Univerza v Mariboru

Sabancı Üniversitesi

WORLD RECIPES

From raw data to an interactive map

Author: Matjaž Moser, University of Maribor

Supervisor: Dr. Kamer Kaya, Sabanci University

ISTANBUL
2015

Abstract

In this text we present the steps that were taken from acquiring a dataset, extracting information and in the end visualise it in an interactive way. Along the way we decided to make a performance comparison of Neo4j and MySQL on our dataset. Neo4j is a member of the NoSQL movement and is in its essence a graph database. It is queried with Cypher, a query language specific for Neo4j. On the other hand MySQL is the god old relational database management system and to communicate with the data, traditional SQL is used.

In addition, we also provide a comparison and short explanation of three different methods for calculating cuisine similarities, that in essence return the same result but different execution times. We try to explain why this difference occurs.

Contents

INTRODUCTION AND MOTIVATION

It's no mystery that today's world is governed by data. This data has many forms and shapes but only one size- it's big. The challenge of today is how to govern this data, how to store it efficiently and how to bring it to the user as fast and as easy as possible. Furthermore, mining this data is like finding a needle in a haystack.

NoSQL refers to a mechanism that provides storage and retrieval of data in a form other than tables of the conventional relational model. NoSQL databases are increasingly used in big data and real-time web applications. These systems are sometimes have been called (one in many names) "Not only SQL" so to emphasize that they may support SQL-like query languages (Wikipedia, 2015).

Sahil Bahl (Sahil, 2014) concluded that NoSQL is a category of databases which provide flexibility and scalability. In such databases data is not stored in fixed schemas – it is this flexibility that makes NoSQL databases best suited for huge web – scale applications. The need for such systems clearly exists.

We can observe obvious trend that conforms the statements above. Google uses BigTable. Facebook uses MySQL with some high level enhancements in combination with Cassanra. Amazon uses DynamoDB. Youtube, owned by Google, uses BigTable and LinkedIn uses Voldemort. All of these companies are the leaders in their game and, daily, they produce vast amounts of various types of data. They all use NoSQL systems or relational with high level enhancements.

But traditional relational database systems are still the dominant force in the corporate world and they will probably stay for some time. According to DB-engines (Andlinger, 2013) share of RDMS is 77%. This is the share of all databases in their listings. This is because of its maturity, long tradition and standardisation. But SQL market share is slowly decreasing (Asay, 2015) on the expense of MySQL. Oracle and SQL Server market shares remain roughly constant in the past few years.

The future remains a mystery but one thing is for sure – the dominant share of SQL is slowly decreasing.

The main objective of this project was to explore some possibilities of exploration that raw data provides. We didn't reinvent the wheel. Emphasize is not on the result but on the approach. We have shown that, out of raw structured in CSV files, we can extract some valid information. That information can then be presented in an interactive way to the general public. Given the know-how, it is simple but along the way we overcame many obstacles.

Given the fact that data is growing rapidly daily and there are thousands of similar opportunities and aspects of data that cannot even be imagined gives us plenty of motivation for work and also fills us with curiosity - what the future holds for us.

1. The data

Here we present the procedure of how we acquired the dataset and how we manipulated our data. We present the steps of importing it into our databases and querying it. At the end of the chapter we also provide a comparison of the two systems used based on literature and our own observations.

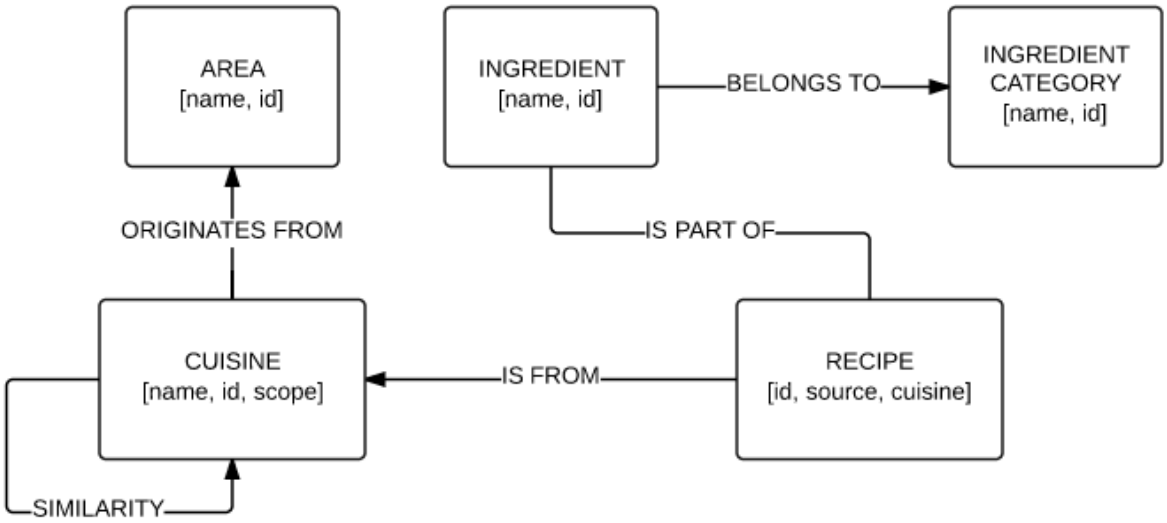
1.1. Graph database – Neo4j

The data was acquired from a blog owned by Rik Van Bruggen (Bruggen, 2015) and was made public. Thanks to the author of the blog the dataset was already processed to some extent. Some minor changes had to be made for it to be used as input to our databases:

- Due to some technical difficulties (still unresolved) we were unable to use the Neo4j import tool provided with the database. This tool makes large imports to the database fast and easy but it was unavailable to use so we used Cypher to import the data. A big portion of the initial data was in one big file containing ingredients, categories, areas, cuisines and compounds. Each line was labelled so it could be determined where it belongs. This label caused difficulties (probably because of our method of import) so we decided to partition the dataset and in the end instead of one file we had 5.
- Some of the data was also corrupted or duplicated, especially the cuisines part. Just to give an example, there were cuisines named Italy and Italian (both lower and uppercase) which all had to be joined into one and this change had to be reflected in all other CSV files to keep the data consistent (there were several such cases).
- Compounds were removed from the dataset because they didn't provide anything useful to us and they just clogged up space in the database

At the end of the cleaning process we had 8 CSV files ready to be imported to the database. These files included data for the nodes as well as data for the relationships between the nodes.

Demonstration of the data model is given in the picture below:



Similarity is not part of the original data model and was added once it was calculated.

As we mentioned above we used Cypher to import our data, instead of Ne4j import tool. The dataset was cleaned, partitioned and ready to be imported. Steps taken for a successful import are given below (with examples):

Create indexes for our data:

```
CREATE INDEX ON :AREA(name);
CREATE INDEX ON :AREA(ID);

CREATE INDEX ON :INGREDIENT(name);
CREATE INDEX ON :INGREDIENT(ID);

CREATE INDEX ON :CUISINE(name);
CREATE INDEX ON :CUISINE(ID);
CREATE INDEX ON :CUISINE(scope);

CREATE INDEX ON :INGREDIENT_CATEGORY(name);
CREATE INDEX ON :INGREDIENT_CATEGORY(ID);

CREATE INDEX ON :RECIPE(ID);
CREATE INDEX ON :RECIPE(source);
CREATE INDEX ON :RECIPE(cuisine);
```

Once indexes were created we could start importing the nodes and the relationships between them. The first example depicts the creation of nodes and the second one depicts the creation of a relationships:

```
USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:.../RECIPES.csv" AS csvLine
CREATE(rec:RECIPE {ID:toInt(csvLine.id), source:csvLine.source, cuisine:csvLine.cuisine});
```

According to the Neo4j manual (Neo4j, 2015) periodic commit, given the parameter, commits the transaction when it reaches the value specified with the parameter. This is used for large imports with one single Cypher query to avoid out of memory errors. In our case the transaction gets committed every 1000 rows. The statement then reads the data from a CSV file and creates nodes with attributes given in that file:

```
USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:/.../RECIPES.csv" AS csvLine
MERGE(cuis:CUISINE{name:csvLine.cuisine})
MERGE (rec:RECIPE {ID:toInt(csvLine.id)})
CREATE (rec)-[:IS_FROM ]->(cuis)
```

The statement above is used for relationship import. We again used the same approach for periodic commit and CSV file read. The statement then merges the nodes (the nodes are read from the file as a source and a target for the relationship). According the Neo4j manual (Neo4j, 2015) merge merges existing nodes and bind the relationship to them or creates new ones if they don't exist. We could tell that our data is consistent and clean because, using the statement above, only relationships were created and no additional nodes.

Once all our data was imported it reflected the data model introduced before. The main idea of the project was to use Neo4j and Cypher for all our transactions with the database and the dataset but that proved to be impossible due to the following reasons:

- Neo4j is designed to perform, without any trouble, with tenths of billion nodes so from the perspective of the database it shouldn't be a problem to handle our little dataset. Trying to perform complex queries we were often faced with out of memory errors. Trying to solve the problem we increased the memory available to the database but the problem persisted (along the way we crashed the database several times). We can probably conclude that the machine, where the database was running, was not suitable.
- For our transactions and communication with the database we used the browser and the visual tool provided by Neo4j. This visual tool is awesome and designed very nicely but due to limitations in its design, it can only output 1000 rows at most. This came as a problem to us so we decided to go around it.

Final state of our database is presented in the table:

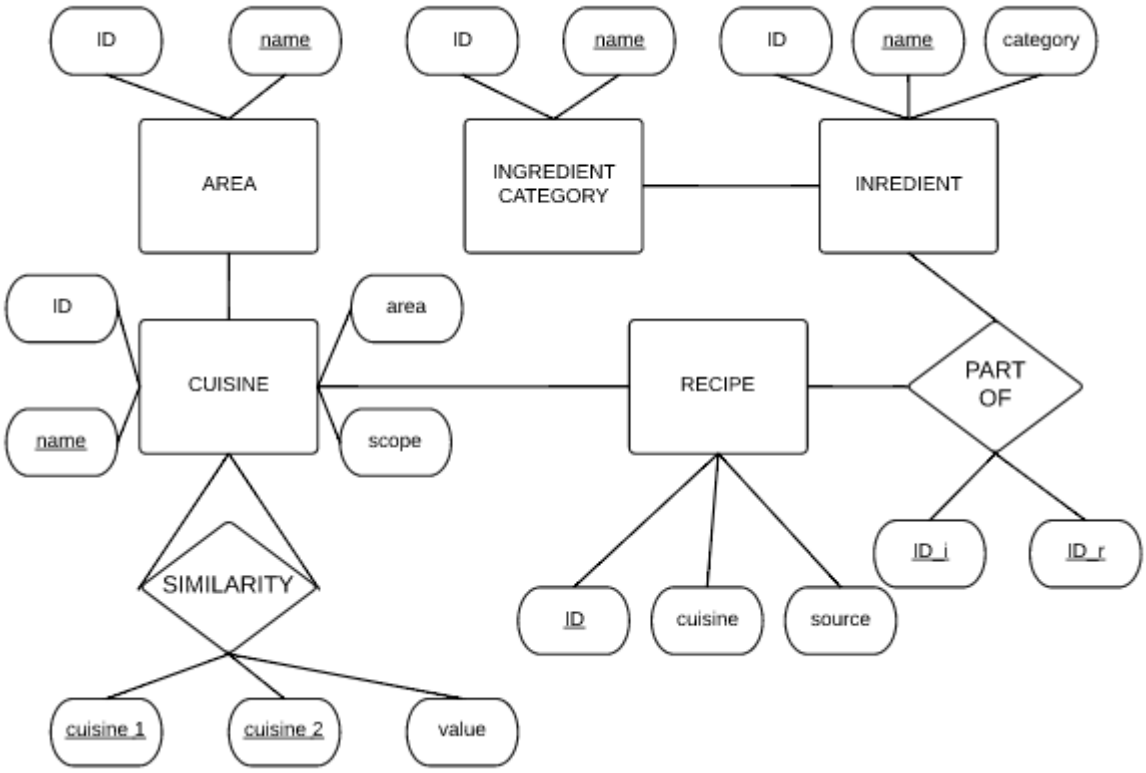
Number of nodes	59300
Number of relationships	540983
Number of properties	2690
Size	94 MB

Once we discovered these limitations we decided to do all the heavy lifting with Python. The work done in Python is discussed in the data chapter.

1.2. Relational database – MySQL

For the purpose of testing the performance of the database we decided to duplicate the schema from Neo4j to MySQL database. All the data sources remain the same and number of columns in a table equals the number of properties in a node (unless foreign key constraint prevents that). Details about the comparison will be given later in this chapter.

Given the data model from above we duplicate it in a simplified ER diagram shown below.



Similarity is not part of the original data model and was added once it was calculated.

For this purpose we used MySQL 5.6 in combination with wamp server for visual representation of the data in phpMyAdmin. All the queries were performed using MySQL command line. For creating the database SQL was used.

PHPMyAdmin offers a visual tool for creating the dataset but using SQL statements is more straightforward and certainly easier. Tables were created, as they usually are, with a statement below:

```
CREATE TABLE recipe(
    id int,
    source char(20),
    cuisine char(20),
    PRIMARY KEY(id),
    FOREIGN KEY(cuisine) REFERENCES cuisine(name)
);
```

In some tables certain columns are actually redundant but we kept them to make the schema as equal as possible to the one used in Neo4j. We imported the data from the same CSV files as before using this statement:

```
SET foreign_key_checks = 0;
LOAD DATA LOCAL INFILE 'c:/.../INGREDIENT.csv'
INTO TABLE ingredient
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
SET foreign_key_checks = 1;
```

Due to an unexplained error while importing the data (foreign key constraint failure) we set the foreign key check to 0 before importing the data and set it back to 1 afterwards. This error was unexpected. We populated the tables in the correct order. First we populated the primary tables and then secondary ones. By secondary we mean tables that reference other tables via foreign keys.

After the database has been populated we ran some test queries and some of them returned no rows even though the syntax was correct and data was there to be fetched.

The problem was solved by exporting the whole database as a .sql file and altering the import statements. The problem was caused by a “\r” carriage return which is used in MAC operating systems and it is not interpreted properly in Windows. After deleting it and reimporting the database queries worked fine.

The final state of the database is shown in the table below:

Number of tables	7
Number of columns	19
Number of rows	582875
Size	82 MB

1.3. Comparison of database management systems

In this chapter a comparison of both database systems is given, using the same types of transactions in both of them. IN our own conclusions we focused on execution times of queries.

As we mentioned before Neo4j uses its own query language called Cypher and for querying MySQL we used the good old SQL. In this chapter we provide a Cypher query together with its equivalent in SQL and execution times of both queries.

Here we also provide some results from tests done by professionals since our test are not made at appropriate scale and/or complexity to make any significant conclusions.

It is fairly obvious that in the sense of data modelling the Neo4j is much simpler (see pictures of both data models). This property is called whiteboard friendliness (Redmond & R. Wilson, 2012). It basically means that anyone can create a valid data model without the knowledge of diagram techniques used in relational databases. In a usual relational database the data model consist of tables which are linked together with additional tables that represent relationships. The additional table is created if the relationship between initial tables is many-to-many. If the relationship is of type one-to-many the additional table is redundant and an additional columns in introduced to one side of the relationship. Two tables are thus connected using foreign key constraints. Data in a relational database is stored in forms of columns and rows.

In a graph database the data model is represented using a graph with vertices and edges – or nodes and relationships. If we translate this to a relational database, every node is a row and every property of that node is a column. Relationships work in a very similar manner. The most popular form of a graph model is the property graph which is defined by the following characteristics:

- It contains nodes and relationships
- Nodes contain properties (key-value pairs)
- Relationships are named and directed and always have start and end node
- Relationships can also contain properties

In the book Graph Databases (Robinson, Webber, & Emil, 2013) authors point out three practical benefits of graph databases:

- Performance: in contrast to relational databases, where join-intensive query performance deteriorates as the dataset gets bigger, with graph database performance tends to remain relatively constant, even if the dataset grows. This is because queries are localized to a portion of the graph. As a result, the execution time for each query is proportional only to the size of the part of the graph traversed to satisfy that query rather than the size of the overall graph.
- Flexibility: we want to connect data as the domain dictates thereby allowing structure and schema to emerge in tandem with our growing understanding of the problem space, rather than being imposed upfront, when we know least about the real shape and intricacies of the data. The graph model expresses and accommodates business needs in a way that enables IT to move at the speed of business. Graphs are naturally additive, meaning we can add new kinds of relationships, new nodes and new subgraphs to an existing structure without disturbing existing queries and application functionality. The additive nature of graphs also means we tend to perform fewer migrations, thereby reducing maintenance overhead and risk.
- Agility: we want to be able to evolve our data model in step with the rest of our application using technology aligned with today's incremental and iterative software delivery practices. Modern graph databases equip us to perform frictionless development and graceful system

maintenance. In particular, the schema-free nature of the graph data model, coupled with the testable nature of a graph database's application programing interface (API) and query language, empower us to evolve an application in a controlled manner.

If we summarize the properties of a graph data model expressed above we can conclude that graph databases perform faster, they need less maintenance and are able to evolve in a synchronized manner with its purposes. All fair and square - according to the literature.

In the next chapter we will present some conclusions done by other authors when they compared the two database types. Furthermore we will present our own conclusions of tests and observations while working with both Neo4j and MySQL database.

1.3.1. Literature review and preliminary research

Several papers have been already published on this matter. They all seem to have a similar scope of aspects of comparison. There is need for us to dive deeper into those aspects so a summary of the literature is here given, to whom it might concern.

Level of support:

Comparing the level of support between these two database systems somehow makes no sense since relational databases have been around since the dawn of time. Graph databases are a new a concept and the first Neo4j prototype was developed in 2000 and version 1.0 was published in 2010.

We can define the level of support with respect to the maturity of the application. We can assume that more time the application has been in service and production more is known about it and through time, its flaws are repaired. All vendors provide a good support for all relational databases and even if one of them goes belly up the support is still there since all relational databases use the standard SQL.

This cannot be said for Neo4j whose community is relatively small and the majority of support is given by the owners themselves and their manuals published on-line. Third party support is provided by the growing community on forums and wikis. The conclusion is obvious

Security:

Vicknair and others (Vicknair, et al., 2010) have pointed out that relational databases in general and MySQL specially have and extensive built-in multi user support. Many graph databases on the other hand lack such support. Neo4j is among them. It forces all user management to be handled at the application level. By extension, then, Neo4j does not have any built-in security support. It assumes trusted environment.

That doesn't sound very good. Assuming anything in today's world, especially, digital security is bananas.

Ease of programming:

As mentioned before most of relational database systems use SQL as the standard language for (S)CRUD operations. Graph databases are not (yet) standardized to such extent. Neo4j for example uses Cypher to manipulate and communicate with the data. Cypher has its own API and is relevant only for Neo4j. The fact that there is no standard language makes transitions between DBMS more complex.

Vicknair and others (Vicknair, et al., 2010) have pointed out that the actual ease of programming is task-dependent. Graph traversals are fairly simple in Neo4j and its API contains methods for doing so. MySQL graph traversals are much more complicated and can involve looping or recursing through the graph possibly execution multiple expensive joins along the way.

Flexibility:

This was mentioned before as one of the strong benefits of the graph based DBMS and all other NoSQL systems. The absence of a fixed schema makes this possible. RDBMS is obsolete without the schema providing the framework and structure of the whole system. This schema is very rigid and it is proven very difficult to extend or alter it in any kind

Batra and Tyagi (Batra & Tyagi, 2012) in their paper present the flexibility aspect of comparison as: although relational databases are more mature and secure, but their schema is fixed, which makes it difficult to extent these databases and less suitable to manage ad-hoc schemas that evolve over time. However in graph databases, there is no need to restructure the entire schema every time a new relationship is added, only new edges and nodes are added to the graph. With this the authors conclude that Neo4j has an easily mutable schema while relational databases are less mutable

Queries:

All of the authors also provided some query execution times. Here we will describe their ideas, the types of queries and their own conclusions since their queries have nothing to do with our dataset.

Batra and Tyagi (Batra & Tyagi, 2012) used MySQL 5.1.41 and Neo4j Community 1.6. They queried the MySQL database with PHP and they ran three prewritten queries on both databases. All of their queries were of traversal nature, friend-of-friend pattern. They concluded that Neo4j outperformed MySQL in execution times of those queries.

Vicknair and others (Vicknair, et al., 2010) had a more extensive approach to the tests. They build 12 different databases with different random values stored (integers, 8K strings, 32 K strings). All of their data was random. They wrote two types of queries: structural (traversing the DB) and data queries.

They had concluded that both databases performed adequately in fields that they were designed for. MySQL was faster in data oriented queries and on the other hand Neo4j was faster in traversing the database.

Basically anything written about graph databases emphasises the fact that they are fast in traversing the data.

Similar tests were also performed by ourselves using, to data, relative transactions. We were not testing or looking for anything particular. We were just wondering how a relational database would compare to graph oriented one in the case of our dataset Results are presented in the next chapter.

1.3.2. Internal conclusions and tests

Data model and database creation:

In the data part we already presented the data model that was used in both data models. In theory and in content, both are the same but they clearly vary in complexity. From the perspective of a viewer, that is not familiar with ER diagrams, the relational one seems quite confusing but on the other hand, the data model that represents graph database is fully understandable even to the general public. By the authors of Seven databases in seven weeks (Redmond & R. Wilson, 2012) this property is called whiteboard friendliness, as mentioned before.

From the practical viewpoint of the domain creation, they are both the same but as mentioned drawing a relational data model requires knowledge of ER diagrams and graph is something natural, sort of instinct.

In our case, both databases are very simple with no complex relations or special constraints so in both cases the actual implementation of the database was simple and no relevant differences were spotted.

Data import:

Dataset used, didn't change a lot beyond the initial changes described in the data chapter. For both databases we used the same files containing comma separated values (CSV). Data import caused some issues with MySQL which were already described and are henceforth given in summary:

- Foreign key constraints failed every time. To solve this we had to disable the foreign key check and enable it again. Since the scope of our project was fully educational this did not pose any problems but in the real world things are different and such actions are not acceptable.
- Another problem was caused by a "\r" carriage return which is used in MAC operating systems and it is not interpreted properly in Windows. Every row in every CSV file contained this carriage return symbol which had to be deleted in order for the database to function properly. We are aware that this problem is solemnly related to our data but since we are giving a comparison it is worth mentioning that Neo4j had no problem with that.

We are aware that the second problem is unrelated to the actual tool used but it's a product of "unclean" data but since we are giving a comparison – Neo4j had no problem with that.

Usage:

Both databases were generally used through GUI within the browser. Neo4, by default, provides a visualisation and querying interface which is great and useful for exploring the data**.** For creating the MySQL database and populating it (and also other simply queries) MySQL command line was used.

For all other communication with the data wamp server and phpMyAdmin were used. From the viewpoint of interaction with the dataset we cannot point out any technical differences, at this level of comparison. We can only conclude that Neo4j web based visualisation tool is more user friendly that the crude nature of MySQL command line and phpMyAdmin old-school tables look.

Queries.

As mentioned before we also ran some tests on our two databases. For this purpose we wrote 10 queries that vary in complexity and scope. Queries are designed to test both traversing and data acquisition.

Cypher is the query language that is used in Neo4j, it is native to it. According to the Neo4j manual (Neo4j, 2015), Cypher is declarative graph query language that allows for expressive and efficient querying and updating of the graph store. Cypher is relatively simple but still very powerful language. Cypher is designed to be a humane query language. We guess they were trying to say that it is intuitive and it is.

Structured Query Language or SQL is well known to everybody so no time will be lost here.

All queries were ran on a household portable machine with the following relevant system specifications:

- PROCESSOR: Intel core i3-3217U CPU 1.8 GHz
- RAM: 4 GB
- OS: Win 8.1 64 bit
- MySQL 5.6.26
- Neo4j Community 2.2.3
- Google Chrome version 44.0.2403.157

The queries were not ran in isolation. During the tests normal system processes were running but no user programs (except for browser). The computer was also connected to the internet so interferences are not excluded (software updates). The queries that we ran are listed below and the queries themselves are available in appendixes (with their respective execution times):

1. Name and ID of the most used ingredient in Africa
2. All areas with number of recipes, cuisines and ingredients in them
3. Top five cuisines for Israel
4. Mutual ingredients between Bangladesh and Pakistan
5. Count all entities in the system
6. Number of ingredient categories used in areas
7. Percentage of recipes using alcohol in a specific area.
8. Percentage of fruits used in recipes for all cuisines.
9. Sum of all numeric values (ID’s)
10. Number of recipes using alcohol in all areas.

Results of the query average times are shown below. Execution time in Neo4j is presented in millisecond, MySQL outputs it in seconds. Full table is available in appendixes.

	1	2	3	4	5	6	7	8	9	10
MySQL	14.17	5296.8	0.84	3.13	0	5108.66	43	9726	0	3842.159
Neo4j	103.500	4027.9	38.4	81	7978.9	6885.4	18181	0	7534	511.9

As mentioned before, these tests have not been done to prove anything but just to compare both databases in our specific case.

The first thing that comes as a surprise is that MySQL was not able to perform queries 5 and 9 which are basically just summing and counting the elements. First thought was that there is something wrong with the database or the tests but after checking all the values and rerunning the queries, they seem to be valid but very strange and probably wrong but we were not able to find the source of this in the time period given to us.

The second most obvious observation is how much faster MySQL is. Before starting the queries I assumed that Neo4j will be faster than or at least as fast as but that was proven wrong. If we observe results from queries 1, 3, 4, 7 we can easily see the big gap in execution times.

This can be simply explained by the level of optimisation join operations have experienced through years. One could quote the literature and claim that Neo4j is faster in case of joins and traversals. Yes, that might be and possibly is true on a bigger (much bigger) scale and on more complicated datasets, but in our small and relatively simple dataset that was proven wrong. Furthermore, our queries are simple and do not really take advantage of Neo4j's strengths (path based querying such as shortest path).

Neo4j completely failed in query 8, which is very similar to queries 7 and 10. It is similar to query 7 because it includes the WITH clause, this allows piping results from one query to another. But 7 only deals with one portion of the data while 8 traverses the whole data model. If we compare it to 10 we see the similarity – it also traverses the whole data model but completes (and much faster than MySQL), the difference is that query number 10 doesn't use WITH clause. Using this we can conclude that piping results from one query to another is very slow and expensive, it should probably be avoided.

In case of queries 2 and 6 both systems were relatively equal (comparing the differences in other queries). Both of those queries go through the whole data model a count entities. One would expect Neo4j to be faster (claims by literature) but on this scale the difference cannot be observed, as mentioned before.

1.3.3. Conclusions

Without taking into consideration the results obtained from other authors, observing only experiences gained from our work, we can conclude that:

- Modelling and implementation of a Neo4j database is easier and faster – this might come out strange but I have been in contact with SQL and relational databases for some time now while I first met with NoSQL and graph databases just prior to the start of this project and I think that from a perspective of a student or someone who is trying to learn (even company staff maybe) Neo4j wins the day.
- All compliments to the creators of Cypher. At first glance the syntax is funny and makes little sense but once you get the hang of it really works great. It is easy to visualise the path of the query and execution patterns. The only thing terribly missing is a GROUP BY clause. Absence of this makes some queries complicated for no reason.
- The visualisation capabilities of Neo4j's GUI are great. It is hard to say more. If you compare it to the rough nature of phpMyAdmin it is obvious. This scores some extra points to the user friendliness – in production systems this wouldn't really matter.
- The flexibility of Neo4j is a plus. Adding nodes, relationships and properties to both of them is no problem. While in the case of MySQL if we had to make major changes to the whole schema it was sometimes easier just to delete the whole database and start from scratch.
- Support for both database systems is available in working just fine. All of the problems were solved with Neo4j manual or the help with growing online community. SQL (being the de-facto standard for many years) has wikis and online tutorials all over the place so obtaining and getting help was no problem. This might pose as a counter argument to the maturity and level of support discussed earlier. We should point out that the problems we encountered were not as complex as they might be in real-time and real-life systems so the arguments from before are still valid.

To give any judgment on which DBMS is better would be arrogant and plain nonsense. Deciding on the winner would require more time and a more complex system of tests. We can only conclude that both systems performed well – one is not absolutely better than the other but both have strengths and weaknesses but from a perspective of a student such as myself in taking into the consideration the nature of our project we can say that Neo4j was more suitable and more friendly to work with.

Times are turbulent in the data world. In the introduction we mentioned the growing rate of data. This phenomena can be described with three V's: velocity, volume and variety, and only time will tell which kind of systems will "survive" (this being the battle of SQL vs NoSQL).

2. Mining the data

In this chapter we will provide an insight to our calculation methods. We provide detailed descriptions of the methodology and of the function written and used in the process,

2.1. Introduction

The script is written in Python 3.4. For all computation Numpy module is used and for importing the values, CSV module. Timing is done with Time module. The IDE used are Anaconda and PyCharm. All variations of the script consist of four functions. The variations vary in purpose in approach.

First three methods were used for calculation the distances between recipes and they serve as a training ground for the last method which calculates the actual distance between the cuisines.

The similarity between two recipes is defined by the equation:

$$s(\mathrm{r_i}, \mathrm{r_{i+1}}) = \frac{Ir_i \ \cap \ Ir_{i+1}}{Ir_i \ \cup \ Ir_{i+1}}$$

The numerator is the count of mutual ingredients in the ingredient sets of two recipes. The denominator represents the union of both recipes.

The similarity between two cuisines is then defined by the equation:

$$s(C_i, C_j) = \frac{\sum d(\mathrm{r_i}, \mathrm{r_j})}{x * y} \ ; r_i \ \ni C_i \ and \ r_j \ \ni C_j$$

Where x is the number of recipes in C_i and y is the number of recipes in C_j. The similarity is further calculated as:

$$d(C_i, C_j) = \frac{1}{s(C_i, C_j)}$$

For the purpose of explaining the process of calculating the distance between two cuisines we will use the example matrixes below:

$$C_i = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \qquad C_j = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

In our example both cuisines contain three recipes and the total ingredient set is of size 3 (maximum number of ingredients that a recipe can contain is 3). From the matrixes we can compose the recipes arrays (ingredient sets of individual recipes) this is the approach of second and third method:

$$r_{i1} = [1 \quad 0 \quad 1\,] \ r_{i2} = [1 \quad 1 \quad 0\,] \ r_{i3} = [1 \quad 0 \quad 1\,]$$

$$r_{j1} = [1 \quad 1 \quad 0\,] \ r_{j2} = [0 \quad 1 \quad 0\,] \ r_{j3} = [0 \quad 1 \quad 1\,]$$

Ingredients in recipes are marked with indexes of array elements. Using this we translate the arrays as follows:

$$r_{i1} = [\,0 \quad 2\,] \ r_{i2} = [\,0 \quad 1] \ r_{i3} = [\,0 \quad 2]$$

$$r_{j1} = [\,0 \quad 1\,] \ r_{i1} = [\,1\,] \ \ r_{i3} = [1 \quad 2]$$

Once we obtain these values we can easily compare the recipes between each other. While doing that we store counters for number of ingredients in both recipes and a counter of mutual ingredients.

Description of common functions:

Array filler:

This function reads the data from a CSV file. Data from this file is stored in an array of size (57691, 1531). Being the number of recipes in our dataset and the number of ingredients in the same dataset. The array is initialized as all-zero array (contains only zeros). This function marks the ingredients by changing the value from 0 to 1.

```
def array_filer():
    with open('input_file.csv', 'r') as csvfile:
        reader = csv.reader(csvfile)
        for row in reader:
            ingredient, recipe = row
            myArray[int(recipe), int(ingredient)] = 1
        csvfile.close()
```

Array mutilator:

This function is used in all methods but with different approach. Its main purpose is to delete rows or columns if their total sum is 0 (empty rows and/or columns)

```
def array_mutilator(the_array):
    new_array = the_array[~numpy.all(the_array == 0, axis=1)]
    new_new_array = new_array[:, numpy.sum(new_array, axis=0) != 0]
    return new_new_array
```

The array mutilator in the shape shown above is used only in the first method. All other methods use a modified version which only eliminates empty rows since index values of columns are important for further calculations.

Indexer: this is a function that takes the overall array as an input parameter and from it creates two new arrays. An array of indexes and an array of or recipes. The array of indexes contains the starting indexes of recipes in recipes array (see example):

```
def indexer(the_array):
    ALL_RECIPES_ARRAY = []
    RECIPE_INDEXES_ARRAY = [0]
    next_recipe_index = 0
    for recipe in range(0, the_array.shape[0]):
            CURRENT_RECIPE_ARRAY = []
            value = 1
            for column in range(0, the_array.shape[1]):
                if value == the_array[recipe, column]:
                    CURRENT_RECIPE_ARRAY.append(column)
            next_recipe_index += len(CURRENT_RECIPE_ARRAY)
            RECIPE_INDEXES_ARRAY.append(next_recipe_index)
            ALL_RECIPES_ARRAY.append(CURRENT_RECIPE_ARRAY)
    return RECIPE_INDEXES_ARRAY, ALL_RECIPES_ARRAY
```

Let X be an array of indexes and Y an array of recipes (based on example above for C_i):

$$X = [\,0 \quad 2 \quad 4\,]$$

$$Y = [\,0 \quad 2 \quad 0 \quad 1 \quad 0 \quad 2\,]$$

The X array above indicates that first recipe starts at index 0, second one at index 2 and third one at index 4 in the Y array.

Calculator:

This is the function that actually makes the calculation. Given the equations given above, for distances between recipes and cuisines, we calculate those values and store them in a CSV file of results. All methods have different calculator functions and they will be described separately.

The general purpose of functions disused is the same in all methods but some details vary from the approach used. The differences or deviations from the general approach are given bellow, for each method used.

2.2. First method

Method one is a method that, not like other two, does not reform the big array into two arrays but deals with just one from the beginning. The array gets filled and mutilated using standard methods. The only difference is the calculator function shown in the picture below.

```
def calculator(the_array):
    output_file = open("output_file.csv", "a")
    for x in range(0, 11):
        for y in range(0, the_array.shape[0]):
            if y > x:
                mutual_counter = 0
                union = 0
                distance = 0
                RECIPE_1 = []
                RECIPE_2 = []
                for column_1 in range(0, the_array.shape[1]):
                    if 1 == the_array[x, column_1]:
                        RECIPE_1.append(column_1)
                for column_2 in range(0, the_array.shape[1]):
                    if 1 == the_array[y, column_2]:
                        RECIPE_2.append(column_2)

                for z in range(0, len(RECIPE_1)):
                    if RECIPE_1[z] in RECIPE_2:
                        mutual_counter += 1
                if mutual_counter > 0:
                    union = len(RECIPE_1) + len(RECIPE_2) - mutual_counter
                    distance = mutual_counter / union
                    output = str(x) + "," + str(y) + "," + str(distance) + "\n"
                    print(output)
                    output_file.write(output)
    output_file.close()
```

As we mentioned before, the input parameter is a processed overall array. The main idea is to compare each row (recipe) with every other one. Repetitions are meant to be avoided.

For each row, and the next one, two recipe arrays are created and then compared to find mutual elements and thus produce the mutual ingredient counter used in further calculations.

This mutual ingredient counter is used to calculate the union and the union is then used to calculate the distance. This distance is then written to the output file in CSV format.

2.3. Second method

Second method takes the big overall array and uses the indexer function to create an array of recipe indexes and an array of recipes. After those two arrays are created the overall array is deleted to release the memory occupied by it. The calculator function takes the newly created arrays as input parameters.

The calculator function in shown in below and will be further discussed:

```
def calculator(RECIPE_INDEXES_ARRAY, ALL_RECIPES_ARRAY):
    output_file = open('output_file.txt', 'a')
    for x in range(0, len(RECIPE_INDEXES_ARRAY)):
        for z in range(x + 1, len(RECIPE_INDEXES_ARRAY)- 1):
            mutual_counter = 0
            index_1 = RECIPE_INDEXES_ARRAY.index(RECIPE_INDEXES_ARRAY[x])
            index_2 = RECIPE_INDEXES_ARRAY.index(RECIPE_INDEXES_ARRAY[z])
            index_3 = index_2 + 1

            comparisson_recipe_1 = ALL_RECIPES_ARRAY[slice(index_1, index_2)]
            comparisson_recipe_2 = ALL_RECIPES_ARRAY[slice(index_2, index_3)]

            for y in range(0, len(comparisson_recipe_1[0])):
                if comparisson_recipe_1[0][y] in comparisson_recipe_2[0]:
                    mutual_counter += 1

            if mutual_counter > 0:
                union = len(comparisson_recipe_1[0]) + len(comparisson_recipe_2[0]) - mutual_counter
                float(mutual_counter)
                float(union)
                distance = mutual_counter / union
                float(distance)

                output = str(x) + ',' + str(z) + ","+ str(distance) + '\n'
                print(output)
                output_file.write(output)
    output_file.close()
```

Step-by-step execution of the function:

- Open output file
- The first loop loops from 0 to the length of the indexes array (which is the number of recipes from our dataset. The second loop loops the same but is always incremented by one to avoid comparison of same recipes
- Values x and z are stored as indexes and are further used to slice up the recipes arrays to proper chunks. For slicing we use three indexes to determine the start and the end of the first recipe as well as the start and the end of the second recipe. This approach is clearly explained with the declaration of two comparison_recipe lists.
- The two lists are then compared element by element and for each match the mutual counter is incremented. Using the mutual counter value we calculate the union and the distance
- Write the output to the file and close the file

2.4. Third method

This method is very similar to the previous one. It incorporates the same functions with same outputs but as it is shown further in this text, is much faster.

Just as the second method, this one also uses two arrays created by the indexer function. These arrays are obviously of the same size and contain the same data as in the method before. The only difference is the calculator function which was improved substantially. The calculator function from this method is shown in the picture:

```
def calculator(RECIPE_INDEXES_ARRAY, ALL_RECIPES_ARRAY):

    output_file = open('output_file_mutual_method_2.txt', 'w')
    dense_array = numpy.zeros(57691)
    for x in range(0,len(dense_array)):
        dense_array[x] = -1;

    for x in range(0, len(RECIPE_INDEXES_ARRAY)):
        comparisson_recipe_1 = ALL_RECIPES_ARRAY[x]
        xsize = RECIPE_INDEXES_ARRAY[x+1] - RECIPE_INDEXES_ARRAY[x]
        output = str(x) + ":"

        for k in range(0,len(comparisson_recipe_1)):
            dense_array[comparisson_recipe_1[k]] = x;

        for z in range(x + 1, len(RECIPE_INDEXES_ARRAY) - 1):
            comparisson_recipe_2 = ALL_RECIPES_ARRAY[z]
            zsize = RECIPE_INDEXES_ARRAY[z+1] - RECIPE_INDEXES_ARRAY[z]

            mutual_counter = 0
            for k in range(0,len(comparisson_recipe_2)):
                if dense_array[comparisson_recipe_2[k]] == x:
                    mutual_counter += 1

            union = xsize + zsize - mutual_counter
            distance = mutual_counter / union
            output = output + " " + str("%.4f" % distance)

        output = output + "\n"
        output_file.write(output)
    output_file.close()
```

Step-by-step execution of the function:

The function that calculates the distances is almost identical to the one used for calculating the distances between cuisines and is a much improved and faster version of the second method.

As input it takes two arrays, indexes and recipes that were created by indexer function. This function uses a different method of slicing the recipes array for comparison which makes more efficient and thus faster as we will see at the end of this chapter.

The function also uses an array that acts like a buffer for recipes. The variable is called dense array and serves as a deposit for a recipe that is currently in process. Once the current recipe is processed a new one is read to the dense array. Using this dense array as a buffer we managed to get a big speed improvement mainly because the recipe currently processed is initialized only once per cycle. In previous method the processed recipe was initialized for every recipe in the recipes array which produced a big overhead and a slow execution.

2.5. Similarity calculation and comparison

2.5.1. Similarity calculation

This method is the one that was actually used for calculation of the distances between cuisines. The previous methods focused on recipes regardless of the cuisine the recipe originates from. Since this is the method that was used and it is different from the other three we will discuss it in detail. All the relevant functions are discussed below:

Partition is a function that takes the input file containing all recipes and partitions it into 53 smaller files (the number of different cuisines in our dataset). The output of this function is an array of all cuisines which is then used to open files one by one in next steps.

```
def partition():
    with open('data2.csv', 'r', ) as csvfile:
        reader = csv.reader(csvfile)
        cuisines_array = []
        for row in reader:
            file_name = ""
            recipe, ingredient, cuisine = row
            if file_name != cuisine:
                file_name = "cuisine_" + str(cuisine) + ".csv"
                with open(file_name, 'a') as csvfile_partitioned:
                    csvfile_partitioned.write(recipe + ',' + ingredient + ',' + cuisine + '\n')
                if cuisine not in cuisines_array:
                    cuisines_array.append(cuisine)
    csvfile.close()
    csvfile_partitioned.close()
    return cuisines_array
```

The function takes every row from the input file and unpacks it into three variables (recipe, ingredient and cuisine). For every row in the file it checks if the cuisine variable is stored in the cuisines array and if not the value gets appended and a row is written to the file. When the value of cuisine changes the procedure gets repeated with a new value added to cuisines array and a new file is created. This file contains data about recipes from the next cuisine

The caller function takes the cuisines array, provided by partition, as input parameter and loops through it in a nested loop. In the process of traversing through all cuisines files are opened in pairs and recipes in those files are compared. Distance is calculated by the same calculator function as discussed in method 3. This approach has proven itself to be the fastest (see table below) and thus chosen as the appropriate method for calculation

```
def caller(cuisines_array):
    for x in range(0, len(cuisines_array) - 1):
        for y in range(1, len(cuisines_array)):
            file_name_1 = "cuisine_" + str(cuisines_array[x]) + ".csv"
            file_name_2 = "cuisine_" + str(cuisines_array[y]) + ".csv"
            with open(file_name_1, 'r', ) as csvfile_1:
                with open(file_name_2, 'r', ) as csvfile_2:
                    myCuisine_1 = numpy.zeros((57691, 1530), dtype='int')
                    myCuisine_2 = numpy.zeros((57691, 1530), dtype='int')
                    reader_1 = csv.reader(csvfile_1)
                    reader_2 = csv.reader(csvfile_2)
                    for row in reader_1:
                        recipe, ingredient, cuisine = row
                        myCuisine_1[int(recipe), int(ingredient)] = 1
                        for row in reader_2:
                            recipe, ingredient, cuisine = row
                            myCuisine_2[int(recipe), int(ingredient)] = 1

            myCuisine_1 = array_mutilator(myCuisine_1)
            myCuisine_2 = array_mutilator(myCuisine_2)

            RECIPE_INDEXES_ARRAY_CUISINE_1, ALL_RECIPES_ARRAY_CUISINE_1 = indexer(myCuisine_1)
            RECIPE_INDEXES_ARRAY_CUISINE_2, ALL_RECIPES_ARRAY_CUISINE_2 = indexer(myCuisine_2)

            if cuisines_array.index(cuisines_array[x]) < cuisines_array.index(cuisines_array[y]):
                calculator(RECIPE_INDEXES_ARRAY_CUISINE_1, ALL_RECIPES_ARRAY_CUISINE_1,
                    RECIPE_INDEXES_ARRAY_CUISINE_2, ALL_RECIPES_ARRAY_CUISINE_2,
                    cuisines_array[x], cuisines_array[y])

        csvfile_2.close()
        csvfile_1.close()
```

Step-by-step execution of the function:

- Open a pair of cuisine files and store that data in two arrays of the same size as the overall array used in previous methods
- The overall arrays are then mutilated accordingly. The mutilator function removes all empty rows but not columns since ingredient IDs are important for calculations.
- Both arrays are then processed by the indexer function in the same fashion as in second and third method
- With these steps we obtain the total of four arrays which are further passed to calculator function along with names of the paired cuisines for the output. Calculator function acts in similar fashion as in method 3 (some changes were made but are not significant for this discussion).

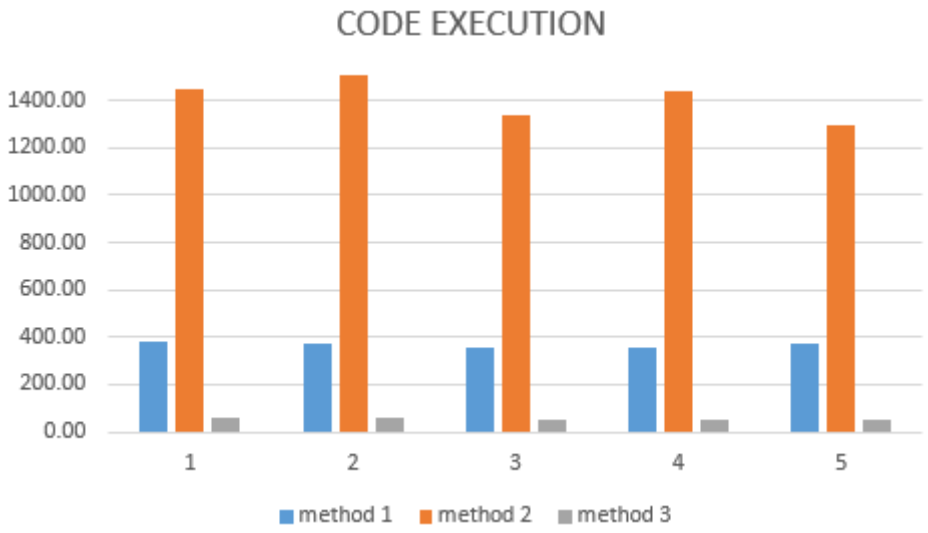
The total execution time for calculating the similarities between cuisines, using the approach just described, was 21381.237164974213 seconds, as measured by Pythons time module. This roughly translates into 6 hours.

2.5.2. Comparison and explanation

In the previous chapter we described all the Python scripts that were used in our project. As mentioned before first three methods focus on recipes and individuals and the fourth one focuses on cuisines. Since the last one is different in a conceptual way we have excluded it from the tests.

	method 1	method 2	method 3
1	377.72	1451.21	56.23
2	374.07	1584.75	56.05
3	356.05	1337.76	53.44
4	360.40	1442.50	52.76
5	370.57	1296.34	53.60
100	6625.6316	11500.8403	358.2118

All three compared methods were executed five times in the same environment. For calculating the execution time we used time module which measured time in form of seconds. The table above shows five executions for each method for the first 10 recipes. We have also provided data about the execution time for 100 recipes but only once since it takes too long. Data from the table is shown in the graph below.



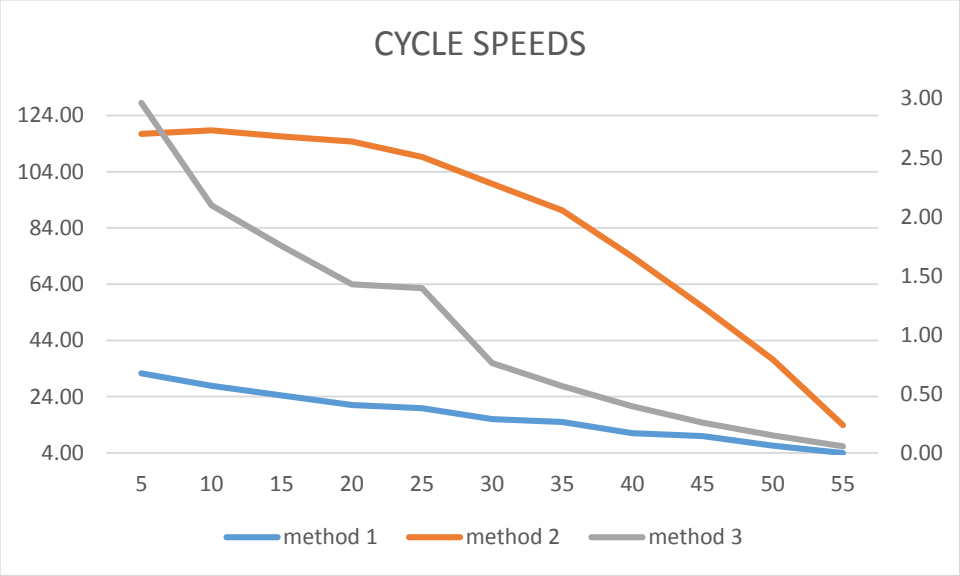
Our data set includes 57691 different recipes and by observing execution times of method one and two one can easily conclude that these two methods were out of the question. Method three was written as a result of that obvious fact. Total execution time of method 3 was a little less than 19 hours. Given the equation below we can calculate the number of operations that need to be executed for comparison of recipes (each to each without repetitions). The result is 1664096895 operations.

$$x = (n-1) * \frac{n}{2} \; ; \; where \; n = 57691$$

The table below also shows execution times for all three methods with purpose of showing the change in the duration of the cycle with respect to x value. The values in the right most column are in thousands and represent the starting point of x in the loop. All methods have been executed for one cycle (e.g. x goes from 10000 to 10001). Final row in the table represents the average time difference per 5000 units.

	method 1	method 2	method 3
5	32.352	117.483	2.964
10	27.885	118.663	2.097
15	24.457	116.577	1.752
20	21.030	114.699	1.426
25	19.895	109.244	1.394
30	16.018	99.740	0.761
35	14.987	90.189	0.566
40	11.033	73.780	0.395
45	10.034	55.953	0.255
50	6.621	37.162	0.149
55	3.975	13.843	0.056
AVG Δ	2.838	10.364	0.291

The data shown in the table above is visualised with the graph below. For better visualisation method three is linked to right side axis because its execution times are much lower than they are for the other two methods.



It comes as a surprise that method one was faster than method 2. It is obvious that the slice method used in method 2 is computationally very complex and thus contributes to the execution time. If we observe the cycle times of method 1 and 2 we can see the trend. Method one is almost linear which makes sense. It is the simplest one – with every iteration the array size decreases by one, and thus the complexity. The graph could be described by f(x) = m - n; where m number of rows and n is row index in array. Graphs of method 2 and 3 are opposite – one is convex and the other one is concave. They have a more complex structure but method 3 is almost linear (the glitch can be explained by interruption from an outer process while calculating). Graph of method 3 is almost perfectly concaved. I guess we could say that they (2 and 3) can be described by a logarithm of some sort.

1. Visualisation of our work

1.1. Introduction

All the work done with processing the data, that was described in previous chapters, and all the information gathered this way would have no value if we would have not visualized it and shared with others.

Data visualisation, in its definition, stands for presentation of data in some graphical format. These can be video, pictures, charts, etc. Main purpose of data visualisation is to ease the understanding of information that data is trying to portrait.

In the past data that was visualised was mostly static. Once a chart was made it stayed the same. Interactive visualisation is the current name of the game and enables users (viewers) to tailor the input at their own need – it enables interaction.

For this purpose we created a web site that provides a detailed insight to the work done and also provides interaction between the user and data gathered. Main focus is on visualisation of the results of code described in previous chapters.

Within this chapter we describe how the web site was built. All the tools used and a detail description of the code used for visualisation of the results. We start by describing the JavaScript code used for web application.

In the last part, a description of the page structure is given with short explanation of the process. We will finish by explain how our web app was deployed.

1.2. Using D3.js

Given the data obtained using Python scripts we proceed to visualise this information. This is done using JavaScript and D3.js library (stands for data-driven documents).

D3.js is a JavaScript library for manipulating documents based on data. It provides powerful visualisation components and data driven approach to document object model (DOM) manipulation. The library allows the user to bind arbitrary data to a DOM and then apply data-driven transformations to the document (D3, 2015).

D3 API provides a vast range of different visualisation techniques, from simple tables to complex customized maps and colourful data displays. Since our data is world oriented (main focus is on geo data with respect to different regions of the world) we decided to use the capabilities of the library and visualise our results in form of a map. The visualisation is the final stage of the project and will be described in detail in this chapter.

All of the visualisation is done within a SVG component of our webpage. SVG stands for scalable vector graphics and it defines vector based graphics in a XML format. According to its developers at the World Wide Web consortium (W3C), SVG is a mark-up language for describing two dimensional graphics applications and images, and a set of related graphics script interfaces (W3, 2015).

```
var svg = d3.select("body").append("svg")
          .attr("width", width)
          .attr("height", height);
```

Our SVG element is initialized with the code shown in the picture. The code selects the body element of the HTML frame and appends the SVG component to it, with the attributes of height and width defined as separate variables.

Next step is the definition of projection. According to Snyder and Voxland (Snyder & Voxland, 1994) projection is a systematic transformation of the latitudes and longitudes of locations on the surface of a sphere or an ellipsoid into locations on a plane. That is, locations in three dimensional space are made to correspond to a two dimensional representation.

```
var projection = d3.geo.mercator()
          .center([50,70])
          .scale(150)
          .rotate([0,0,0]);
```

Within the D3 library API there are 36 different kinds of projections that vary in the representation and types of use. For our project we used the Mercator projection which is a cylindrical map projection presented by Gerardus Mercator in 1569. It is standard map projection nautical purposes.

The Mercator projection distorts the size of objects as the latitude increases from the equator to the poles. That is why Greenland and Antarctica appear much larger relative to the landmasses near the equator than they actually are (Wikipedia, 2015)

In the scope of initializing our projection we define three aspects of it. Center, scale and rotate. Center defines the initial point of view when the page is loaded. In our case the centre is set for 50 degrees to the right of the prime meridian (Greenwich) and 70 degrees north of the prime parallel (Equator).

Scale defines the level of zoom or the height from which the world is observed and it is defined with respect to the projection. In our case the scale is defined at 150. If we would increase the scale the point of view would be closer to the surface, if we decrease it we farther our point of view from it.

The rotate property is defined with values for longitude, latitude and roll. In our example we use the default values which are all zero. Which means that our map is not rotated in any direction.

```
var path = d3.geo.path()
        .projection(projection);
```

Next we define the path. A path class similar to a line. It's basically a line between points in our SVG that have been modified using the projection so they are synchronized with our map and they do not distort it.

These points are stored in a JSON file that is manipulated with TopoJSON.js library. This library is an extension of GeoJSON that encodes topology. Rather than representing geometries discretely, geometries in TopoJSON files are stitched together from shared line segments (Hugo, 2015). This means that border between two countries is drawn only once – it is optimized.

Another library that has been used in our project is queue library. It is used for loading the data to the page and prevents execution of the code before data finished loading.

```
queue()
  .defer(d3.json, "world-110m2.json")
  .defer(d3.csv, "country_data.csv")
  .await(main);
```

Queue is a plugin that loads external files into the script and ensures each file is completely loaded before allowing the script to continue running (Matthews, 2013). In our case we load the topoJSON file, the data about the countries and after these files are fully loaded we proceed with the main function.

We used jQuery to interact with the database. All the countries are coloured with respect to the similarity value. JQuery is a cross platform JavaScript library designed to simplify the client side scripting of HTML. It is the most popular JavaScript library in use today, with installation on 65% of the top 10 million highest trafficked sites on the web (jQuery, 2015).

First we define the query that we would like to perform on the database. In our case the query searches for data about similarities between the cuisine chosen by the user and all others in the dataset.

```
var cypher = {"statements":[{"statement":"MATCH (a)-[n:SIMILIARITY]->(b) " +
                                         "WHERE a.name= " + '"' + country_selected + '"' +
                                         " AND b.scope = " + '"Country" ' +
                                         "RETURN b.name AS CUISINE, 1/n.value AS SIMILARITY " +
                                         "ORDER BY n.value DESC"}]};
```

Next we pass this query through ajax to our database and retrieve the results. The function takes several parameters which define the database (port and authentication), it parses to query to JSON and returns the data.

```
$.ajax({
    type: "POST",
    accept: "application/json",
    contentType:"application/json; charset=utf-8",
    url: "http://localhost:7474/db/data/transaction/commit",
    headers: {"Authorization":"Basic <bmVvNGo6bW9ycGhsaW5n>"},
    data: JSON.stringify(cypher),
    success: function(data, textStatus, jqXHR){
    for(z = 0; z < 31; z++){
        var output = data.results[0].data[z].row[0] + ", " + data.results[0].data[z].row[1];
        console.log(output);
        var paragraph = document.createElement("p");
        var text = document.createTextNode(output);
        paragraph.appendChild(text);
        var text_output = document.getElementById("results");
        text_output.appendChild(paragraph)}
    ;},

      failure: function(msg){console.log("failed")}
    });
} else {window.alert("NOT IN DATASET. Please choose a shaded country.");}
```

The output is then written to a HTML document specified in the loop. All of the countries are coloured with respect to their similarity value. This colouring is done by scale (a D3 function) and as a parameter we pass the index value of the country in the array that gets returned by our query.

```
var colour_scale = d3.scale.linear()
                     .domain([0,32])
                     .range(["yellow", "black"]);
```

```
d3.select("path#_" + data.results[0]
    .data[z].row[2])
    .style("fill", colour_scale(z))
    .style("stroke-width", "1px")
    .attr("class", "selected");
```

These were the most important definitions for our visualisation. There some more but to not play significant role.

The example shows the app being connected to a localhost database but in the end be used GrapheneDB which is a hosting service for Neo4j in the AWS cloud. They offer free hosting up to 1000 nodes and 10000 relationship. Since our whole database is much bigger than that we took just a portion of it to the cloud.

1.3. Building the webpage

As mentioned before, the web site was built so that a potential visitor can get acquainted with the work done in as much detail as possible. In this chapter we will briefly describe the structure of the web site and the tools used in this stage. We conclude by a description of deployment.

Information provided on the site includes all the work done within my internship as Sabanci University as well as personal information about me. The web site serves as a form of CV. With respect to this the webpage includes:

- Description of the project
    - Shorter version of this text
    - References and external links
- Description of the author (CV)
- Contact information including:
    - Git hub account information
    - LinkedIn account information
    - Facebook account information and
    - Email address
- The project itself (the app)

The web page was built, as any other, with standard HTML framework and with Sublime text editor. As a fundamental rule of web development we respect the separation of content and design so all the design parts are linked to the webpage as an external source (.css and .js file).

Design of the web page is done with the combination of cascading style sheets (CSS) and JavaScript. CSS is used for making the web site visually appealing and JavaScript is used for animation of the index page as well as some banners, providing content description.

I can only say that web design's syntax is simple but there are so many tricks and pitfalls that it makes it quite an adventure.

All the documents used in the project have been uploaded to the GitHub repository for everyone to see them. This repository is linked to an application launched on Heroku. In essence: every time we make a change to our app we commit and push to the repository. This change is automatically transferred to the app on Heroku – pretty cool.

Unfortunately Heroku service does not provide hosting for pure HTML and JavaScript applications. We had to pull out a trick from our sleeve. In the project repository we included an empty .php file and a composer.json file. By doing this Heroku thinks that we have a PHP application and it deploys it like a charm.

2. CRITICAL CONCLUSIONS

We started this project with an idea and a very broadly scoped plan. The plan was to get some data and do something with it. There were several different topics in mind before we decided to go for recipes.

Once data was obtained and deemed suitable we started thinking. What now? The idea from all along was to calculate the distances of cuisines but we also thought of making the whole database presentable to the user. This would mean that the user should be able to query the dataset at its own will. Well this would be quite a challenge so we narrowed it a little bit, as noticed.

Before starting on the visualisation and whole web site design we did some work in Gephi. It is a really remarkable tool for exploration of graphs and visualisation on the local end. It is still in beta version and has many bugs but it is slowly becoming the tool to use. Some results of this work could be presented here but we are already too long.

Well, needless to say, we did what we set out to do, with many problems (solved) in the way but we would also like to present a critical evaluation of our work:

- Data: even with the improvements described in the data chapter it was still incomplete and with many flaws, not to the GIGO extent! The result would have been better if we would have more countries (now we only have 32). Some countries cannot be descripted like countries (North America) and would have to be partitioned and separated. And ingredients set is not big enough. Out of a little more than 1500 only 400 are actually used in recipes, some of them are very strange and make little sense (for example wood, fish, vine, etc.). We have almost 600000 recipes in our dataset which is very good but these are defined only by their ID's. We would have more options if we would also have names.
- Webpage: the design of the webpage is like something from early 00's. We could have done it better and we will but at the time of this writing it is not nice. It is frustrating to be a web designer. We could use a template to do it but what is the point in that. Also the website is not secured. Everything is done in the front end with JavaScript – but given the nature of the project this is not such an issue. Also the majority of text is static. This makes page loading time longer and the HTML code is ugly. The text should be fed dynamically from a database.
- Database comparison tests: for a more valid comparison we would have to write more complex queries with longer and complex traversals to point out the power of the graph. All the queries should also be executed in isolation and with automated tools.

During the time spent working on this project we learned a lot. The biggest challenge was to change the mind-set from relational to graph data model.

Given the fact that I had little experience with java script prior to this and beside a short web development class none in HTML and CSS - every new idea was a challenge, happily tackled. I am very pleased to say that the project was fun and educational.

The knowledge and the know-how obtained this period can and will be used in the future. There are many such and similar opportunities to express it. We might also fix the issues pointed out in the critical evaluation and make all of this more presentable. THANK YOU FOR READING!

References

Andlinger, P. (2013, 11 21). *DB-engines*. Retrieved from http://db-engines.com/en/blog_post/23

Asay, M. (2015, 4 3). Retrieved from techrepublic.com: http://www.techrepublic.com/article/nosql-databases-eat-into-the-relational-database-market/

Batra, S., & Tyagi, C. (2012). Comparative analysis of relational and graph databases. *International Journal of Soft computing and Engineering* .

Bruggen, R. V. (2015, 7 1). *Fascinating food networks in Neo4j*. Retrieved from blog.bruggen.com: http://blog.bruggen.com/2013/12/fascinating-food-networks-in-neo4j.html

D3. (2015, 8 1). Retrieved from http://d3js.org/: http://d3js.org/

Hugo, L. (2015, 27 8). *Topojson*. Retrieved from github.com: https://github.com/mbostock/topojson/wiki

jQuery. (2015, 8 24). *jquery.com*. Retrieved from https://jquery.com/

Matthews, S. (2013, 8 8). *D3 queue js*. Retrieved from giscollective.org: http://giscollective.org/d3-queue-js/

Neo4j. (2015, 27 8). *The Neo4j Manual v2.2.5*. Retrieved from neo4j.com/: http://neo4j.com/docs/stable/

Redmond, E., & R. Wilson, J. (2012). *Seven databases in seven weeks: a gude to modern databases and NoSQL movement.* Dallas, Texas: The Pragmatic Progamers.

Robinson, I., Webber, J., & Emil, E. (2013). *Graph databases.* Sebastopol: O'Reilly Media.

Sahil, B. (2014, 6 5). MySQL vs Neo4j vs MongoDB.

Snyder, P. J., & Voxland, M. (1994). An album of map projections.

Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., & Wilkins, D. (2010). A comparisson of a graph database and a relational databse: A data provenance perspective.

W3. (2015, 8 15). *SVG*. Retrieved from http://www.w3.org: http://www.w3.org/Graphics/SVG

Wikipedia. (2015, 9 3). *Mercator projection*. Retrieved from en.wikipedia.org: https://en.wikipedia.org/wiki/Mercator_projection

Wikipedia. (2015, 9 1). *NoSQL*. Retrieved from wikipedia.org: https://en.wikipedia.org/wiki/NoSQL

Appendix

Table of all query execution times:

	1		2		3		4		5		6		7		8		9		10	
	Cyphe	SQL	Cyphe	SQL	Cyp	SQL	Cyph	SQL	Cypher	SQ	Cypher	SQL	Cypher	SQL	Cyp	SQL	Cypher	SQ	Cypher	SQL
1	100	0.0182	4140	5.1727	38	0.0008	98	0.0028	7858	0	6595	5.8882	18810	0.04	0	11.02	7605	0	504	3.8135
2	84	0.0139	4309	5.3688	35	0.0008	98	0.0037	8773	0	7545	5.0802	17962	0.04	0	10.62	7494	0	469	3.5889
3	92	0.0136	4079	5.2462	43	0.0009	80	0.0028	7607	0	6453	5.0764	17860	0.04	0	9.45	8047	0	511	3.9159
4	106	0.0135	4026	5.2029	42	0.0008	85	0.0052	8652	0	6683	5.0682	18257	0.04	0	9.45	7457	0	459	3.9632
5	88	0.0136	4103	5.3481	38	0.0009	70	0.0029	7374	0	6541	5.0649	18407	0.06	0	9.48	7265	0	480	3.7144
6	94	0.0138	3884	5.4109	38	0.0009	82	0.0028	7611	0	7426	4.9989	18300	0.04	0	9.21	7338	0	495	3.7525
7	156	0.014	3844	5.267	41	0.0009	88	0.0028	8417	0	6725	5.0183	18161	0.04	0	9.65	8132	0	505	3.8816
8	135	0.0136	3924	5.3872	33	0.0008	60	0.0029	7629	0	6649	4.9576	18154	0.04	0	9.59	7324	0	469	4.243
9	102	0.0139	3898	5.3097	34	0.0008	81	0.0027	7567	0	6619	4.972	17832	0.05	0	9.51	7344	0	761	3.8648
10	78	0.0136	4072	5.2541	42	0.0008	68	0.0027	8301	0	7618	4.9619	18068	0.04	0	9.28	7335	0	466	3.6838
AV	104	14.17	4028	5296.8	38	0.84	81	3.13	7979	0	6885	5108.7	18181	43	0	9726	7534	0	512	3842.2

1. Name of the most used ingredient in Africa:
Cypher:

```
MATCH (i:INGREDIENT)-[n:IS_PART_OF]->(r:RECIPE)-[:IS_FROM]->(c:CUISINE)-[:ORIGINATES_FROM]->(a:AREA)
WHERE a.name = "African"
RETURN a.name as AREA_NAME,i.name, count(n) AS OCCURANCES
ORDER BY OCCURANCES DESC
LIMIT 1
```

SQL:

```
SELECT area.name, ingredient.name, count(ingredient.id) AS COUNT
FROM area
LEFT JOIN cuisine ON area.name = cuisine.area_name
LEFT JOIN recipe ON recipe.cuisine = cuisine.name
LEFT JOIN ingred_recipe ON ingred_recipe.id_rec = recipe.id
LEFT JOIN ingredient ON ingredient.name = ingred_recipe.ing
WHERE area.name = "African" AND cuisine.area_name = "African"
GROUP BY ingredient.id
ORDER BY COUNT DESC
LIMIT 1
```

2. Areas and number of recipes in them + total number of ingredients in those recipes + number of cuisines in those areas:
Cypher:

```
MATCH (i:INGREDIENT)-[:IS_PART_OF]->(r:RECIPE)-[:IS_FROM]->(c:CUISINE)-[:ORIGINATES_FROM]->(a:AREA)
RETURN a.name AS AREA_NAME, count(distinct(i.ID)) AS INGREDIENTS, count(distinct(r.ID)) AS RECIPES,
count(distinct(c.ID)) AS CUISINES
ORDER BY AREA_NAME
```

SQL:

```
SELECT area.name, count(distinct cuisine.id), count(distinct recipe.id), count(distinct ingred_recipe.ing)
FROM area
LEFT JOIN cuisine ON area.name = cuisine.area_name
LEFT JOIN recipe ON recipe.cuisine = cuisine.name
LEFT JOIN ingred_recipe ON ingred_recipe.id_rec = recipe.id
GROUP BY area.name
```

3. Top five cuisines for Israel:
Cypher:

```
MATCH (c1:CUISINE) - [n:SIMILIARITY] - (c2:CUISINE)
WHERE c1.name = "Israel" AND c2.scope = "Country"
RETURN c2.name, n.value
ORDER BY n.value DESC
LIMIT 5
```

SQL:

```
SELECT cuisine.ID, similarity.cuisine_2, cuisine.scope, similarity.value
FROM similarity
LEFT JOIN cuisine ON similarity.cuisine_2 = cuisine.name
WHERE similarity.cuisine_1 = "Israel" AND cuisine.scope = "Country"
ORDER BY similarity.value DESC
LIMIT 5
```

4. Mutual ingredients between two cuisines (Bangladesh and Pakistan):
Cypher:

```
MATCH (c1:CUISINE)<-[:IS_FROM]-(r1:RECIPE)<-[:IS_PART_OF]-(i:INGREDIENT)-[:IS_PART_OF]->(r2:RECIPE)-[:IS_FROM]-
>(c2:CUISINE)
WHERE c1.name = 'Bangladesh' AND c2.name = 'Pakistan'
RETURN c1.name AS C1, c2.name AS C2, count(distinct(i.name)) AS MUTUAL
```

SQL:

```
SELECT DISTINCT ingredient.id, ingredient.name
FROM cuisine
LEFT JOIN recipe ON recipe.cuisine = cuisine.name
LEFT JOIN ingred_recipe ON ingred_recipe.id_rec = recipe.id
LEFT JOIN ingredient ON ingredient.name = ingred_recipe.ing
WHERE ingredient.id IN (SELECT DISTINCT ingredient.id
                        FROM cuisine
                        LEFT JOIN recipe ON recipe.cuisine = cuisine.name
                        LEFT JOIN ingred_recipe ON ingred_recipe.id_rec = recipe.id
                        LEFT JOIN ingredient ON ingredient.name = ingred_recipe.ing
                        WHERE cuisine.name = "Pakistan"
                    ) AND cuisine.name = "Bangladesh"
```

5. Count all:
Cypher:

```
MATCH (c:INGREDIENT_CATEGORY)<-[:BELONGS_TO]-(i:INGREDIENT)-[:IS_PART_OF]->(r:RECIPE)-[:IS_FROM]->(k:CUISINE)-
[:ORIGINATES_FROM]->(a:AREA)
RETURN count(distinct(c.ID)),count(distinct(i.ID)),count(distinct(r.ID)),count(distinct(k.ID)),count(distinct(a.ID))
```

SQL:

```
SELECT count(area.ID), count(cuisine.ID), count(ingredient.ID), count(ingredient_category.ID), count(recipe.ID)
FROM area, cuisine, ingredient, ingredient_category, recipes
```

6. Number of ingredient categories in areas:
Cypher

```
MATCH (c:INGREDIENT_CATEGORY)<-[:BELONGS_TO]-(i:INGREDIENT)-[:IS_PART_OF]->(r:RECIPE)-[:IS_FROM]->(k:CUISINE)-
[:ORIGINATES_FROM]->(a:AREA)
RETURN COUNT(DISTINCT(c.ID)), a.name
ORDER BY a.name
```

SQL:

```
SELECT area.name, count(distinct ingredient.category)
FROM area
LEFT JOIN cuisine ON area.name = cuisine.area_name
LEFT JOIN recipe ON recipe.cuisine = cuisine.name
LEFT JOIN ingred_recipe ON ingred_recipe.id_rec = recipe.id
LEFT JOIN ingredient ON ingredient.name = ingred_recipe.ing
GROUP BY area.name
```

7. How many recipes include alcohol in area(%) :
Cypher:

```
MATCH (c:INGREDIENT_CATEGORY)<-[:BELONGS_TO]-(i:INGREDIENT)-[:IS_PART_OF]->(r:RECIPE)-[:IS_FROM]->(k:CUISINE)-
[:ORIGINATES_FROM]->(a:AREA)
WITH c, i, r, k, a
MATCH (recipe)-[:IS_FROM]->(cuisine)-[:ORIGINATES_FROM]->(area)
WHERE c.name = "alcoholic beverage" AND a.name = "SouthAsian" AND area.name = "SouthAsian"
RETURN toFloat(COUNT(DISTINCT(r.ID)))*100/toFloat(COUNT(DISTINCT(recipe.ID))) AS BANANA, a.name
```

SQL:

```
SELECT area.name AS BANANA, count(distinct recipe.id)*100/ (
SELECT count(distinct recipe.id)
FROM area
LEFT JOIN cuisine
ON area.name = cuisine.area_name
LEFT JOIN recipe
ON recipe.cuisine = cuisine.name
LEFT JOIN ingred_recipe
ON ingred_recipe.id_rec = recipe.id
LEFT JOIN ingredient
ON ingredient.name = ingred_recipe.ing
WHERE area.name = "SouthAsian"
GROUP BY area.name) AS PERCENTAGE
FROM area
LEFT JOIN cuisine
ON area.name = cuisine.area_name
LEFT JOIN recipe
ON recipe.cuisine = cuisine.name
LEFT JOIN ingred_recipe
ON ingred_recipe.id_rec = recipe.id
LEFT JOIN ingredient
ON ingredient.name = ingred_recipe.ing
WHERE ingredient.category = "alcoholic beverage" AND area.name = "SouthAsian"
GROUP BY area.name
```

8. Most fruity cuisine (%):
Cypher:

```
MATCH (c:INGREDIENT_CATEGORY)<-[:BELONGS_TO]-(i:INGREDIENT)-[:IS_PART_OF]->(r:RECIPE)-[:IS_FROM]->(k:CUISINE)
WITH c, i, r, k
MATCH (recipe)-[:IS_FROM]->(cuisine)
WHERE c.name = "fruit" AND k.name = cuisine.name
RETURN toFloat(COUNT(DISTINCT(r.ID)))*100/toFloat(COUNT(DISTINCT(recipe.ID))) AS BANANA, k.name
ORDER BY BANANA DESC
```

SQL:

```
SELECT cuisine.name AS BANANA, count(distinct recipe.id)*100/ (
SELECT count(distinct recipe.id)
FROM cuisine
```

```
LEFT JOIN recipe
ON recipe.cuisine = cuisine.name
LEFT JOIN ingred_recipe
ON ingred_recipe.id_rec = recipe.id
LEFT JOIN ingredient
ON ingredient.name = ingred_recipe.ing
WHERE cuisine.name = BANANA
GROUP BY cuisine.name) AS PERCENTAGE
FROM cuisine
LEFT JOIN recipe
ON recipe.cuisine = cuisine.name
LEFT JOIN ingred_recipe
ON ingred_recipe.id_rec = recipe.id
LEFT JOIN ingredient
ON ingredient.name = ingred_recipe.ing
WHERE ingredient.category = "fruit"
GROUP BY cuisine.name
ORDER BY PERCENTAGE DESC
```

9. Sum all numeric values (id values):
Cypher:

```
MATCH (c:INGREDIENT_CATEGORY)<-[:BELONGS_TO]-(i:INGREDIENT)-[:IS_PART_OF]->(r:RECIPE)-[:IS_FROM]->(k:CUISINE)-
[:ORIGINATES_FROM]->(a:AREA)
RETURN SUM(c.ID) + SUM(i.ID) + SUM(r.ID) + SUM(k.ID) + SUM(a.ID) AS SUMA
```

SQL:

```
SELECT SUM(ingredient_category.ID) + SUM(ingredient.ID) + SUM(recipe.ID) + SUM(cuisine.ID) + SUM(area.ID) AS SUMA
FROM ingredient_category, ingredient, recipe, cuisine, area
```

10. How many recipes include alcohol in all areas!!!
Cypher:

```
MATCH (c:INGREDIENT_CATEGORY)<-[:BELONGS_TO]-(i:INGREDIENT)-[:IS_PART_OF]->(r:RECIPE)-[:IS_FROM]->(k:CUISINE)-
[:ORIGINATES_FROM]->(a:AREA)
WHERE c.name = "alcoholic beverage"
RETURN count(DISTINCT(r.ID)) AS BANANA, a.name
ORDER BY a.name
```

SQL:

```
SELECT area.name AS BANANA, count(distinct recipe.id)
FROM area
LEFT JOIN cuisine
ON area.name = cuisine.area_name
LEFT JOIN recipe
ON recipe.cuisine = cuisine.name
LEFT JOIN ingred_recipe
ON ingred_recipe.id_rec = recipe.id
LEFT JOIN ingredient
ON ingredient.name = ingred_recipe.ing
WHERE ingredient.category = "alcoholic beverage"
GROUP BY area.name
```