Data Models and Data Bases Assignment II.

Vitrai Gábor – ABIOWE

Technologies for using the selected database management systems:

Choice of NoSQL databases. Why did you choose these databases?

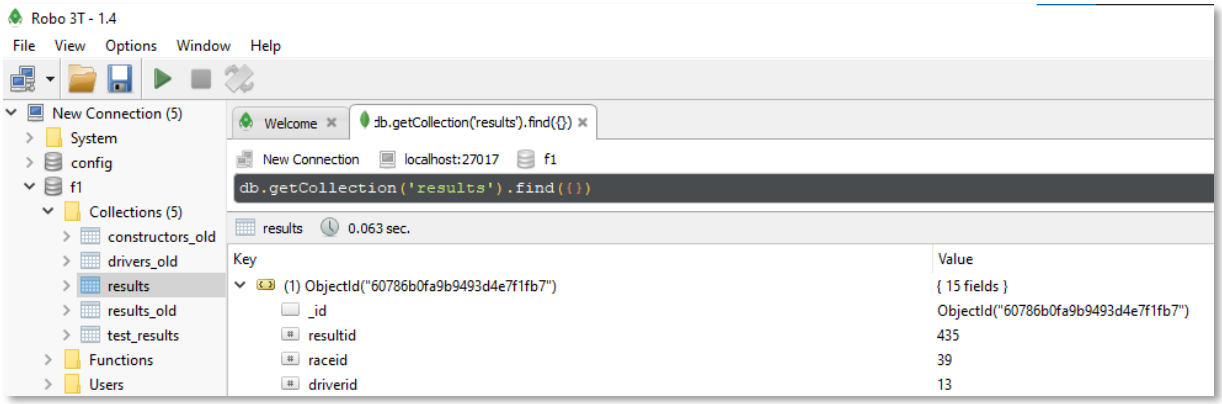
My selected 2 databases are **MongoDB** and **Neo4J**. Originally, I selected MongoDB because I had former experience with it, but I also know that it is commonly used in modern services. Neo4j was completely new for me, and I found it very interesting that we can observe databases as collections of nodes. I always loved the concept of storing data in JSON format. Earlier I used online cluster based mongoDB services and I never had any issues or complains about it.

MongoDB:

In the repository, the RAR file, named *db.rar* contains the database which was used for this assignment.

After downloading and installing the mongoDB server "mongodb-win32-x86_64-2012plus-4.2.12" create a folder named "data" in the giver drive's root folder. Inside that folder create another folder called "db". After launching the database server for the first time, shut down the DB and copy paste the content of this RAR file inside the "db" folder.

When we got to the point to have a running server, we need a tool to connect to it and run queries. For this purpose, I used a tool called Robo 3T (1.4.3.). This is a free and easy to use software for mongoDB.



Neo4j:

I downloaded the community version of the server and used the Cypher online tool to implement queries and commands.

The RAR file called *neo4j.rar* is the compressed database used for this assignment. In order to use it, copy it inside the data/databases folder.

Since this is the default database, it is enough to overwrite the originally generated empty database.

Every resource can be found in the following repository:

https://github.com/BSSB33/data-models-and-databases-class

Process of loading the data:

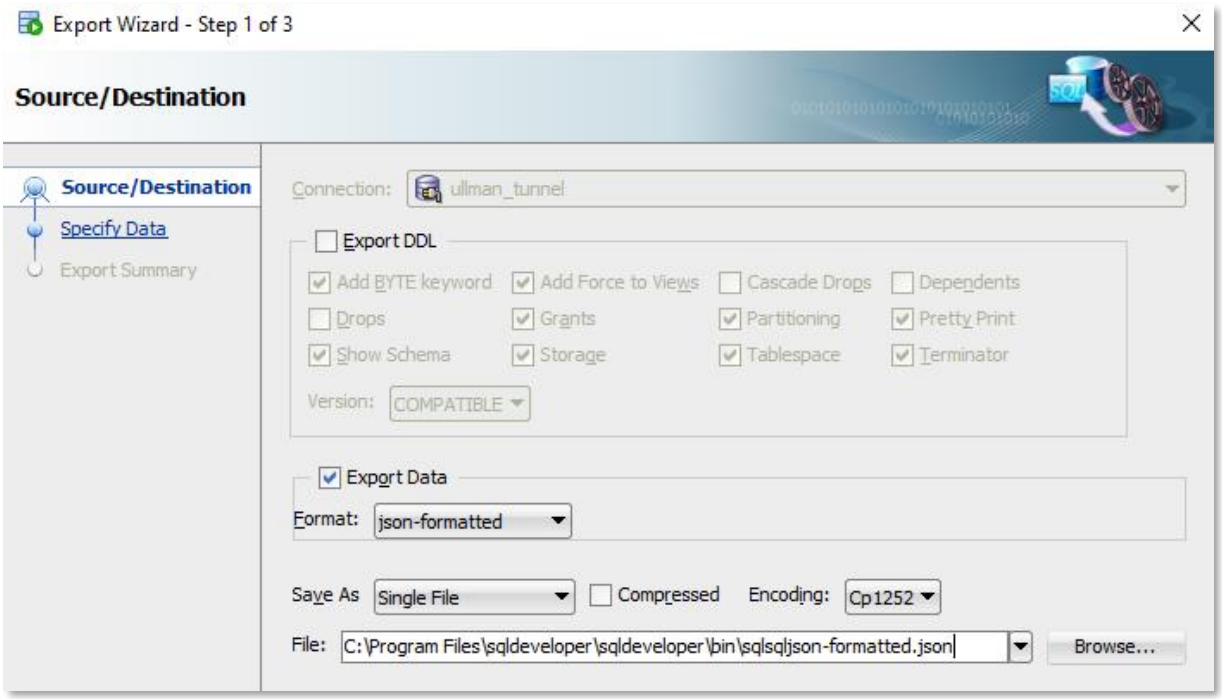
How did you import the data? With command line examples, or screenshots.

MongoDB:

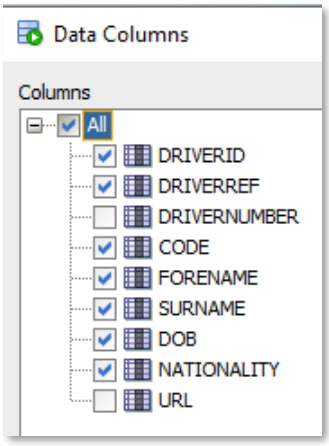
Loading the data was quite challenging. First, I was not sure if I want to create 2 (or more) collections out of my relational database tables or merge them into a single collection. Based on the practice class materials I decided to merge hem together so it will be similar to the twitter dataset we used while I could make better and more interesting queries for the assignment.

I started with exporting into JSON the selected tables from Oracle's SQL developer, as this tool was used in the previous assignment. I also selected only the interesting attributes.

The Export Wizard:



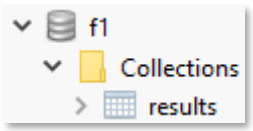
First I realized that merging 3 tables is challenging, but luckily I only had One – to – Many relationships so I quickly wrote a script to replace the ID-s with the given objects, so in no time, I had the merged data of the 3 selected tables (results, drivers, races).



The exported/merged JSON file was formatted in a different standard which the mongoDB import tool supports, so I had to convert them, by removing some colons, and brackets.

With the final json input file, (results.json) I copied it to the bin folder of the mongoDB server and I gave out the following single line order, to import the collection to the running database.

```
mongodbimport.exe --db f1 --collection results --file results.json
```



The console gives us feedback that the data was imported displaying the number of successfully imported lines. After this process, the data could be found in the collections folder of the database.

Neo4j:

I Started by exporting the same data tables from oracle, but now with CSV format. (delimiters are ‘,’ characters).

Next, I casted the types of the attributes (originally in the case of neo4j, every datatype is simple String, so we have to manually modify it) and I filtered out some attributes for simplicity.

The following codes were used to import the 3 datasets:

Results:

```
LOAD CSV WITH HEADERS FROM 'file:///results.csv' AS row
WITH toInteger(row.DRIVERID) as driverId, toInteger(row.RESULTID) as resultId, toInteger(ro
w.RACEID) as raceId, row.POINTS as points, toInteger(row.LAPS) as laps, row.TIME as time, r
ow.FASTESTLAP as fastestLap, row.FASTESTLAPSPEED as fastestLapSpeed
MERGE (r:Result {resultId: resultId})
  SET r.driverId = driverId, r.raceId = raceId, r.points = points, r.laps = laps, r.time = time, r.fas
testLap = fastestLap, r. fastestLapSpeed = fastestLapSpeed
  RETURN count(r);
```

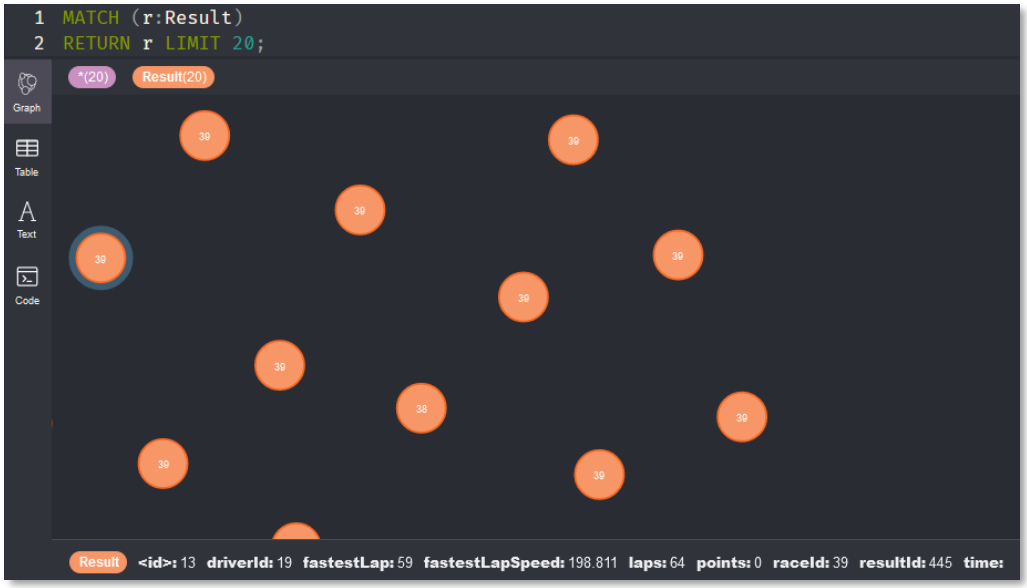
Races:

```
LOAD CSV WITH HEADERS FROM 'file:///drivers.csv' AS row
WITH toInteger(row.DRIVERID) as driverId, row.DRIVERREF as driverRef, row.DRIVERNUMBE
R as driverNumber, row.FORENAME as foreName, row.SURNAME as surName, row.DOB as d
ob, row.NATIONALITY as nationality
MERGE (d:Drivers {driverId: driverId})
  SET d.dob = dob, d.driverNumber = driverNumber, d.driverRef = driverRef, d.foreName = fo
reName, d. surName = surName, d.nationality = nationality
  RETURN count(d);
```

Races:

```
LOAD CSV WITH HEADERS FROM 'file:///races.csv' AS row
WITH toInteger(row.RACEID) as raceId, toInteger(row.YEAR) as year, row.ROUND as round, r
ow.NAME as name, row.DATEOFRACE as date
MERGE (ra:Races {raceId: raceId})
  SET ra.year = year, ra.round = round, ra.name = name, ra.date = date
  RETURN count(ra);
```

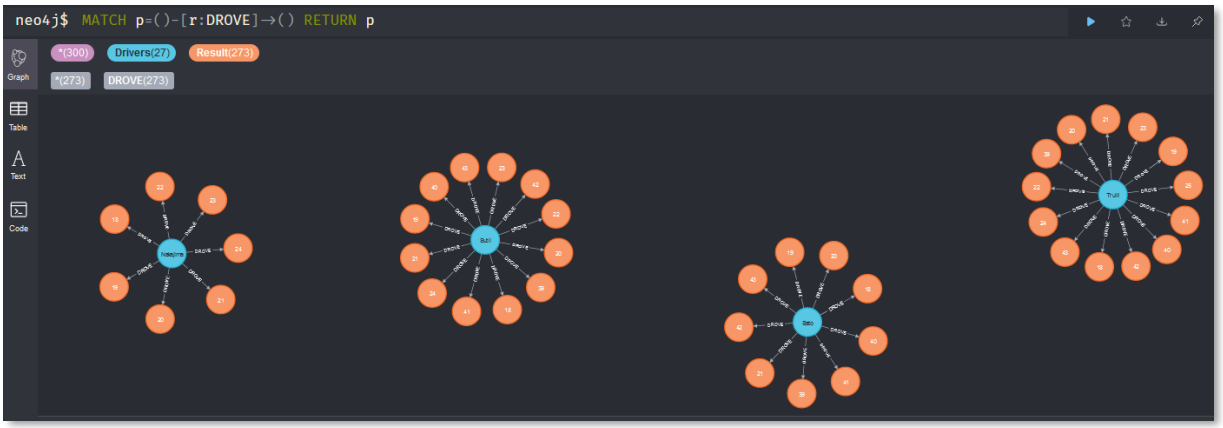
Result after the first import:



Now that I have our collections, I had to create the relations between the nodes. The following code was used to create the DROVE relation between the Drivers and Results set.

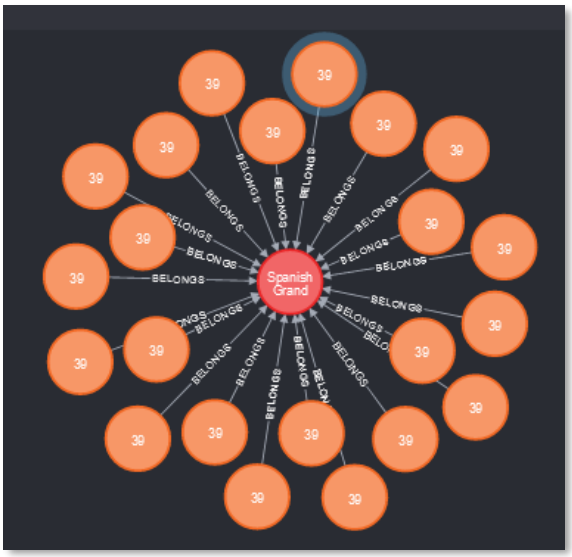
```
MATCH
  (d:Drivers),
  (r:Result)
WHERE d.driverId = r.driverId
CREATE (d)-[relation:DROVE]->(r)
RETURN type(relation), relation.name
```

Afterwards the graphs looked the following: What we can see are different drivers and their connected results.



Let's create the relation between the results and the races sets.

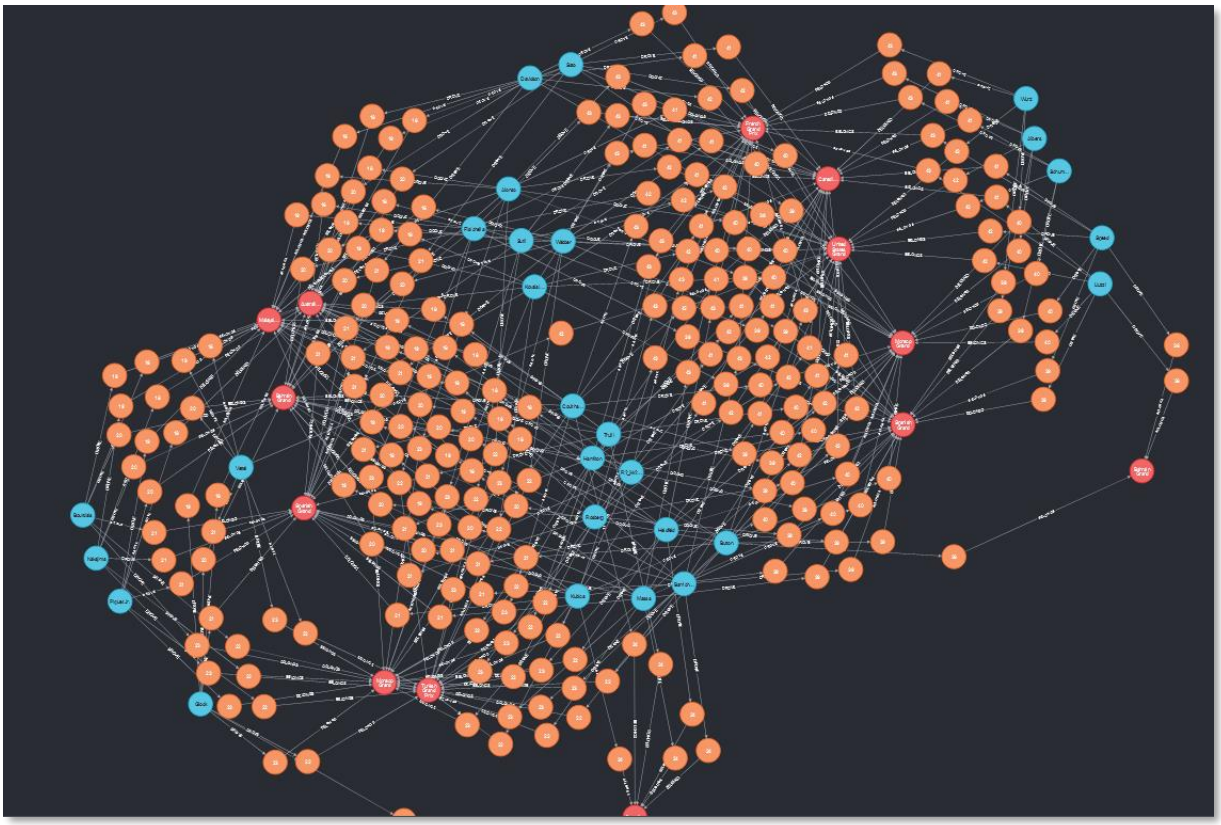
Code and result example:




```
MATCH
  (ra:Races),
  (res:Result)
WHERE ra.raceId = res.raceId
CREATE (res)-[relation:BELONGS]->(ra)
RETURN type(relation), relation.name
```

On the left side we can see 25 connected results which belong to the Spanish Grand Prix.

Now the moment of truth, lets visualize the 3 collection on a graph (with 250 limit)



What we can see is the results of connecting the collections in a proper way. The red dots are the races, connected to their correct result records (orange dots), which are connected to the given drivers (blue dots). So, more result records are connected to one driver as a race consists of many drivers.

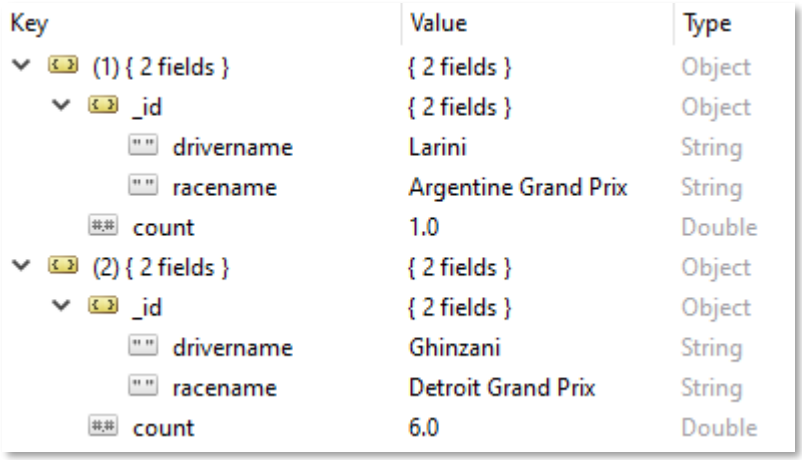
NoSQL queries with results, screenshots:

The formulation, code, and result of the 3-3 queries

MongoDB:

1. The very first query I created tells us **How many results are there per driver for each racetrack.** With the help of this query, we get the distinct sets of a driver and a race name, with the count of how many times did the given driver did a race on a specific Grand Prix.
   As a possible improvement we can use this to filter further by drivers and see on which races did our selected driver started on.

   Output example:



Key	Value	Type
(1) { 2 fields }	{ 2 fields }	Object
_id	{ 2 fields }	Object
drivername	Larini	String
racename	Argentine Grand Prix	String
count	1.0	Double
(2) { 2 fields }	{ 2 fields }	Object
_id	{ 2 fields }	Object
drivername	Ghinzani	String
racename	Detroit Grand Prix	String
count	6.0	Double

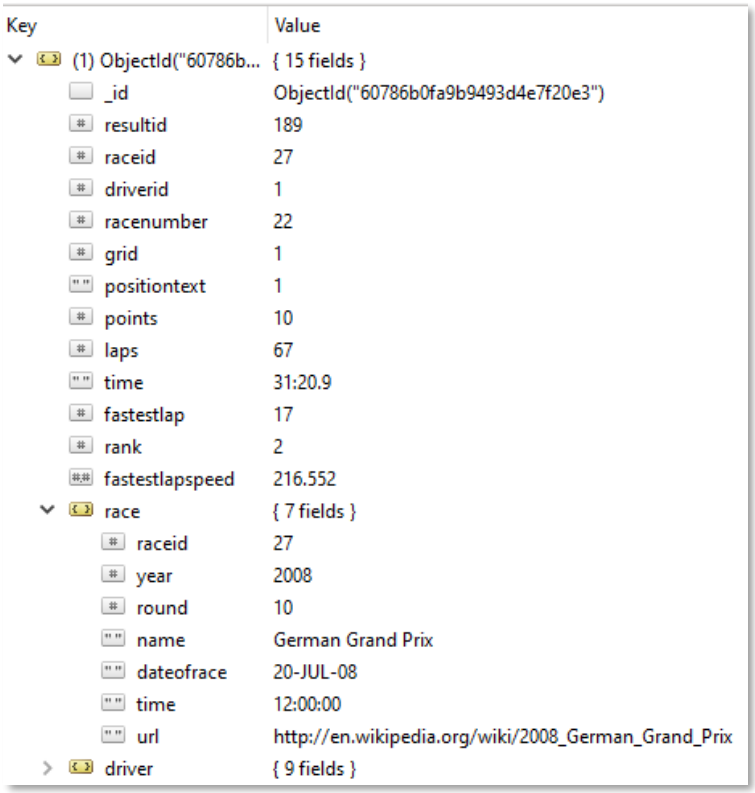
Code:

```
db.results.aggregate([
    {
        $group: {
            _id: {"drivername": "$driver.surname", "racename":"$race.name"},
            count: { $sum: 1 } }
        }
])
```

The query groups by the driver's name and the race's name, while creating a counter (count) which summarizes the number of occurrences.

2. The second query returns the most recent 30 races in Germany. To properly phrase the question: **What were the results of the most recent 30, F1 races on the German Grand Prix?**

The output looks the following:



Every record contains the complete result log, including the race and the corresponding driver for each record. If we take a closer look to some of the other outputs, we can notice that the result is listed with the more recent races on top.

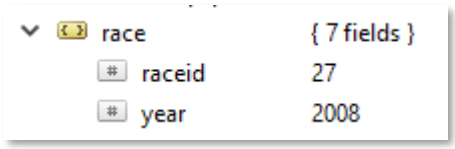


Figure 1: First output

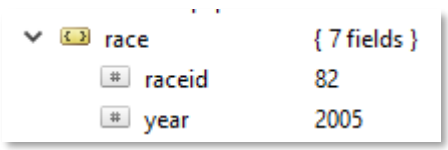


Figure 2: 30th output

The code itself looks the following:

```
db.results.aggregate([
        { $match: {"race.name": "German Grand Prix"} },
        { $limit: 30 },
        { $sort: {"race.year": -1} },
        ])
```

The aggregation filters out all the races which happened on the German Grand Prix with the *$match* attribute and then in the next line, the code limits the number of outputs while sorting the returned records in a descending was examining the year of the race.

3. The last query answers the following questions: **Which driver has the greatest number of races? How Many?**

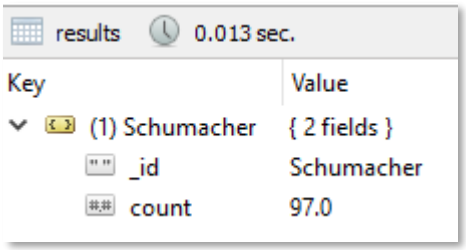
For this, we needed another aggregation, like the previous one. This query includes tools form the two previous queries. It uses a filter – now for the final position of the given driver – to find who won the given races, then it groups the drivers by name and counts how many times does the match attribute returned the given driver. After this complicated part, the computer just sorts the output in a descending way and shows us the first result, as we are only interested in who had the most win until the final year of our database.

Code:

```
db.results.aggregate(
        [
                { $match: {"positiontext" : "1"}},
                { $group:
                        {_id:"$driver.surname",
                            count: {$sum: 1}
                        }
                },
                { $sort: {count: -1}},
                { $limit: 1 }
        ]
)
```

Results:

Just in 0.013 seconds we can see that the database was already surveyed and queried, with the non-so surprising result that based on the knowledge of our database, Michael Schumacher won the most races, with the total of 97 victories.



Neo4j:

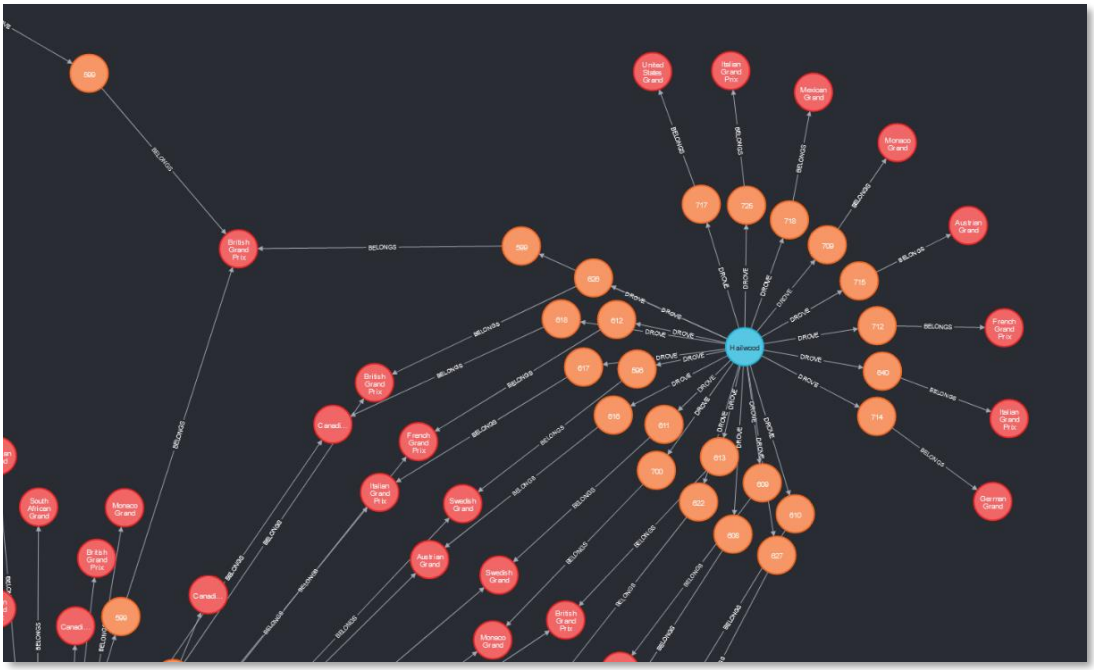
1. **Query: All data, visualize the relations between the 3 node types.**

The first query is quite simple, I just wanted to list showcase the connections:

The code I used will query 250 nodes plus their relations from the Drivers, Results, and Races collections. Here on the second image we can see the relationships between a driver, his results and how a race can be connected to multiple results.

```
MATCH p=(d:Drivers)-[dr:DROVE]-(re:Result)-[be:BELONGS]-(ra:Races)
RETURN p LIMIT 250;
```

Here is a closer look where we can observe the relations:



2. **Query: On each race, how many times did Lewis Hamilton started on?**

For this, we need all the three collections with both of our relationships. On the driver query I filtered down to the name of Lewis Hamilton and asked the database server to list and count each race where he ever started. The count(*) will count all the occurrences from every different record.

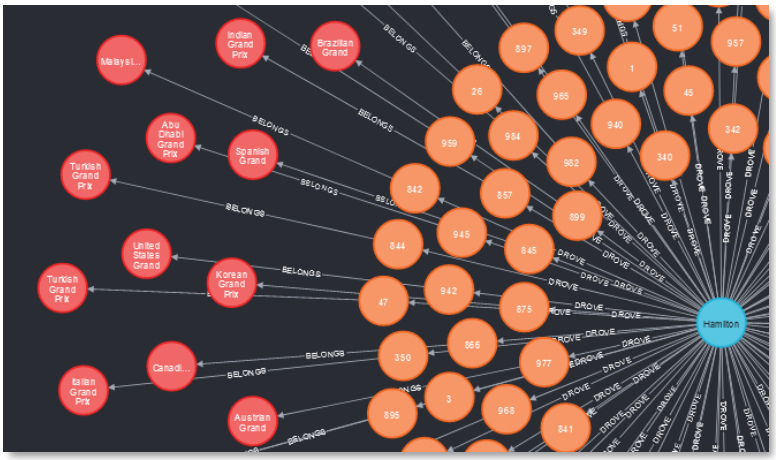
```
MATCH p=(driver {surName:"Hamilton", foreName:"Lewis"}) -[:DROVE]-> (result)-
[:BELONGS]-(race)
RETURN race.name,  count(*)
ORDER BY count(*) DESC
LIMIT 100
```

On the result page we can see all the 25 racetracks Hamilton every competed on. (Image is only an extract)

It is clear, that he started 11 times on the Spanish, Malaysian, Chinese […] Grand Prix, and later, on the 12-14th records, we can notice that the list is really ordered by the number of race events.

Table | Text | Code

	race.name	count(*)
1	"Spanish Grand Prix"	11
2	"Malaysian Grand Prix"	11
3	"Chinese Grand Prix"	11
12	"Singapore Grand Prix"	10
15	"Abu Dhabi Grand Prix"	9
16	"German Grand Prix"	8
17	"European Grand Prix"	7



If we add the p parameter to the return values, we can get this graph, showing us Hamilton (Driver) in the middle, his corresponding results, and the Races (Red) connected to each result.

3. Query: Drivers competing on the Turkish Grand Prix

On the Turkish Grand Prix, F1 race was organized 7 times based on this dataset. Let's get those drivers who competed against each other at least twice.

```
MATCH p=(race {name: "Turkish Grand Prix"}) <- [b:BELONGS]-(result) <- [d:DROVE] - (drivers)
WITH drivers, count(d) as rel_count
WHERE rel_count > 1
RETURN drivers.surName, rel_count
```

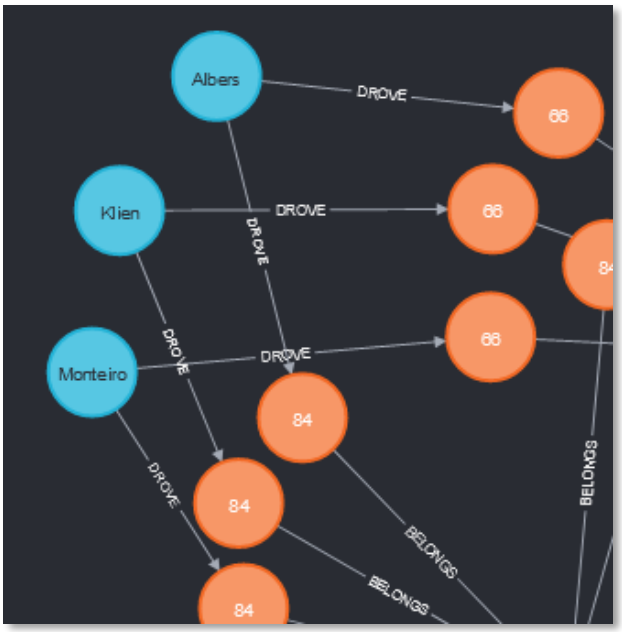
Table | Text | Code

	drivers.surName	rel_count
1	"Petrov"	2
2	"Glock"	4
3	"Buemi"	3
4	"Massa"	7
5	"Kobayashi"	2
6	"Kubica"	5

Form the output, we can see, that we have 34 drivers who have already competed against each other in the Turkish Grand Prix.

The code here filters the races to the given name and counts those driver nodes which has more then 1 relation towards them. As a driver can have multiple nodes (results) connected to them, and results connected to races, it is enough to count the relations connected to the driver nodes.

Visualization of the example drivers:



32	"Monteiro"	2
33	"Albers"	2
34	"Karthikeyan"	2

Comparison of the used DBMSs:

Analysis of the differences, benefits, and disadvantages between the three database management systems (Oracle and selected two NoSQL)

Pros:

- MongoDB
  - MongoDB is very flexible with document schemas, as it stores everything in JSON.
  - Another huge positive aspect is that it has a change-friendly design. This means, if we modify one of our tables, or datatypes, it can be done relatively easily.
  - It also provides powerful querying and analytics, as I showcased it in the queries section.
  - It is easily scalable and provides native code-data access.
- Neo4j
  - Neo4J is one of the first graph-based database server which aims to connect everyone to graphs. Therefore, it has the biggest community.
  - Thanks to native graph storage, its performance is high.
  - Querying and data writing is incredibly fast.
  - Highly protected data integrity.
  - Easy to load the data inside the database. We only need the CSV files and a few lines of codes.
- Oracle
  - Offers a ton of functionalities:
  - Amazing Import and Export tools are provided.
  - Simple UI + Autocomplete feature.
  - Supports various kind of DB not only SQL.
  - Commit and Rollback options.
  - System tables are recording everything what users need.

Cons:

- MongoDB
  - No transactions
  - No collection connection is possible
  - Memory is limited
- Neo4J
  - It has optimization issues. For example, Cypher often gets super slow after a few queries.
  - Though the performance is good on a small dataset it requires a well configured server for a large dataset.
  - Graphical representation for less complex dataset is good but for complex dataset in which more than 10 relation possible graphs are not good.
- Oracle
  - There are fewer features compared to other software available in market in terms of the free software
  - Good quality reporting, sorting is not available.
  - Often Lags upon heavy usage
  - Sometimes it randomly gets disconnected from remote database.

Conclusion:

In the case of Graph databases, every node is explicitly connected with its neighbours, while in the case of relational databases, the relationships between elements are only there in the form of primary and foreign keys. Graph databases and databases using a JSON style storage are schema independent, scalable.

Additionally, for graph databases, data modelling is very strong, as they are more general then other regular databases, but what is really worth mentioning is that query times on joint data is many times faster than RDBMS systems. On the other hand, it is quite hard to imagine everything in graphs and relationships.

Other requirements:

Include all scripts and codes you used to load and query the data. **DONE**

Description: Store the data stored in the Oracle database or a properly specified part of it (**at least the data of two tables**) in two of the learned NoSQL databases. Write 3-3 complex queries on the stored data for the two NoSQL DBMSs! Describe the differences, benefits, and drawbacks between the three database management systems (Oracle and the two selected NoSQL) through specific examples. **DONE**