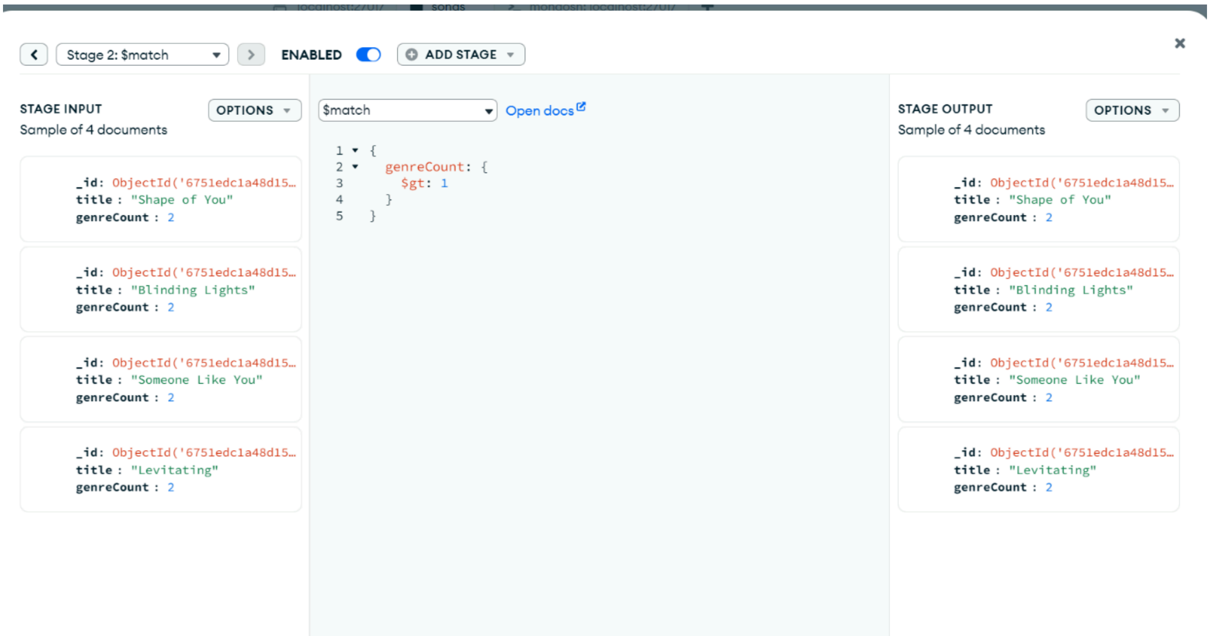
This code snippet groups documents based on the singer.name field and calculates the total duration of songs for each singer. The _id: "$singer.name" groups the documents by the singer's name, so each group represents a unique singer. The totalDuration: { $sum: "$duration" } part sums up the duration field for all songs by each singer, giving the total duration of all their songs combined.



1.7

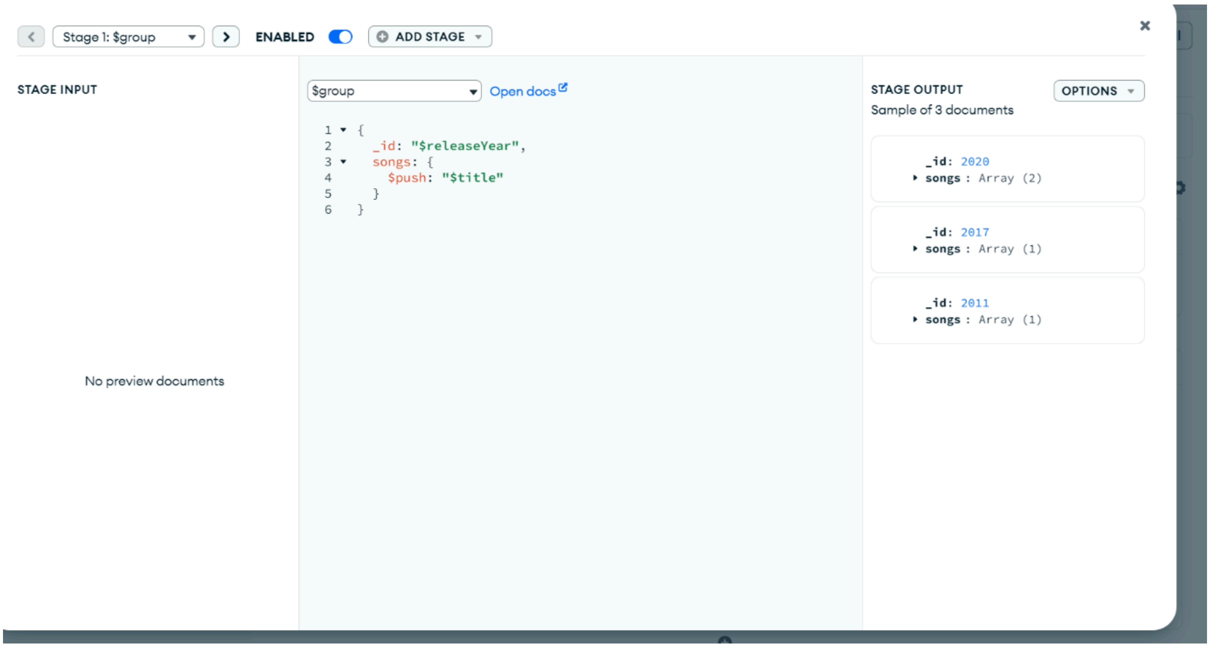
This code snippet calculates how many genres are associated with each document and then filters those based on the count. The first part { title: 1, genreCount: { $size: "$genres" } } projects the title of each document and calculates the number of genres listed in the genres array for each document using the $size operator. This results in a field called genreCount that holds the count of genres for each document. The second part { genreCount: { $gt: 1 } } filters the results to only include documents where the genreCount is greater than 1, meaning it will return documents where there are more than one genre associated.



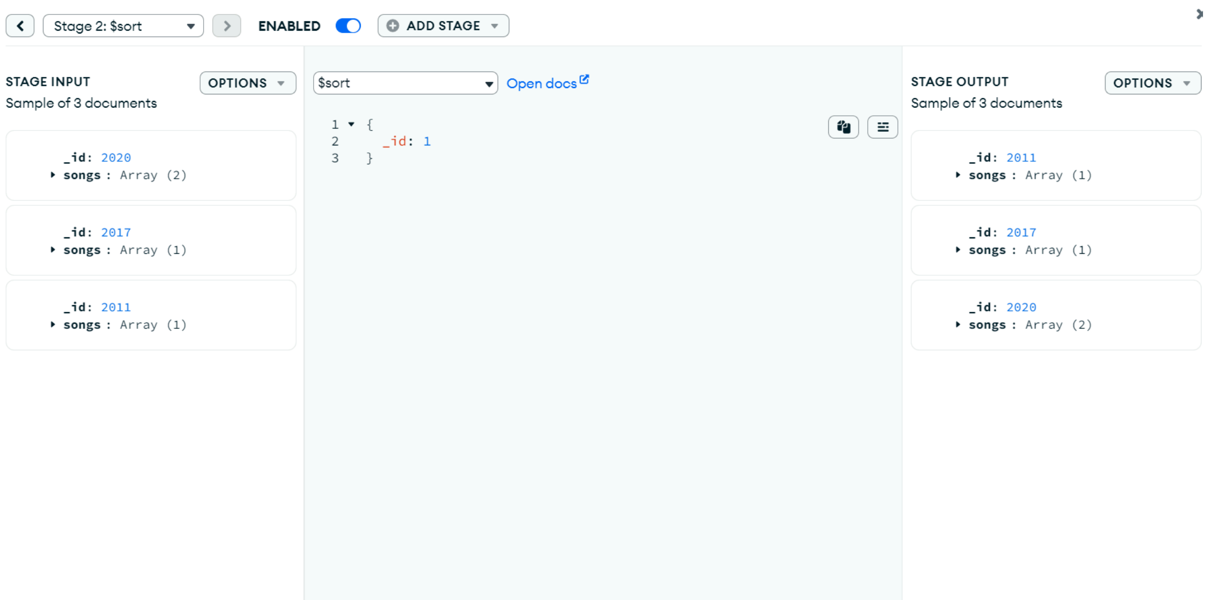


1.8

This code snippet groups documents by the releaseYear field and collects the titles of the songs within each group. The first part { _id: "$releaseYear", songs: { $push: "$title" } } groups the documents by the releaseYear and uses the $push operator to create an array of title values for each year. This means for each unique release year, a list of song titles will be included in the output. The second part { _id: 1 } specifies that only the grouped releaseYear (as the _id) should be included in the output, along with the list of titles under the songs field

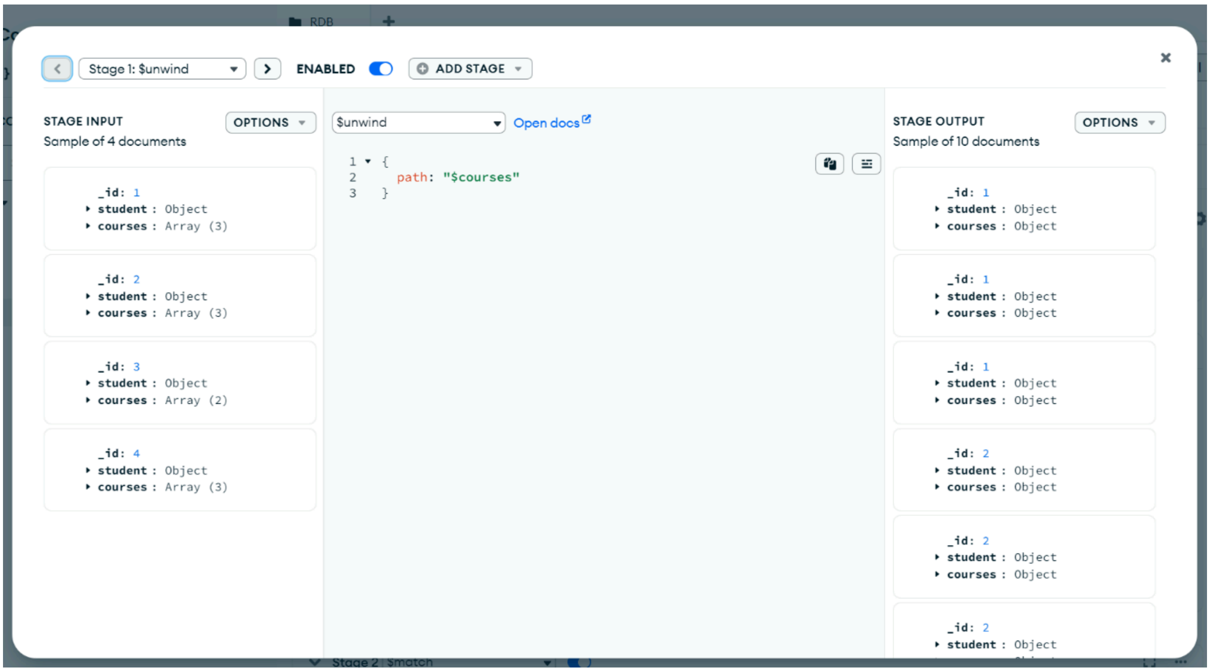


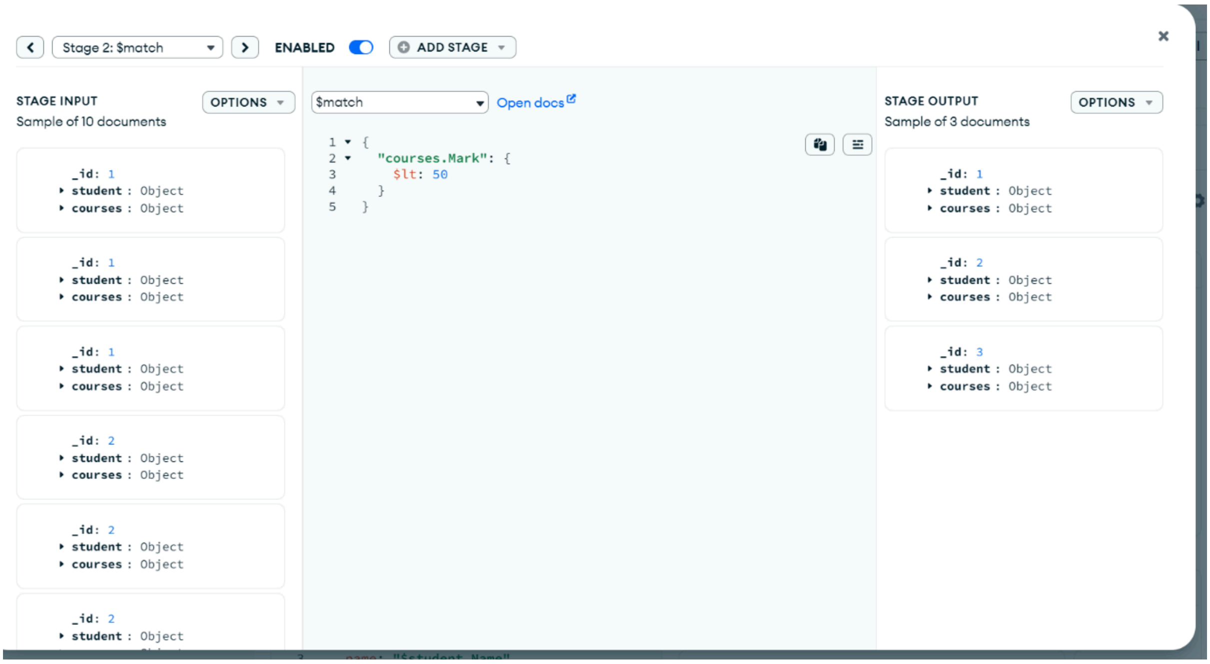
.

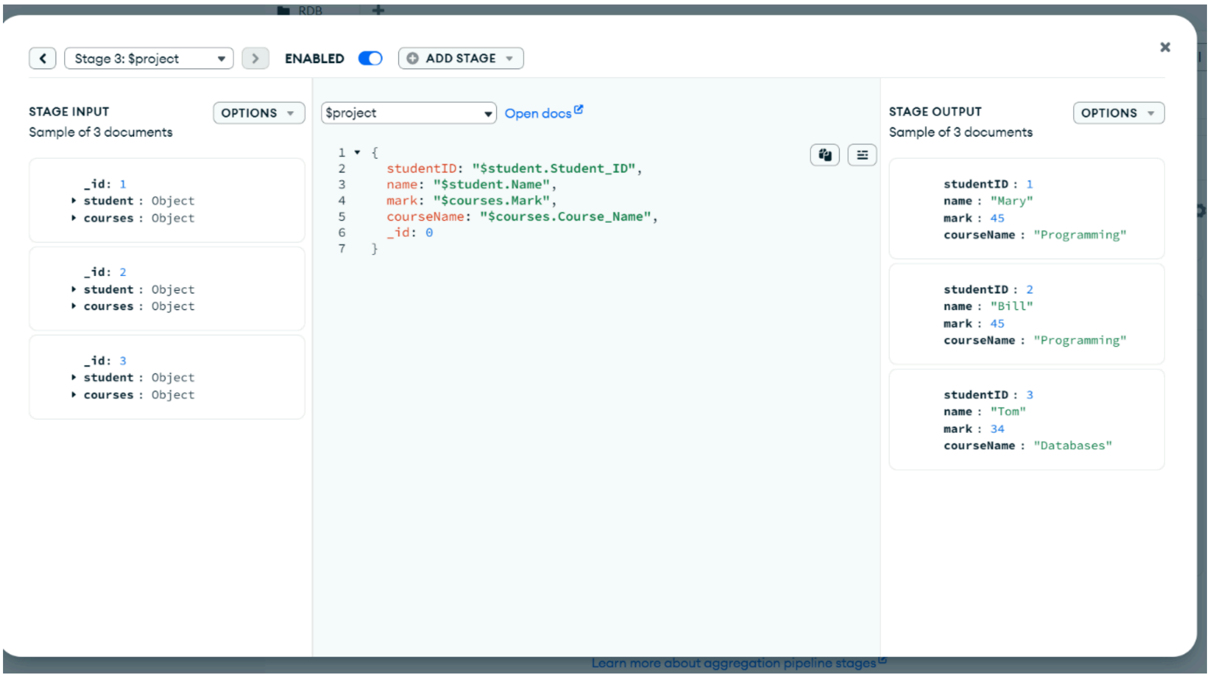


2.1

This code snippet is designed to find students who scored less than 50 in any of their courses. The first part { path: "$courses" } indicates that the query is working with the courses array.The second part { "courses.Mark": { $lt: 50 } } filters the documents to only include those where the Mark field in the courses array is less than 50, identifying students who performed poorly in at least one course. The third part projects specific fields to be included in the output: it retrieves the studentID and name from the student subdocument, as well as the Mark and Course_Name from the courses array. The _id: 0 ensures that the MongoDB default _id field is excluded from the results. As a result, the query will return a list of students, their course names, and the marks they received in courses where they scored below 50.

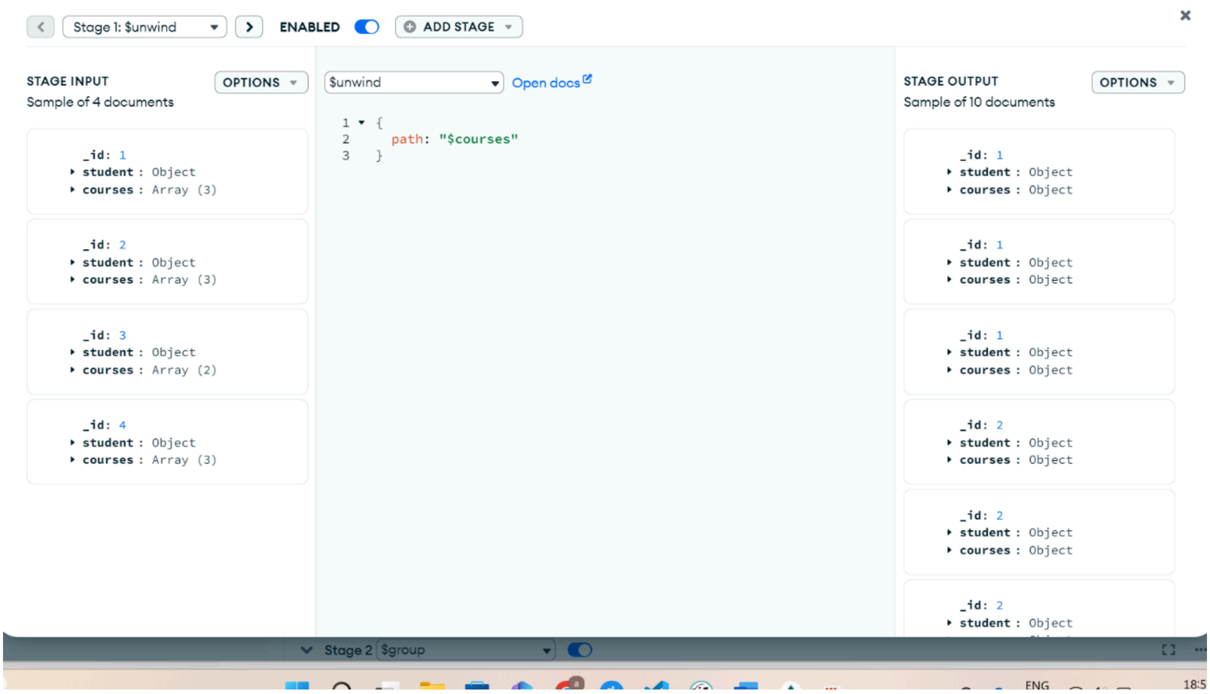


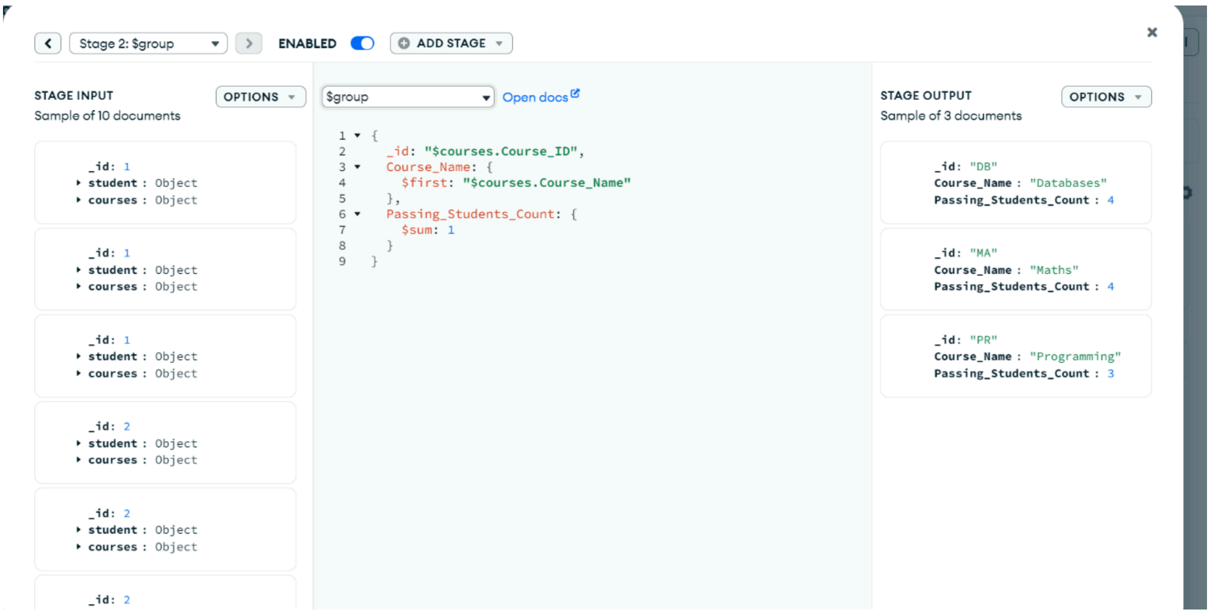




2.2

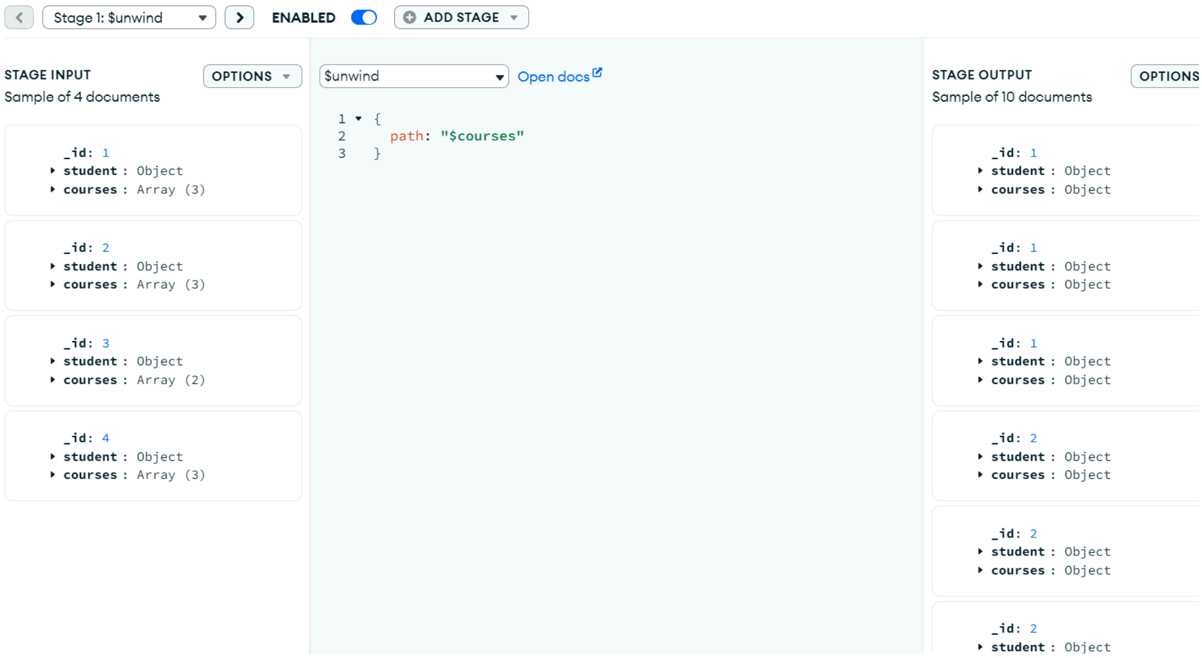
This code is used to group and count the number of students passing each course. The first part { path: "$courses" } indicates that the operation will be performed on the courses array within each document. The second part of the pipeline groups the courses by their Course_ID, using the _id field to store the unique Course_ID for each group. It also uses the $first operator to retrieve the Course_Name for each course, ensuring that the first occurrence of Course_Name in the courses array is included in the result. Additionally, the Passing_Students_Count field is calculated using $sum: 1, which increments by 1 for each student in the array, effectively counting the number of students enrolled in each course. The result will show each course's Course_ID, Course_Name, and the count of students associated with that course.





2.3

This code processes data to calculate the average mark for each student across all their courses. The first part { path: "$courses" } indicates that the operation is being performed on the courses array, which contains the courses for each student. The pipeline then groups the data by Student_ID (using the _id field) to ensure that the calculations are done per student. It also retrieves the Name and Surname of the student using the $first operator, which takes the first value of each field for the grouped student. The AverageMark is calculated using the $avg operator, which computes the average of the Mark field from the courses array for each student. As a result, the output will show each student's Student_ID, Name, Surname, and their average mark across all enrolled courses.


Stage 1: $unwind
ENABLED
ADD STAGE
STAGE INPUT
OPTIONS
Sample of 4 documents
_id: 1
student : Object
courses : Array (3)
_id: 2
student : Object
courses : Array (3)
_id: 3
student : Object
courses : Array (2)
_id: 4
student : Object
courses : Array (3)
$unwind
Open docs
1 {
2 path: "$courses"
3 }
STAGE OUTPUT
OPTIONS
Sample of 10 documents
_id: 1
student : Object
courses : Object
_id: 1
student : Object
courses : Object
_id: 1
student : Object
courses : Object
_id: 2
student : Object
courses : Object
_id: 2
student : Object
courses : Object
_id: 2
student : Object


Stage 2: $group
ENABLED
ADD STAGE
STAGE INPUT
OPTIONS
Sample of 10 documents
_id: 1
student : Object
courses : Object
_id: 1
student : Object
courses : Object
_id: 1
student : Object
courses : Object
_id: 2
student : Object
courses : Object
_id: 2
student : Object
courses : Object
_id: 2
$group
Open docs
1 {
2 _id: "$student.Student_ID",
3 Name: {
4 $first: "$student.Name"
5 },
6 Surname: {
7 $first: "$student.Surname"
8 },
9 AverageMark: {
10 $avg: "$courses.Mark"
11 }
12 }
STAGE OUTPUT
OPTIONS
Sample of 4 documents
_id: 4
Name : "John"
Surname : "Bale"
AverageMark : 84.666666666666…
_id: 3
Name : "Tom"
Surname : "Brady"
AverageMark : 45
_id: 2
Name : "Bill"
Surname : "Bellichick"
AverageMark : 62.333333333333…
_id: 1
Name : "Mary"
Surname : "Murray"
AverageMark : 59

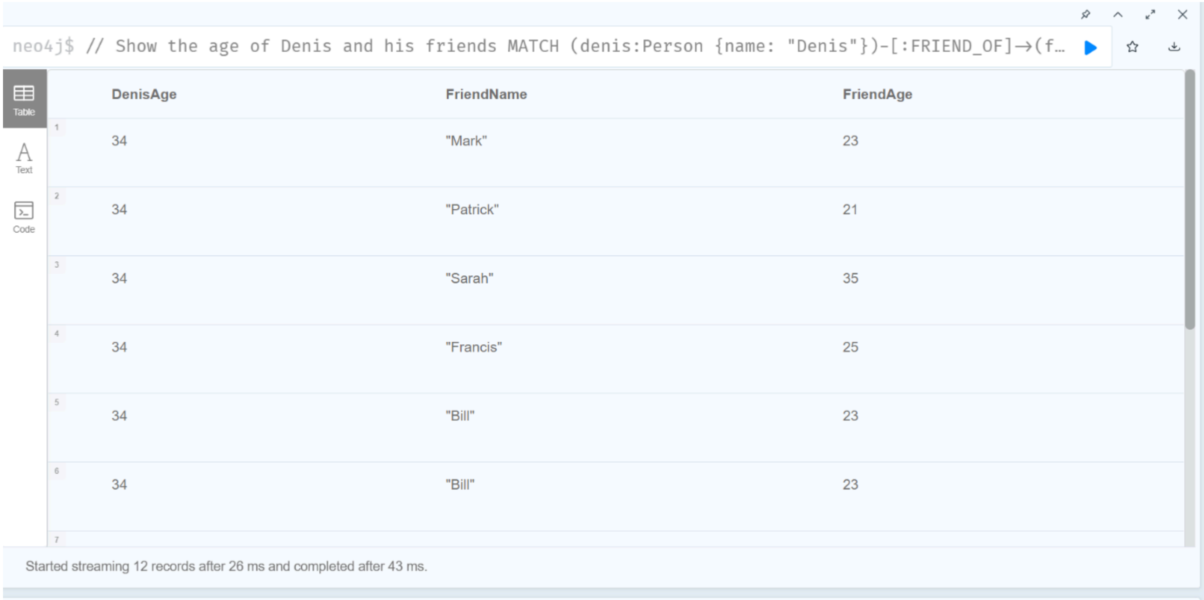
Lab 9

Q1 Show the age of Denis and his friends

MATCH (denis:Person {name: "Denis"})-[:FRIEND_OF]->(friend)

RETURN denis.age AS DenisAge, friend.name AS FriendName, friend.age AS FriendAge;

This query matches Denis (Person {name: "Denis"}) and finds all of his direct friends connected through the FRIEND_OF relationship. The RETURN statement selects Denis's age and the names and ages of his friends by traversing the relationships from Denis to each connected friend. It uses the RETURN keyword to structure the results in a readable format with aliases for the properties (DenisAge, FriendName, FriendAge).



	DenisAge	FriendName	FriendAge
1	34	"Mark"	23
2	34	"Patrick"	21
3	34	"Sarah"	35
4	34	"Francis"	25
5	34	"Bill"	23
6	34	"Bill"	23

Q2 Show all the person from Scotland

MATCH (person:Person {country: "Scotland"})

RETURN person.name AS Name, person.age AS Age, person.sport AS Sport;

This query uses a MATCH clause to find all persons in the Person label who have the country attribute set to "Scotland". It then returns a set of properties for each person, specifically the name, age, and sport attributes. The WHERE clause is not needed since all entries are filtered by the country attribute directly.

neo4j$ // Show all the person from Scotland MATCH (person:Person {country: "Scotland"}) RETURN person…

	Name	Age	Sport
1	"Julia"	28	"Football"
2	"Lisa"	25	"Football"
3	"Bart"	25	"Rugby"
4	"Denis"	34	"Football"
5	"Julia"	28	"Football"
6	"Lisa"	25	"Football"
7			

Started streaming 8 records after 32 ms and completed after 36 ms.

Q3 Show all the person with age less or equal than 20 from Ireland

MATCH (person:Person {country: "Ireland"})

WHERE person.age <= 20

RETURN person.name AS Name, person.age AS Age;

This query matches all Person nodes from Ireland with an age less than or equal to 20. The WHERE clause filters the results to include only those whose age is 20 or under. The RETURN statement selects the name and age of the person who meets these criteria.

neo4j$ // Show all the person with age less or equal than 20 from Ireland MATCH (person:Person {count…

	Name	Age
1	"Emily"	19
2	"Kevin"	17
3	"Emily"	19
4	"Kevin"	17

Started streaming 4 records after 40 ms and completed after 43 ms.

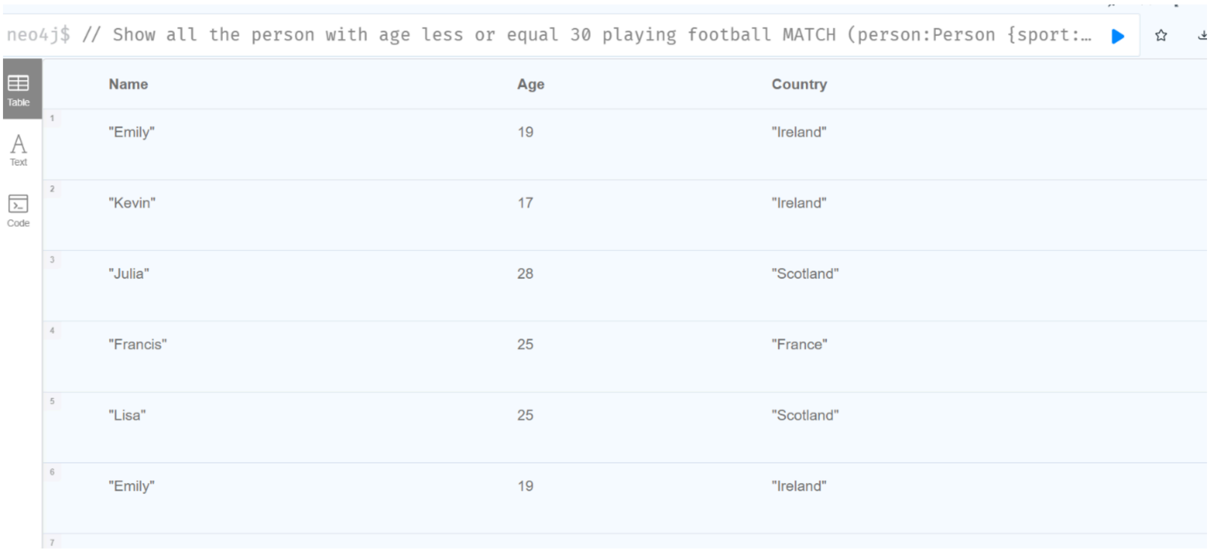
Q4 Show all the person with age less or equal 30 playing football

MATCH (person:Person {sport: "Football"})

WHERE person.age <= 30

RETURN person.name AS Name, person.age AS Age, person.country AS Country;

This query retrieves all persons who play football and have an age of 30 or less. The MATCH clause finds all Person nodes with sport: "Football", and the WHERE clause restricts the results by filtering on age <= 30. It returns the person's name, age, and country.



neo4j$ // Show all the person with age less or equal 30 playing football MATCH (person:Person {sport:…

	Name	Age	Country
1	"Emily"	19	"Ireland"
2	"Kevin"	17	"Ireland"
3	"Julia"	28	"Scotland"
4	"Francis"	25	"France"
5	"Lisa"	25	"Scotland"
6	"Emily"	19	"Ireland"
7			

Q5 Count the person by country

MATCH (person:Person)

RETURN person.country AS Country, COUNT(person) AS Count;

This query counts the number of Person nodes per country. The MATCH clause selects all Person nodes, and the RETURN statement uses the COUNT function to group the results by the country attribute. It returns the country and the corresponding count of people in that country.

neo4j$ MATCH (person:Person) RETURN person.country AS Country, COUNT(person) AS Count;

	Country	Count
1	"Sweden"	10
2	"Ireland"	9
3	"France"	6
4	"Scotland"	8
5	"Spain"	1

Started streaming 5 records after 15 ms and completed after 18 ms.

Q6 Show the average age of the person group by sport

MATCH (person:Person)

RETURN person.sport AS Sport, AVG(person.age) AS AverageAge;

This query calculates the average age of persons, grouped by their sport. The MATCH clause selects all Person nodes, and the RETURN statement uses the AVG aggregation function to calculate the average age for each sport. The results are grouped by the sport attribute, showing the average age for each group.

neo4j$ // Show the average age of the person group by sport MATCH (person:Person) RETURN person.sport…

	Sport	AverageAge
1	"Hockey"	28.666666666666664
2	"Football"	26.823529411764703
3	"Rugby"	23.2
4	null	23.0

Started streaming 4 records after 25 ms and completed after 33 ms.

Q7 Show all the direct friends of Mary

MATCH (mary:Person {name: "Mary"})-[:FRIEND_OF]->(friend)

RETURN friend.name AS FriendName, friend.age AS FriendAge, friend.country AS Country;

This query matches the Person node labeled as "Mary" and finds all her direct friends via the FRIEND_OF relationship. It uses the MATCH clause to navigate from Mary to her connected friends. The RETURN statement fetches the friends' name, age, and country attributes.

neo4j$ // Show all the direct friends of Mary MATCH (mary:Person {name: "Mary"})-[:FRIEND_OF]→(frien…

	FriendName	FriendAge	Country
1	"Emily"	19	"Ireland"
2	"Mark"	23	"Sweden"
3	"Joe"	32	"Sweden"
4	"John"	31	"Ireland"
5	"Peter"	23	"France"
6	"Paul"	25	"Sweden"

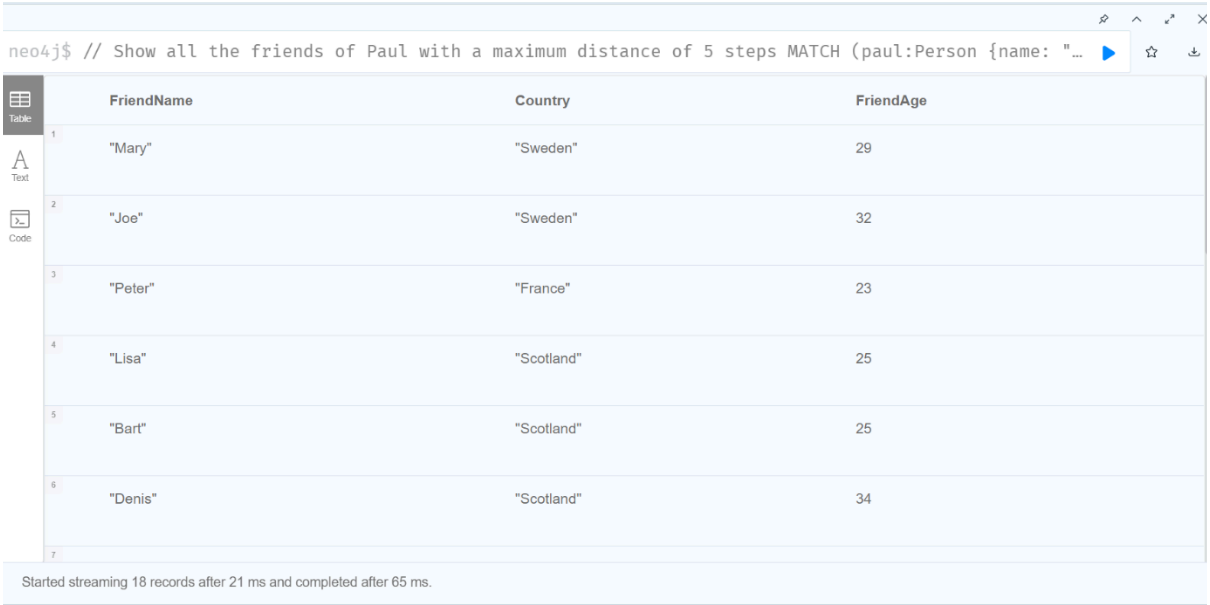
Started streaming 18 records after 32 ms and completed after 32 ms.

Q8 Show all the friends of Paul with a maximum distance of 5 steps

MATCH (paul:Person {name: "Paul"})-[:FRIEND_OF*..5]->(friend)

RETURN DISTINCT friend.name AS FriendName, friend.country AS Country, friend.age AS FriendAge;

This query finds all friends of Paul within a maximum of 5 steps, traversing the FRIEND_OF relationships. The *..5 syntax specifies that the relationship traversal can extend up to 5 hops. The query returns distinct friends, with their name, age, and country attributes.


neo4j$ // Show all the friends of Paul with a maximum distance of 5 steps MATCH (paul:Person {name: "…

	FriendName	Country	FriendAge
1	"Mary"	"Sweden"	29
2	"Joe"	"Sweden"	32
3	"Peter"	"France"	23
4	"Lisa"	"Scotland"	25
5	"Bart"	"Scotland"	25
6	"Denis"	"Scotland"	34
7			

Started streaming 18 records after 21 ms and completed after 65 ms.

Q9 Count all the friends of Paul with maximum distance 5 steps by nationality

MATCH (paul:Person {name: "Paul"})-[:FRIEND_OF*..5]->(friend)

RETURN friend.country AS Country, COUNT(DISTINCT friend) AS FriendCount;

Similar to the previous query, this one counts the distinct friends of Paul within 5 steps, grouped by nationality. It uses *..5 to define the 5-step distance and COUNT(DISTINCT friend) to ensure unique friends are counted. It returns the country and the count of distinct friends for each nationality.


neo4j$ // Count all the friends of Paul with maximum distance 5 steps by nationality MATCH (paul:Pers…

	Country	FriendCount
1	"Sweden"	10
2	"France"	6
3	"Scotland"	8
4	"Ireland"	9
5	"Spain"	1

Started streaming 5 records after 25 ms and completed after 34 ms.

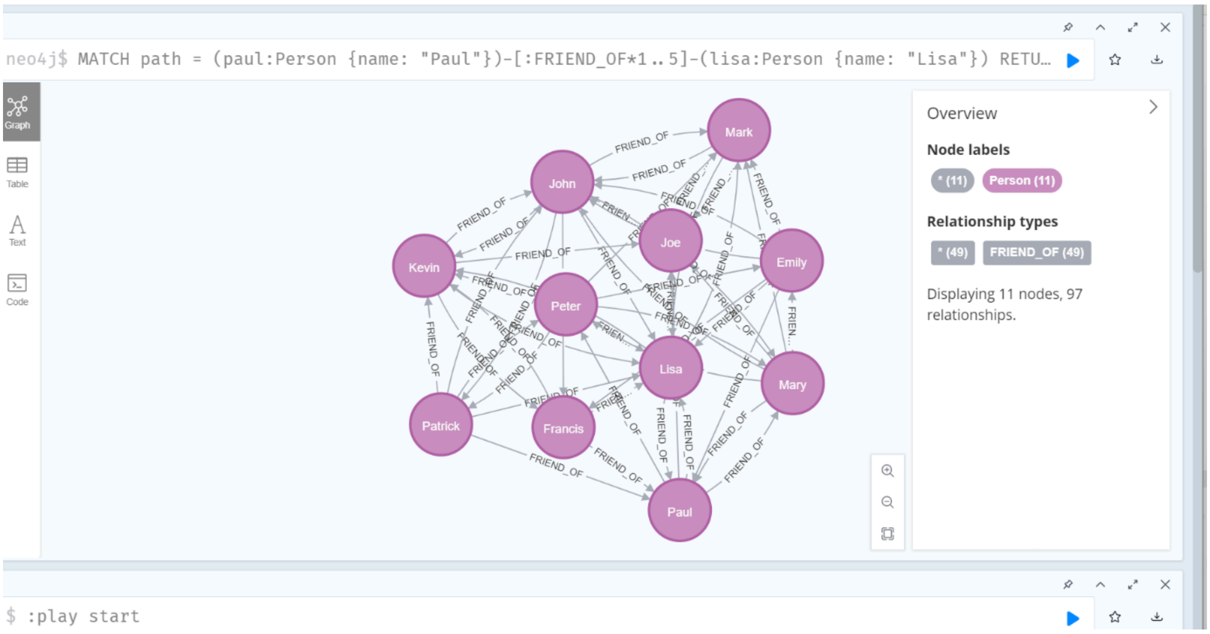
Q10 Show the path(s) between Paul and Lisa. For each path show the length. How many paths are there?

MATCH path = (paul:Person {name: "Paul"})-[:FRIEND_OF*1..5]-(lisa:Person {name: "Lisa"})

RETURN DISTINCT path, LENGTH(path) AS PathLength, COUNT(DISTINCT path) AS PathCount

LIMIT 20;

This query searches for paths between two people, Paul and Lisa, in a graph where the relationship between them is defined as FRIEND_OF. It specifically looks for paths with a length between 1 and 5 hops (relationships) between the two, meaning it considers any path that connects Paul and Lisa through 1 to 5 intermediaries. The query then returns each unique path found, alongside the length of each path (number of relationships), and the total count of distinct paths. The LIMIT 20 ensures that no more than 20 paths are returned, even if more matches exist, due to build reasons.

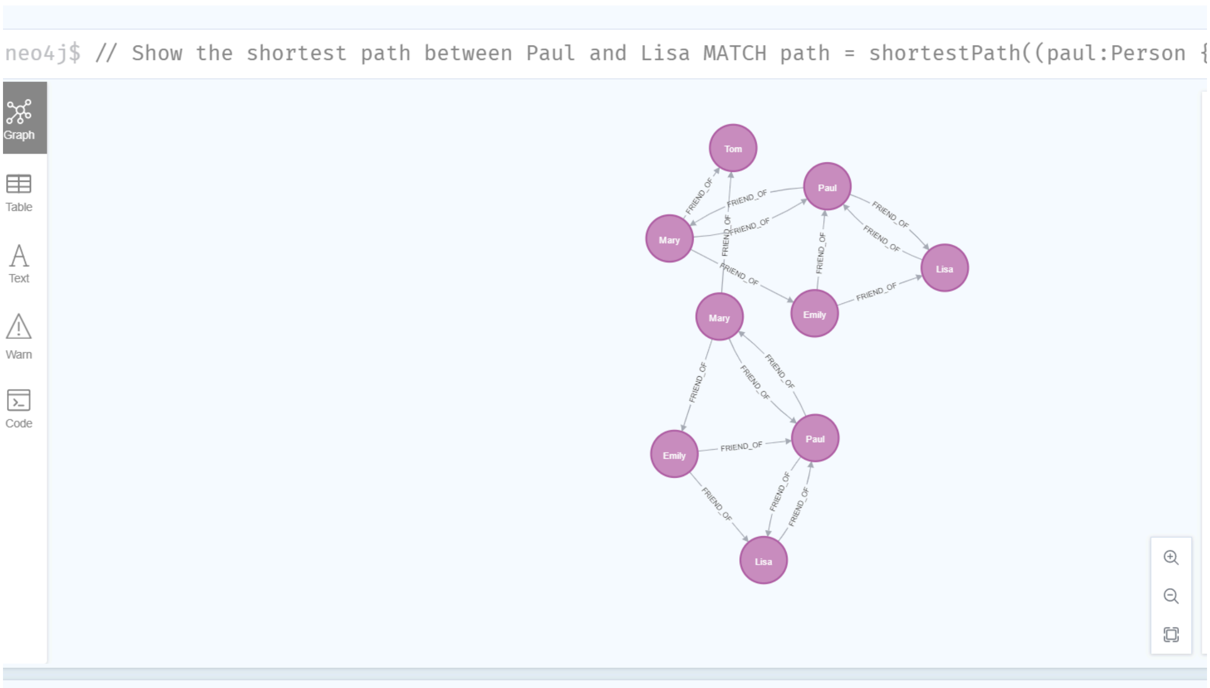


Q11 Show the shortest path between Paul and Lisa

MATCH path = shortestPath((paul:Person {name: "Paul"})-[:FRIEND_OF*]-(lisa:Person {name: "Lisa"}))

RETURN path, LENGTH(path) AS PathLength;

This query uses the shortestPath() function to find the shortest path between Paul and Lisa, which ensures that the number of relationship hops is minimized. The RETURN statement includes both the path and its length using the LENGTH(path) function.



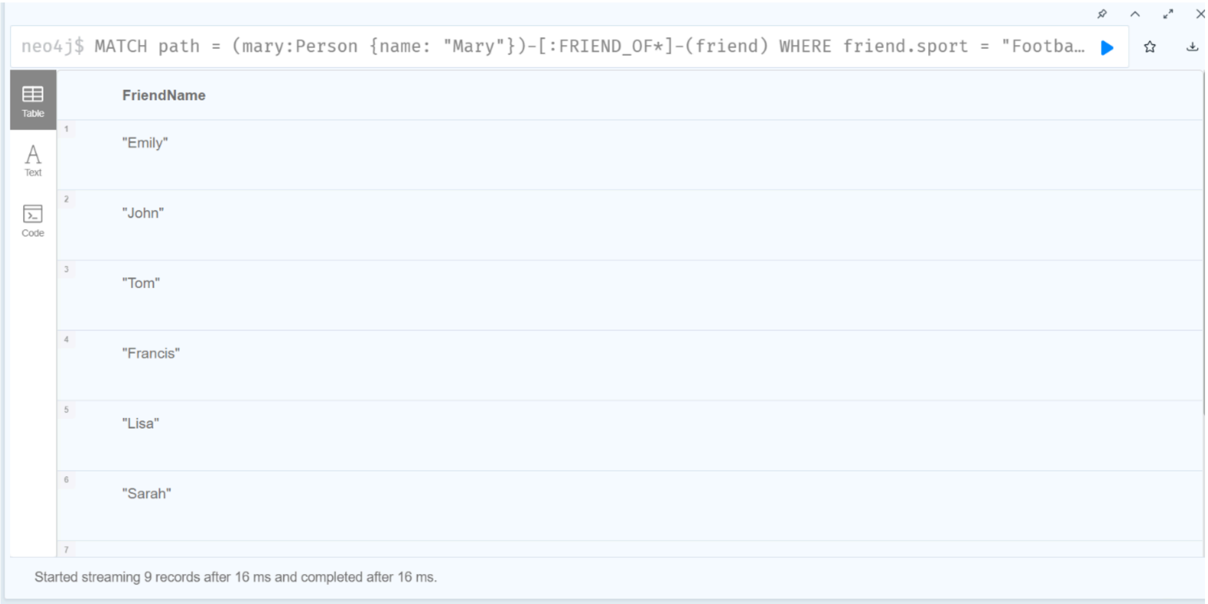
Q12 Show a connection between Mary and all her friends, where the path can only contain persons that play football

MATCH path = (mary:Person {name: "Mary"})-[:FRIEND_OF*]-(friend)

WHERE friend.sport = "Football"

RETURN DISTINCT friend.name AS FriendName;

This query retrieves all friends of "Mary" who play football. It begins by matching paths from Mary to other nodes (friends) in the graph, where the relationship between them is FRIEND_OF. The * in the relationship pattern [:FRIEND_OF*] means that it looks for friends connected by any number of hops.. The WHERE clause then filters the results to only include those friends whose sport property is set to "Football". Finally, the RETURN DISTINCT statement ensures that only unique names of football-playing friends are returned, with the alias FriendName used for their names in the output.



Part 2

> Users > Taime > OneDrive - Technological University Dublin > databases > databases > {} lab9.2.json

```
CREATE (dublin:Airport {city: 'Dublin', country: 'Ireland', code: 'DUB'})
CREATE (cork:Airport {city: 'Cork', country: 'Ireland', code: 'ORK'})
CREATE (london:Airport {city: 'London', country: 'UK', code: 'LHR'})
CREATE (rome:Airport {city: 'Rome', country: 'Italy', code: 'FCO'})
CREATE (moscow:Airport {city: 'Moscow', country: 'Russia', code: 'DME'})
CREATE (hongkong:Airport {city: 'Hong Kong', country: 'China', code: 'HKG'})
CREATE (amsterdam:Airport {city: 'Amsterdam', country: 'Holland', code: 'AMS'})
CREATE (berlin:Airport {city: 'Berlin', country: 'Germany', code: 'TXL'})
CREATE (paris:Airport {city: 'Paris', country: 'France', code: 'CDG'})
CREATE (newyork:Airport {city: 'New York', country: 'USA', code: 'JFK'})
CREATE (chicago:Airport {city: 'Chicago', country: 'USA', code: 'ORD'})
CREATE (sao_paulo:Airport {city: 'Sao Paulo', country: 'Brazil', code: 'GRU'})
CREATE (rio:Airport {city: 'Rio', country: 'Brazil', code: 'GIG'})


CREATE (london)-[:CONNECTED_TO {time: 45, price: 150}]->(dublin)
CREATE (rome)-[:CONNECTED_TO {time: 150, price: 400}]->(london)
CREATE (rome)-[:CONNECTED_TO {time: 120, price: 500}]->(paris)
CREATE (paris)-[:CONNECTED_TO {time: 60, price: 200}]->(dublin)
CREATE (berlin)-[:CONNECTED_TO {time: 240, price: 900}]->(moscow)
CREATE (paris)-[:CONNECTED_TO {time: 30, price: 100}]->(amsterdam)
CREATE (berlin)-[:CONNECTED_TO {time: 120, price: 900}]->(dublin)
CREATE (london)-[:CONNECTED_TO {time: 700, price: 1100}]->(newyork)
CREATE (dublin)-[:CONNECTED_TO {time: 360, price: 800}]->(newyork)
CREATE (dublin)-[:CONNECTED_TO {time: 50, price: 50}]->(cork)
CREATE (dublin)-[:CONNECTED_TO {time: 150, price: 70}]->(rome)
CREATE (dublin)-[:CONNECTED_TO {time: 480, price: 890}]->(chicago)
CREATE (amsterdam)-[:CONNECTED_TO {time: 660, price: 750}]->(hongkong)
CREATE (london)-[:CONNECTED_TO {time: 700, price: 1000}]->(hongkong)
CREATE (dublin)-[:CONNECTED_TO {time: 90, price: 60}]->(amsterdam)
CREATE (moscow)-[:CONNECTED_TO {time: 720, price: 1000}]->(newyork)
CREATE (moscow)-[:CONNECTED_TO {time: 420, price: 500}]->(hongkong)
CREATE (newyork)-[:CONNECTED_TO {time: 240, price: 430}]->(chicago)
```

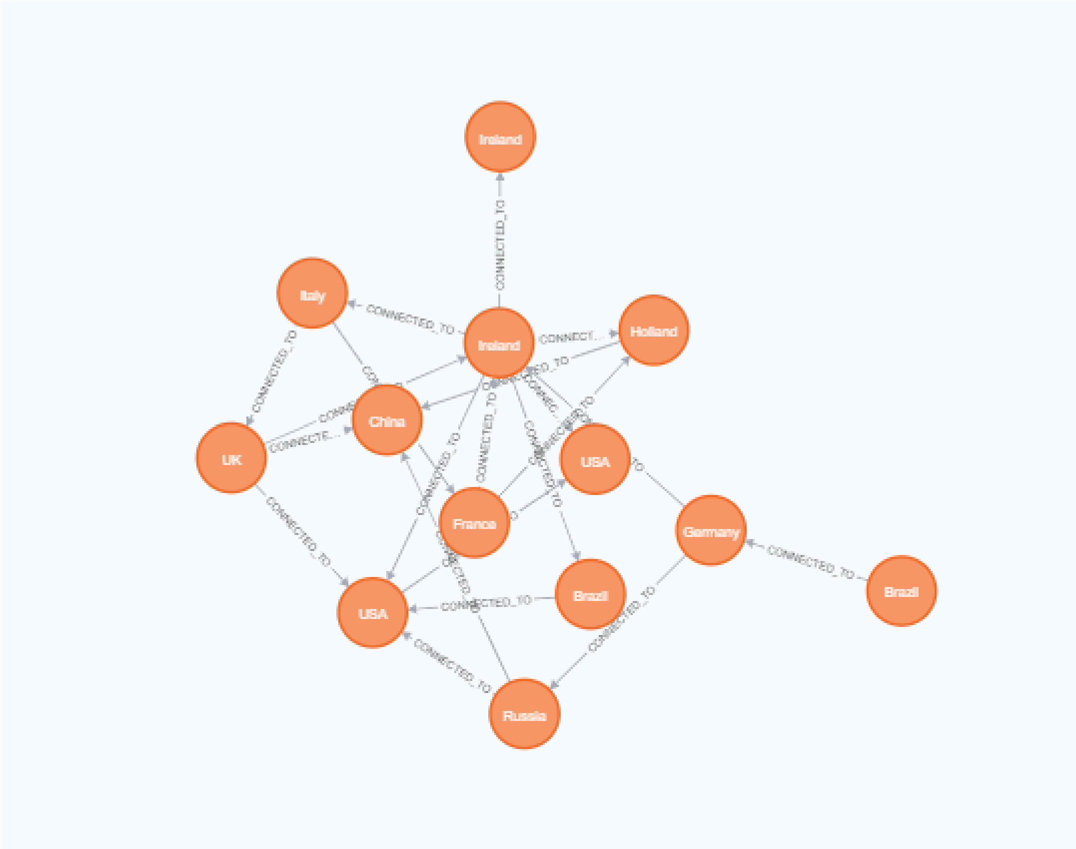
Question1 Find the total time from Moscow to Rio. Show also the path (airport connections)

MATCH p = (moscow)-[:CONNECTED_TO*]->(rio)

WITH p, REDUCE(totalTime = 0, r IN relationships(p) | totalTime + r.time) AS totalTime

RETURN p, totalTime

The query finds all possible paths between the nodes representing Moscow and Rio using the CONNECTED_TO relationship, regardless of the number of intermediate connections. The MATCH clause identifies these paths and stores them in the variable p. The REDUCE function is used to calculate the total travel time for each path by iterating over the relationships (flights) in the path and summing their time properties. Finally, the query returns each path (p) and its corresponding total travel time (totalTime). However, from the providied list of connected airports rio and Moscow do not connect at all. If I added in an extra connection however then there would be a possible solution. Below is a graph of the connected airports.



Q2 Show all the flights from Dublin to any destination and sort them by price (from the most expensive

MATCH (dublin:Airport)-[r:CONNECTED_TO]->(destination:Airport)

RETURN destination.city AS Destination, r.price AS Price, r.time AS Time

ORDER BY r.price DESC;

This query identifies all direct flights from Dublin to other destinations by matching airports connected by the CONNECTED_TO relationship. It retrieves the destination city's name, the flight price, and the travel time. The query then sorts the results in descending order by price, ensuring that the most expensive flights are shown first. This allows for an easy comparison of flight options from Dublin, ordered by their cost.

```
o4j$ MATCH (dublin:Airport)-[r:CONNECTED_TO]→(destination:Airport) RETURN destination.city AS Dest…
```

	Destination	Price	Time
1	"New York"	1100	700
2	"Berlin"	1100	1200
3	"Hong Kong"	1000	700
4	"New York"	1000	720
5	"Dublin"	900	120
6	"Moscow"	900	240
7			

```
eo4j$ MATCH (dublin:Airport)-[r:CONNECTED_TO]→(destination:Airport) RETURN destination.city AS Dest…
```

	Destination	Price	Time
	"New York"	650	840
12	"Paris"	500	120
13	"Hong Kong"	500	420
14	"Chicago"	430	240
15	"London"	400	150
16	"Dublin"	200	60
17	"Dublin"	150	45

Started streaming 21 records after 19 ms and completed after 28 ms.

```
eo4j$ MATCH (dublin:Airport)-[r:CONNECTED_TO]→(destination:Airport) RETURN destination.city AS Dest…
```

	Destination	Price	Time
16	"Dublin"	200	60
17	"Dublin"	150	45
18	"Amsterdam"	100	30
19	"Rome"	70	150
20	"Amsterdam"	60	90
21	"Cork"	50	50

Q3 Show what can be reached from Chicago in one or two steps (= direct flight or 1 change only)

MATCH (chicago)-[:CONNECTED_TO]->(firstHop:Airport)

MATCH (firstHop)-[:CONNECTED_TO]->(secondHop:Airport)

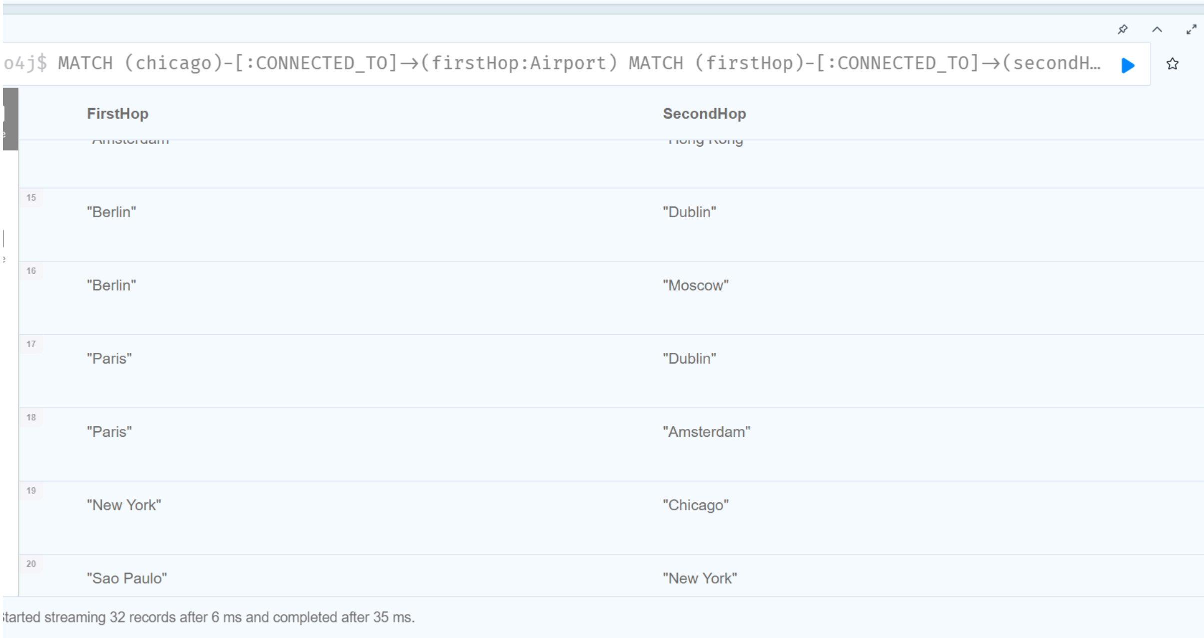
RETURN firstHop.city AS FirstHop, secondHop.city AS SecondHop

UNION

MATCH (chicago)-[:CONNECTED_TO]->(destination:Airport)

RETURN destination.city AS FirstHop, NULL AS SecondHop;

This query finds destinations that can be reached from Chicago in either one or two steps. It first identifies all airports directly connected to Chicago and then matches airports connected to those airports in a second step (representing flights with one change). The query uses UNION to combine two results: one for direct flights (one step) and one for flights with one change (two steps). This allows users to explore both direct and connecting flight options from Chicago.



Q4 what can be reached from London in less than 240 minutes (4 hours).

MATCH (london)-[r:CONNECTED_TO]->(destination:Airport)

WHERE r.time < 240

RETURN destination.city AS Destination, r.time AS Time, r.price AS Price;

This Cypher query is used to find destinations that can be reached directly from London with a travel time of less than 240 minutes (i.e., less than 4 hours). The query works by matching all CONNECTED_TO relationships originating from the London airport node. For each relationship (r), it checks whether the time property (representing the flight duration in minutes) is less than 240 minutes. If this condition is satisfied, the query returns the city of the destination airport (destination.city), the flight time (r.time), and the price of the flight (r.price). This allows you to see which airports can be reached from London in less than 4 hours and the corresponding flight details (time and price).

```
neo4j$ MATCH (london)-[r:CONNECTED_TO]→(destination:Airport) WHERE r.time < 240 RETURN destination.c…
```

	Destination	Time	Price
1	"Dublin"	45	150
2	"Dublin"	120	900
3	"Dublin"	60	200
4	"Cork"	50	50
5	"London"	150	400
6	"Rome"	150	70
7			

Started streaming 9 records after 43 ms and completed after 50 ms.

```
eo4j$ MATCH (london)-[r:CONNECTED_TO]→(destination:Airport) WHERE r.time < 240 RETURN destination.c…
```

	Destination	Time	Price
4	"Cork"	50	50
5	"London"	150	400
6	"Rome"	150	70
7	"Amsterdam"	90	60
8	"Amsterdam"	30	100
9	"Paris"	120	500

Started streaming 9 records after 43 ms and completed after 50 ms.

9.1

```
CREATE (e1:Person { name: "Mary", country: "Sweden", age: 29, sport: "Hockey" }),
(e2:Person { name: "Emily", country: "Ireland", age: 19, sport: "Football" }),
(e3:Person { name: "Mark", country: "Sweden", age: 23, sport: "Rugby" }),
(e4:Person { name: "Joe", country: "Sweden", age: 32, sport: "Hockey" }),
(e5:Person { name: "John", country: "Ireland", age: 31, sport: "Football" }),
(e6:Person { name: "Peter", country: "France", age: 23, sport: "Rugby" }),
(e7:Person { name: "Paul", country: "Sweden", age: 25, sport: "Hockey" }),
(e8:Person { name: "Kevin", country: "Ireland", age: 17, sport: "Football" }),
(e9:Person { name: "Patrick", country: "Sweden", age: 21, sport: "Rugby" }),
(e10:Person { name: "Sarah", country: "Ireland", age: 35, sport: "Football" }),
(e11:Person { name: "Julia", country: "Scotland", age: 28, sport: "Football" }),
(e12:Person { name: "Hilary", country: "France", age: 24, sport: "Rugby" }),
(e13:Person { name: "Francis", country: "France", age: 25, sport: "Football" }),
(e14:Person { name: "Lisa", country: "Scotland", age: 25, sport: "Football" }),
(e15:Person { name: "Bart", country: "Scotland", age: 25, sport: "Rugby" }),
(e16:Person { name: "Denis", country: "Scotland", age: 34, sport: "Football" }),
(e1)-[:FRIEND_OF]->(e2),(e1)-[:FRIEND_OF]->(e3),
(e1)-[:FRIEND_OF]->(e4),(e1)-[:FRIEND_OF]->(e5),
(e1)-[:FRIEND_OF]->(e7),(e1)-[:FRIEND_OF]->(e6),
(e2)-[:FRIEND_OF]->(e13),(e2)-[:FRIEND_OF]->(e3),
(e2)-[:FRIEND_OF]->(e14),(e2)-[:FRIEND_OF]->(e5),
(e2)-[:FRIEND_OF]->(e5),(e2)-[:FRIEND_OF]->(e7),
(e3)-[:FRIEND_OF]->(e10),(e3)-[:FRIEND_OF]->(e4),
(e3)-[:FRIEND_OF]->(e14),(e3)-[:FRIEND_OF]->(e5),
(e4)-[:FRIEND_OF]->(e10),(e4)-[:FRIEND_OF]->(e3),
(e4)-[:FRIEND_OF]->(e11),(e4)-[:FRIEND_OF]->(e5),
(e5)-[:FRIEND_OF]->(e13),(e5)-[:FRIEND_OF]->(e3),
(e5)-[:FRIEND_OF]->(e14),(e5)-[:FRIEND_OF]->(e1),
(e5)-[:FRIEND_OF]->(e8),(e5)-[:FRIEND_OF]->(e12),
(e5)-[:FRIEND_OF]->(e9),(e5)-[:FRIEND_OF]->(e10),
(e6)-[:FRIEND_OF]->(e14),(e6)-[:FRIEND_OF]->(e1),
(e6)-[:FRIEND_OF]->(e8),(e6)-[:FRIEND_OF]->(e2),
(e6)-[:FRIEND_OF]->(e9),(e6)-[:FRIEND_OF]->(e3),
```

```
(e7)-[:FRIEND_OF]->(e14),(e7)-[:FRIEND_OF]->(e1),
(e7)-[:FRIEND_OF]->(e15),(e7)-[:FRIEND_OF]->(e6),
(e7)-[:FRIEND_OF]->(e16),(e7)-[:FRIEND_OF]->(e4),
(e8)-[:FRIEND_OF]->(e13),(e8)-[:FRIEND_OF]->(e14),
(e8)-[:FRIEND_OF]->(e12),(e8)-[:FRIEND_OF]->(e5),
(e8)-[:FRIEND_OF]->(e11),(e8)-[:FRIEND_OF]->(e4),
(e9)-[:FRIEND_OF]->(e8),(e9)-[:FRIEND_OF]->(e14),
(e9)-[:FRIEND_OF]->(e7),(e9)-[:FRIEND_OF]->(e5),
(e9)-[:FRIEND_OF]->(e6),(e9)-[:FRIEND_OF]->(e12),
(e10)-[:FRIEND_OF]->(e3),(e10)-[:FRIEND_OF]->(e5),
(e10)-[:FRIEND_OF]->(e2),(e10)-[:FRIEND_OF]->(e15),
(e11)-[:FRIEND_OF]->(e3),(e11)-[:FRIEND_OF]->(e6),
(e11)-[:FRIEND_OF]->(e4),(e11)-[:FRIEND_OF]->(e8),
(e12)-[:FRIEND_OF]->(e4),(e12)-[:FRIEND_OF]->(e1),
(e13)-[:FRIEND_OF]->(e8),(e13)-[:FRIEND_OF]->(e14),
(e13)-[:FRIEND_OF]->(e7),(e13)-[:FRIEND_OF]->(e15),
(e13)-[:FRIEND_OF]->(e16),(e13)-[:FRIEND_OF]->(e12),
(e14)-[:FRIEND_OF]->(e7),(e14)-[:FRIEND_OF]->(e3),
(e14)-[:FRIEND_OF]->(e8),(e14)-[:FRIEND_OF]->(e4),
(e15)-[:FRIEND_OF]->(e1),(e15)-[:FRIEND_OF]->(e13),
(e15)-[:FRIEND_OF]->(e3),(e15)-[:FRIEND_OF]->(e4),
(e16)-[:FRIEND_OF]->(e9),(e16)-[:FRIEND_OF]->(e13),
(e16)-[:FRIEND_OF]->(e10),(e16)-[:FRIEND_OF]->(e3);
```

```
CREATE (tom:Person {name: "Tom", age: 28, country: "Spain", sport: "Football"})
WITH tom
MATCH (mary:Person {name: "Mary"})
CREATE (mary)-[:FRIEND_OF]->(tom);
```

```
CREATE (bill:Person {name: "Bill", age: 23, country: "Ireland"})
WITH bill
MATCH (mary:Person {name: "Mary"}), (denis:Person {name: "Denis"})
CREATE (mary)-[:FRIEND_OF]->(bill), (denis)-[:FRIEND_OF]->(bill);



// Show the age of Denis and his friends
MATCH (denis:Person {name: "Denis"})-[:FRIEND_OF]->(friend)
RETURN denis.age AS DenisAge, friend.name AS FriendName, friend.age AS FriendAge;

// Show all the person from Scotland
MATCH (person:Person {country: "Scotland"})
RETURN person.name AS Name, person.age AS Age, person.sport AS Sport;

// Show all the person with age less or equal than 20 from Ireland
MATCH (person:Person {country: "Ireland"})
WHERE person.age <= 20
RETURN person.name AS Name, person.age AS Age;

// Show all the person with age less or equal 30 playing football
MATCH (person:Person {sport: "Football"})
WHERE person.age <= 30
RETURN person.name AS Name, person.age AS Age, person.country AS Country;

// Count the person by country
MATCH (person:Person)
RETURN person.country AS Country, COUNT(person) AS Count;

// Show the average age of the person group by sport
MATCH (person:Person)
RETURN person.sport AS Sport, AVG(person.age) AS AverageAge;

// Show all the direct friends of Mary
```

```
MATCH (mary:Person {name: "Mary"})-[:FRIEND_OF]->(friend)
RETURN friend.name AS FriendName, friend.age AS FriendAge, friend.country AS Country;

// Show all the friends of Paul with a maximum distance of 5 steps
MATCH (paul:Person {name: "Paul"})-[:FRIEND_OF*..5]->(friend)
RETURN DISTINCT friend.name AS FriendName, friend.country AS Country, friend.age AS FriendAge;

// Count all the friends of Paul with maximum distance 5 steps by nationality
MATCH (paul:Person {name: "Paul"})-[:FRIEND_OF*..5]->(friend)
RETURN friend.country AS Country, COUNT(DISTINCT friend) AS FriendCount;

// Show the path(s) between Paul and Lisa. For each path show the length. How many paths are there?
MATCH path = (paul:Person {name: "Paul"})-[:FRIEND_OF*1..5]-(lisa:Person {name: "Lisa"})
RETURN DISTINCT path, LENGTH(path) AS PathLength, COUNT(DISTINCT path) AS PathCount
LIMIT 20;

// Show the shortest path between Paul and Lisa
MATCH path = shortestPath((paul:Person {name: "Paul"})-[:FRIEND_OF*]-(lisa:Person {name: "Lisa"}))
RETURN path, LENGTH(path) AS PathLength;

// Show a connection between Mary and all her friends, where the path can only contain persons that play football
MATCH path = (mary:Person {name: "Mary"})-[:FRIEND_OF*]-(friend)
WHERE friend.sport = "Football"
RETURN DISTINCT friend.name AS FriendName;
```

9.2

```
CREATE (dublin:Airport {city: 'Dublin', country: 'Ireland', code: 'DUB'})
CREATE (cork:Airport {city: 'Cork', country: 'Ireland', code: 'ORK'})
CREATE (london:Airport {city: 'London', country: 'UK', code: 'LHR'})
CREATE (rome:Airport {city: 'Rome', country: 'Italy', code: 'FCO'})
CREATE (moscow:Airport {city: 'Moscow', country: 'Russia', code: 'DME'})
CREATE (hongkong:Airport {city: 'Hong Kong', country: 'China', code: 'HKG'})
CREATE (amsterdam:Airport {city: 'Amsterdam', country: 'Holland', code: 'AMS'})
```

```
CREATE (berlin:Airport {city: 'Berlin', country: 'Germany', code: 'TXL'})
CREATE (paris:Airport {city: 'Paris', country: 'France', code: 'CDG'})
CREATE (newyork:Airport {city: 'New York', country: 'USA', code: 'JFK'})
CREATE (chicago:Airport {city: 'Chicago', country: 'USA', code: 'ORD'})
CREATE (sao_paulo:Airport {city: 'Sao Paulo', country: 'Brazil', code: 'GRU'})
CREATE (rio:Airport {city: 'Rio', country: 'Brazil', code: 'GIG'})


CREATE (london)-[:CONNECTED_TO {time: 45, price: 150}]->(dublin)
CREATE (rome)-[:CONNECTED_TO {time: 150, price: 400}]->(london)
CREATE (rome)-[:CONNECTED_TO {time: 120, price: 500}]->(paris)
CREATE (paris)-[:CONNECTED_TO {time: 60, price: 200}]->(dublin)
CREATE (berlin)-[:CONNECTED_TO {time: 240, price: 900}]->(moscow)
CREATE (paris)-[:CONNECTED_TO {time: 30, price: 100}]->(amsterdam)
CREATE (berlin)-[:CONNECTED_TO {time: 120, price: 900}]->(dublin)
CREATE (london)-[:CONNECTED_TO {time: 700, price: 1100}]->(newyork)
CREATE (dublin)-[:CONNECTED_TO {time: 360, price: 800}]->(newyork)
CREATE (dublin)-[:CONNECTED_TO {time: 50, price: 50}]->(cork)
CREATE (dublin)-[:CONNECTED_TO {time: 150, price: 70}]->(rome)
CREATE (dublin)-[:CONNECTED_TO {time: 480, price: 890}]->(chicago)
CREATE (amsterdam)-[:CONNECTED_TO {time: 660, price: 750}]->(hongkong)
CREATE (london)-[:CONNECTED_TO {time: 700, price: 1000}]->(hongkong)
CREATE (dublin)-[:CONNECTED_TO {time: 90, price: 60}]->(amsterdam)
CREATE (moscow)-[:CONNECTED_TO {time: 720, price: 1000}]->(newyork)
CREATE (moscow)-[:CONNECTED_TO {time: 420, price: 500}]->(hongkong)
CREATE (newyork)-[:CONNECTED_TO {time: 240, price: 430}]->(chicago)
CREATE (dublin)-[:CONNECTED_TO {time: 900, price: 800}]->(sao_paulo)
CREATE (sao_paulo)-[:CONNECTED_TO {time: 840, price: 650}]->(newyork)
CREATE (rio)-[:CONNECTED_TO {time: 1200, price: 1100}]->(berlin)

MATCH p = (moscow)-[:CONNECTED_TO*]->(rio)
WITH p, REDUCE(totalTime = 0, r IN relationships(p) | totalTime + r.time) AS totalTime
RETURN p, totalTime
```

```
MATCH (dublin:Airport)-[r:CONNECTED_TO]->(destination:Airport)
RETURN destination.city AS Destination, r.price AS Price, r.time AS Time
ORDER BY r.price DESC;
```

```
MATCH (chicago)-[:CONNECTED_TO]->(firstHop:Airport)
MATCH (firstHop)-[:CONNECTED_TO]->(secondHop:Airport)
RETURN firstHop.city AS FirstHop, secondHop.city AS SecondHop
UNION
MATCH (chicago)-[:CONNECTED_TO]->(destination:Airport)
RETURN destination.city AS FirstHop, NULL AS SecondHop;
```

```
MATCH (london)-[r:CONNECTED_TO]->(destination:Airport)
WHERE r.time < 240
RETURN destination.city AS Destination, r.time AS Time, r.price AS Price;
```