Storage Systems Comparison between MySQL, MongoDB and Neo4j using Spark

Federico Fiorini
University of Trento - EIT Digital
Via Sommarive 9, 38123
Povo (TN), Italy
federico.fiorini29@gmail.com

Ana Laura Daniel Diaz
University of Trento - EIT Digital
Via Sommarive 9, 38123
Povo (TN), Italy
anadanield@gmail.com

ABSTRACT

In this paper we present a comparison between different storage systems: a traditional Relational Database (MySQL), a NoSQL Database (MongoDB) and Graph Database (Neo4j). To do so, we use a big dataset containing more than 160M records of commercial flights information in the USA between 1987 and 2015. The raw dataset has been pre-aggregated and formatted accordingly for each of the databases with the use of Spark. In two cases, Spark was also used to import the data to the databases.
This evaluation has been achieved by comparing implementation complexity, integration with Spark, and comparing time performances of running several queries on each of the databases.

Keywords

SQL, NoSQL, MySQL, MongoDB, Neo4j, Graph Database, Document Database.

1. INTRODUCTION

NoSQL refers to different database technologies that were developed in response of the need of the current state of the data world which is now demanding properties such as scalability and flexibility. In fact, when we talk about Big Data (billions/trillions of records) there is a need of change from the old RDBMS and switch to tools and technologies that can handle that amount of data better.

We have a dataset of more than 160M records on which we would like to perform some analysis such as finding communities of cities and states by how well connected they are through their departing and arriving flights. But before carrying out the analysis on the flights data, we want evaluate three different storage systems and decide which option is the most suitable for the dataset we are working with. A traditional Relational Database and two NoSQL Databases, A Document Database and Graph Database, are the types of databases to be compared.

For the present comparison we considered factors such as the implementation complexity considering that the chosen NoSQL technologies are newer than the traditional Relational Database; the available structures in which data can be expressed; runtime performance of query execution as well as the readability of the queries.

Apache Spark is an open source cluster computing framework that plays a big role in the big data world because of its multi-stage in-memory MapReduce implementation which has proven to provide great performance. For this reason we feel it's also important to consider a database's support and ease of integration with Spark.

Project report, EIT Digital, Service Design and Engineering, University of Trento, Italy. Courses: Big Data and Social Networks. Academic Year: 2015/16. Authors: Federico Fiorini (181655), Ana Laura Daniel Diaz (181493).

1.1 The Data

The data that will be used to carry out the comparison was obtained from the Bureau of Transportation Statistics of the United States of America and it consists of information on every commercial flight made between 1987 and 2015 within the USA, which adds up to more than 160 million rows of information. The data is heavily focused on the details about the performance of the departure and arrival of the flight, such as estimated times vs actual times, delay causes, cancellations, etc., However, we've decided to focus only on the information about the origin and destination of the flights.

The format in which the data is provided is through zip compressed comma separated values files that can only be grouped by month. We ended up with 338 files with a size of 827 MB when compressed and around 18 GB uncompressed.

Each csv file contains the following information for each flight: **Year**, **Month**, **Day_Of_Month**, **Origin** (The 3-letter IATA airport code of the origin airport), **Origin_City_Name**, **Origin_State_Abr**, **Origin_State_Nm**, **Dest** (The 3-letter IATA airport code of the destination airport), **Dest_City_Name**, **Dest_State_Abr**, **Dest_State_Nm**.

2. RELATED WORK

Several comparisons have been made between different storage systems. Especially between MySQL and MongoDB we can find lots of articles about performance comparison in the Big Data community. The most commonly compared characteristics are scalability, data distribution, query and data model, performance. Lot of case studies show the great difference between Relational Database Management System (RDBMS) and Document Databases in terms of data modeling, comparing advantages and disadvantages. None of them determines a "winner" but they do underline how each of these systems have different use cases and depending from the requirements you must work with you can choose one or the other.

On comparison between Graph and Relational Databases the articles are not that many, especially because of the relative newness of graph databases, but we can still find some.

The focus here is more on data modeling since the two concepts of relational (tables) and graph (nodes and edges) are quite far from each other even though almost every traditional relational database can be translated in a graph one. The advantages of a graph database are mostly dependent on the type of queries that are ran on it and on the structure of the dataset. We can find lot of examples that show how using a graph database with a dataset that well fits on the nodes-relationships schema, is actually the best solution, especially when we talk about big dataset describing networked data.

Some articles and examples can be found also about comparison between Graph and Document Databases. It is been shown how Document databases lack in relationships because they store sets of disconnected collections. This makes it difficult to use them for connected data, where a graph database suits way better. On the other side, if the dataset contains structured information that can easily fit into a collection of documents, a Document Database (MongoDB in our case) is the way to go.

As we have stated, several case studies and articles have already been written about this topic. In this paper we want to extensively compare MySQL, MongoDB and Neo4j, respectively representing RDBMS, Document and Graph Databases, in terms of data structuring, ease of use and performance. Thus, not so much work has been done in comparing these storage systems in combination with other distributed technologies. In our study we will integrate the databases with Apache Spark to see improvements in performance.

3. PROBLEM STATEMENT

What we want to achieve with this project is to find the storage system that performs and suits the best for the kind of dataset we have.

The first problem to address is the format of the raw data and how to properly structure it for each of the databases before we can run the queries. We want to transform the data keeping in mind the most sensible way it should be stored according to each of the database's properties and use. Therefore, some manipulation of the data will be needed and the fact that there are millions of records needs to be considered.

After we manage to successfully import the flights dataset to each of the databases we will have to deal with the queries that we will use to compare the databases performance. We also want to analyse the way queries are expressed in each of the query languages used for each database so we can also take into consideration the clarity of a query.

4. SOLUTION

The only way to decide which of the databases is more suitable for our flight data is to perform a comparison between them. To address the problem of the millions of records, the pre-aggregation will be executed with Apache Spark. Afterwards, the data will be imported to each of the databases and this process will also be assessed.

Finally, some queries will be run in each of the databases and their performance will be put into comparison. The nature of the queries and how they are executed will also be taken into consideration.

4.1 Use of Spark to load and manipulate data

The import of the raw data to each of the databases we will analyse was done using Apache Spark. We mainly wanted to take advantage of Spark's *Resilient Distributed Dataset* (RDD) objects. An RDD is a collection of elements and one of their primary characteristics is that operations can be performed on them in a parallel way, thus, making processing faster. By making use of this abstraction we are able to easily and efficiently manipulate the huge amount data from the csv files.

The csv files contain rows with information of every flight but for the type of analysis we are planning to do on the data we decided that we only care about how many flights per route were made for each day. For that reason we needed to do some aggregation before importing the data to the databases.

The approach taken is to load one csv file at a time into an RDD. Spark provides a method that allows us to create an RDD from a text file which is loaded into the RDD as a collection of lines. Then, basic string manipulation functions are applied to do some cleaning of the data, such as removing quotes or replacing them so they wouldn't cause problems during further steps.

Next, to count how many flights were made on every route for every day, we applied simple map and reduce functions over the RDD containing the data. We first transformed each string array of the previously created RDD into a **key-value** pair with use of a map function. We used the python and scala programming languages and both of them allowed us to represent a pair with a *tuple* of 2 elements.

For the **key** element of the pair we use a tuple containing the Year, Month, Day_Of_Month, Origin, Dest attributes of the flight. The **value** of every pair is 1 so we can perform a simple count with the reduce function (similar to count-words). After reducing, the resulting key-value pairs gives us the number of flights made per route for every day, which we will refer to as the route *frequency*.

```
Map:                                 Reduce:

( (2015,01,01,DFW,JFK) , 1 )
( (2015,01,03,DFW,JFK) , 1 )   ( (2015,01,01,DFW,JFK) , 2 )
( (2015,01,01,DFW,JFK) , 1 ) → ( (2015,01,02,DFW,JFK) , 2 )
( (2015,01,02,DFW,JFK) , 1 )   ( (2015,01,03,DFW,JFK) , 1 )
( (2015,01,02,DFW,JFK) , 1 )
```

Figure 1. Representation of the MapReduce process to calculate daily route frequency.

Depending on the database we export the RDD to, extra steps are needed but they will be described in the following sections along with the method used for the writing process. Up until this point, we managed to import the lines from a csv files and create a flights RDD that contained the flight route information per day and its frequency.

4.2 MySQL Implementation

Apache Spark includes the Spark SQL module which is used for structured data processing. Within this module we are able to use the *DataFrame* object: a distributed collection of data but, unlike an RDD, it is organized into named columns. Thus, it is conceptually equivalent to a table in a relational database. DataFrames also provide us with the interface to save our data into an external source. In our case, this external source will be MySQL.

4.2.1 Data structure

We decided to store the flights' information in two tables in order to comply with the normalization tendency in relational databases to reduce data redundancy. Figure 2 illustrates the schema used for the MySQL database.

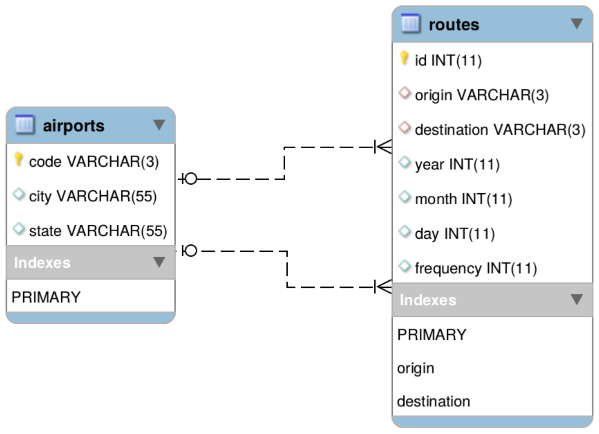


Figure 2. EER Diagram generated with MySQL Workbench.

The **airports** table will have 3 fields: *code*, *city*, and *state* with the *code* field as the primary key. This code refers to the the 3-character IATA code of every airport. The **routes** table has the information regarding the flight route per day. Its fields are: *id*, *origin*, *destination*, *year*, *month*, *day*, and *frequency*. The id is the primary key and is an auto-incremental integer. The *origin* and *destination* fields are foreign keys that reference the primary key on the airports table.

4.2.2 Importing Process

Like mentioned above, DataFrames allows us to write a collection into an external storage and for it to connect with our MySQL server we use the *Java Database Connectivity* (JDBC) API for MySQL. To create a DataFrame from our flights RDD each record has to be transformed into a *Row* object through a map function. The write interface was fairly simple to use but one of the drawbacks we encountered was that it still doesn't have plenty of options when writing into a table.

When saving the flights DataFrame on the routes table, we use the *append* mode of the writing interface which means that when it tries to write into a certain table that already exist, it will append to the table every record in the DataFrame performing the write action. After processing each csv file, the information of the flights on it will be appended to the routes table that already contains information on flights found in previous csvs.

This raised no problem since every row on each of the csv files represents a new flight, but in order to populate the airports table we need to take a different approach in order not to have duplicate airports. When processing each csv file, we have to construct another RDD with every origin and destination airports of the flights listed and select the distinct values. The problem was that once we had the list of airports, at the time of writing we were unable to tell which of them had already been imported to MySQL. As seen on the EER diagram, the primary key of the airports table is the attribute 'code'. So, after processing the first csv the majority of the airports would already be present in the airports table and appending another airport with the same code will raise an exception. To avoid this we need to use the `INSERT IGNORE` MySQL command when inserting new rows to the airports table but currently there is no way to specify that option to the *DataFrame* write interface. Hence, we had to deal with the insertion of the airports with python and its MySQLdb library in order to use that option.

To summarize our implementation in MySQL (sans querying), for every csv file we create an RDD for the flights information and a second RDD containing only airport information. We were able to smoothly export the flights RDD to the routes table in our MySQL server by transforming it to a DataFrame and using its write interface and the JDBC connector for MySQL. For the airports RDD, we extracted its contents and formatted them to execute the `INSERT IGNORE` MySQL command through python with said contents as in `VALUES` segment of the command.

4.3 MongoDB Implementation

4.3.1 Data structure

MongoDB structures data into collections of *JSON* documents which support enhanced and more flexible data structures than the traditional Relational Database tables made up of columns and rows. Not only they do support standard field types like number, string, boolean, etc., but JSON fields can also be arrays or nested sub-objects.

```
{
 "_id" : ObjectId("56aa449361e9c60cce81ce44"),
 "year" : "2015",
 "day" : "01",
 "month" : "01",
 "origin" : {
   "code" : "SFO",
   "city" : "San Francisco CA",
   "state" : "California"
 }
 "destination" : {
   "city" : "Dallas TX",
   "code" : "DFW",
   "state" : "Texas"
 },
 "frequency" : 5
}
```

Figure 3. A document from the routes collection in MongoDB.

To take advantage of this we decided to structure our flights dataset in a way that the airport information would remain inside the flight route document as a nested sub-object. It is extremely unlikely that modifications to an airport's information are needed, which would mean the update of every flight document, so this approach will not bring drawbacks other than information duplication. As a result, we only needed one collection in our MongoDB database which we called routes. Figure 3 displays an example document in this collection.

4.3.2 Data structure

To use Spark with MongoDB we used the *MongoDB Hadoop Connector* developed by the MongoDB organization. This connector allows for RDDs to be created from MongoDB collections and vice versa.

In the MySQL implementation we first had to transform the records in our flights RDD into Row objects. To write the elements from the RDD to a MongoDB collection, the elements have to be in a *key-value* pair format. The **key** would stand for the id of the document in the MongoDB collection but in our case we will leave it as a null value because we intend to create a new object. The **value** will have the information of the flight in the form of a *BSON* object, which stands for "Binary JSON" and they are more efficient both in storage space and scan-speed when comparing them to JSON documents. This BSON object will be structured so it matches the example document presented in Figure x.

Once our RDD is in the correct key-value pair format we can use the `saveAsNewAPIHadoopFile` method that is available thanks to the MongoDB Hadoop Connector. This will not actually save the RDD into a Hadoop file, like the name implies, but this method receives a parameter for configuration settings and this way we can specify the MongoDB server information, the database to use, and the name of the collection on which the documents will be saved.

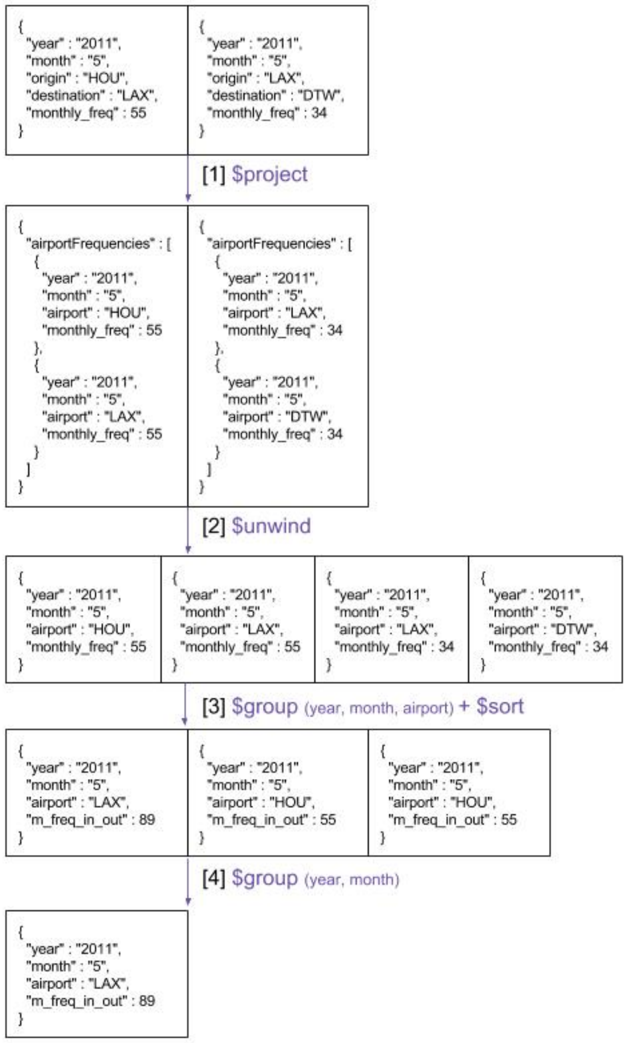
One last thing to note about the Spark and MongoDB integration is that it is still somewhat recent and we found that doing the integration with the scala programming language was more stable since our attempt at doing so with python resulted in encountering a major bug that didn't allowed the data to be written in the MongoDB.

4.3.3 *Query Structure in MongoDB*

To run the queries in MongoDB we use the aggregation pipeline in which documents enter several stages and during each stage they are transformed to be consumed by the following stage. Each of these stages are sent to the aggregate() function in an array with its respecting parameters.

Figure 4. illustrates the way some of the aggregation stages are sent to the aggregation pipeline for one of our evaluation queries and how it transforms the documents along the process.

We wanted to find the airport with more flights (in and out) per month. Since we intend to provide only an example of some of the stages of the pipeline, in Figure x. we start with documents that have already been grouped by *year*, *month*, *origin.code*, *destination.code*, and their frequencies have been summed up into the *monthly_freq* field. Step `[1]` shows how the `$project` stage


```
{                               {
 "year" : "2011",                "year" : "2011",
 "month" : "5",                  "month" : "5",
 "origin" : "HOU",               "origin" : "LAX",
 "destination" : "LAX",          "destination" : "DTW",
 "monthly_freq" : 55             "monthly_freq" : 34
}                               }
```

```
{                               {
 "airportFrequencies" : [        "airportFrequencies" : [
   {                               {
    "year" : "2011",                "year" : "2011",
    "month" : "5",                  "month" : "5",
    "airport" : "HOU",              "airport" : "LAX",
    "monthly_freq" : 55             "monthly_freq" : 34
   },                              },
   {                               {
    "year" : "2011",                "year" : "2011",
    "month" : "5",                  "month" : "5",
    "airport" : "LAX",              "airport" : "DTW",
    "monthly_freq" : 55             "monthly_freq" : 34
   }                               }
 ]                               ]
}                               }
```

```
{                      {                      {                      {
 "year" : "2011",       "year" : "2011",       "year" : "2011",       "year" : "2011",
 "month" : "5",         "month" : "5",         "month" : "5",         "month" : "5",
 "airport" : "HOU",     "airport" : "LAX",     "airport" : "LAX",     "airport" : "DTW",
 "monthly_freq" : 55    "monthly_freq" : 55    "monthly_freq" : 34    "monthly_freq" : 34
}                      }                      }                      }
```

```
{                       {                       {
 "year" : "2011",        "year" : "2011",        "year" : "2011",
 "month" : "5",          "month" : "5",          "month" : "5",
 "airport" : "LAX",      "airport" : "HOU",      "airport" : "HOU",
 "m_freq_in_out" : 89    "m_freq_in_out" : 55    "m_freq_in_out" : 55
}                       }                       }
```

```
{
 "year" : "2011",
 "month" : "5",
 "airport" : "LAX",
 "m_freq_in_out" : 89
}
```
Fig 4. Illustration of the documents generated after the aggregation stages made through the MongoDB aggregation pipeline.

applies transformations to create new documents with a new field called *airportFrequencies* which is an array containing 2 objects. Since we want to get how many flights go through an airport and we do not care if they are arriving or departing flights, we save the year, month, airport code, and monthly_freq first with the origin airport code and secondly with the departure airport code. Basically, we want to account the monthly frequency to both the origin and departure airports.

The `$unwind` stage takes an array field and creates as many documents as there are elements in the specified field. With step `[2],` the `$unwind` stage takes the *airportFrequencies* array and uses it to create 2 new documents for every previous document in the pipeline. This way, we managed to change the documents and instead of being focused on a flight route, they are now focused on an airport. In step `[3]`, the `$group` and `$sort` stages will group them by year, month, and airport and will sum up the monthly frequencies into the *m_freq_in_out* field. This field then will be used to sort documents in a descending order so the airport with the most flights per month will be placed at the top. Finally, in the `$group` stage of step `[4]` documents will be again grouped but only by year and month and since they are already sorted, all we need to do is to take the first element of every group to get the airport with the most flights for every month.

We thought that the way the aggregation stages are expressed and applied were clear and straight-forward which made its implementation easier.

4.4 Neo4j Implementation

4.4.1 Data structure

Neo4j successfully implements the property graph model in which entities, called nodes, are connected to each other through relationships. Both nodes and relationships can have attributes in the form of key-value pairs.

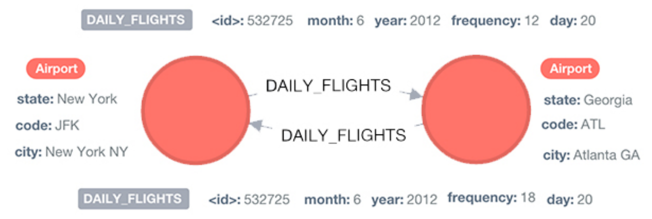


Figure 5. Visualization from Neo4j of two airport nodes and two `DAILY_FLIGHTS` relationship between them.

Figure 5 shows how we've decided to describe the flights information in the form of airport nodes connected through relationships. The airport nodes will contain the airport information as attributes. We've stored the routes information as relationships called `DAILY_FLIGHTS` from airport to airport nodes. The frequency of the route will be stored as an attribute in this relationship.

4.4.2 Importing process

Unlike with MongoDB, currently there is no official Spark and Neo4j connector. There exist some alternatives but we considered them too complicated because we would have to make use of more frameworks, like Docker. So following the recommendation from the Neo4j organization, we only used Spark to generate other temporary csv files that are ready to be imported to Neo4j.

We created the `DAILY_FLIGHTS` relationships from our already cleaned and aggregated flights RDD. However, for a relationship to be valid it must have a starting and ending node so we have to create the airport nodes first.

When processing each raw csv file, we first generate a temporary csv file with the airport nodes to import. We achieve this with the same approach used in the MySQL implementation, creating another RDD with the collection of airports. Then, the csv containing the relationships (our flight routes information) between airport nodes is generated from the flights RDD.

Again, even if we apply the distinct function to the airports RDD, we are processing one raw csv file at the time so we end up creating multiple airports csv files that will be imported to Neo4j. To handle this, we first merge all of the resulting airports csv files and run a bash script that will scan the merged file and will remove the duplicates.

Once the Neo4j csv files are ready we run its import command and send the paths of the files to be imported as nodes or as relationships.

4.4.3 Querying Structure in Neo4j

Finding the airport with more flights (in and out) per month is done by executing the query presented in Figure 6.

```
MATCH (a:Airport)-[n:DAILY_FLIGHTS]-()
WITH a.code AS code, n.year AS year,
    n.month AS month, sum(n.frequency) AS flights
ORDER BY flights DESC
RETURN year, month, collect(code)[0] AS airport,
    max(flights) AS tot_flights
```

Figure 6. Query to find the airport with more flights per month in the Cypher query language.

Cypher is Neo4j's query language and its main goal is to allow programmers to focus only on what they want to select from the database and not so much about how to do it. If we compare the query in Figure x. to how we structured the same query in MongoDB using its aggregation pipeline we can clearly see the differences between taking an approach in which the programmer specifies how to get the data vs what data to get.

Also, the way the data is modeled in Neo4j allowed this query to be much simpler than in MongoDB or MySQL. The relationship created between one node and another can be directed to specify the origin and destination, but when querying, Cypher makes it easier to select every relationship for an airport node without caring whether it's the origin or destination node of the relationships.

5. EXPERIMENTS

The implementation and querying experiments were carried out in a single computer with the following spec:
- MacBook Pro
- Processor: Intel Core i5 2.8 GHz
- RAM: 8 GB 1600 MHz

5.1 Writing

We were able to integrate MySQL and MongoDB with Spark so we compared the writing speeds of these two approaches. For Neo4j we only used Spark to prepare the csv files it would use in its import command, for this reason the writing time reported for Neo4j refers to the time spent pre-aggregating the data plus the time spent executing the import command with the resulting csv files.

All three of the databases needed the same amount of rows (MySQL), documents (MongoDB) or relationships (Neo4j) to store the information on the daily frequency of all of the routes found in the dataset. After the daily aggregation we remained with 35M records to insert into the various databases. So the import times that we are reporting regard the manipulation of 166M records and insertion of 35M aggregated records into the various databases.

As seen on Table x, the MySQL implementation took the longest with more than double of the MongoDB and Neo4j Import. We believe that a reason for this great difference is that every time a table gets new rows, its indexes must be updated. We decided to set up the indexes to keep the querying experiments fair.

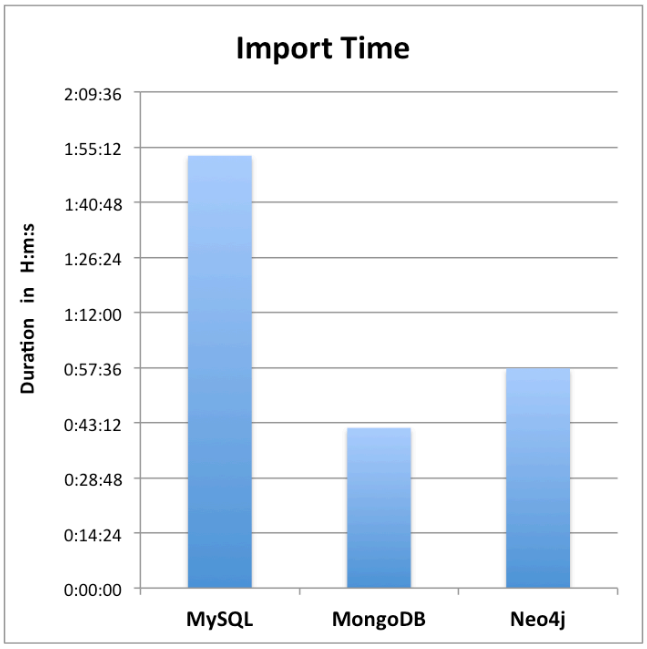


Figure 7. Total importing times for each database.

5.2 Querying

Considering that we are doing this comparison using an historical dataset and that we want to obtain insights from this data, the queries are focused on reading records and performing aggregation in order to extract facts that can be interesting to know about flight routes information.

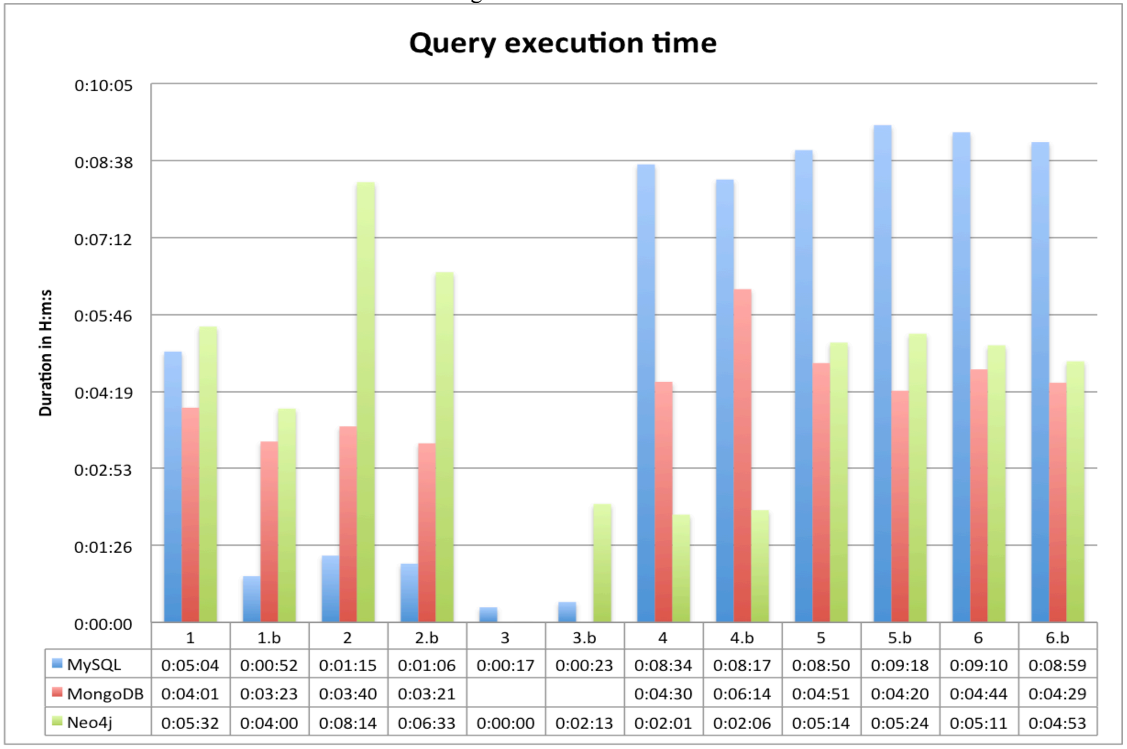
Next is the list of the queries and their variation that we executed. A complete time comparison table can be found in the Figure 8. Each query was executed 10 times and averaged.

List of queries we performed on the databases:

1. Find the most frequent route per month
   1.b Per year
2. Find the airport with more flights (in and out) per month
   2.b Per year
3. Given a departure date find the shortest path from airport A to airport B
   3.b. Consider only high frequency routes (top 75%)
4. Find the state with more internal flights (per month)
   4.b Per year
5. Find the state with more external departure flights per month
   5.b Per year
6. Find the state with more external arrival flights per month
   6.b Per year

We want to draw attention to the following queries and analyse how each of the databases performed better than the other in these queries.

The best performance for MongoDB was in queries 4, 5, and 6 and its variations which involved a reference to the state the airport was located in. Evidently, by keeping the airport as a sub-object in the same route document it was easier to reference the origin and destination airports' states. In contrast, for MySQL to reference the state of both the origin and destination airports required 2 `JOIN` operations. In fact, if we run `EXPLAIN` on the query we can see how many rows MySQL needs to iterate through and the number is more than 50M. Instead, MongoDB has all the needed information within the document and it needs to go through way less documents.



	1	1.b	2	2.b	3	3.b	4	4.b	5	5.b	6	6.b
🟦 MySQL	0:05:04	0:00:52	0:01:15	0:01:06	0:00:17	0:00:23	0:08:34	0:08:17	0:08:50	0:09:18	0:09:10	0:08:59
🟥 MongoDB	0:04:01	0:03:23	0:03:40	0:03:21			0:04:30	0:06:14	0:04:51	0:04:20	0:04:44	0:04:29
🟩 Neo4j	0:05:32	0:04:00	0:08:14	0:06:33	0:00:00	0:02:13	0:02:01	0:02:06	0:05:14	0:05:24	0:05:11	0:04:53

Figure 8. Queries execution time compared between all databases.

We were surprised to see how much better MySQL performed in the 1.b, 2 and 2.b. We can attribute this to the fact that this query needed no expensive `JOIN` operations. Also, while running these queries in MongoDB we had to use a parameter to allow the aggregation pipeline to write to disk when it couldn't load all of the documents in memory. We deduced that MongoDB's aggregation pipeline is not as fast when there are several steps of aggregation and manipulation of the objects.

Another interesting query to analyse is the shortest path query (query number 3). Such query requires a very complex implementation of the Dijkstra's Shortest Path Algorithm in MySQL that we obtained from the www.artfulsoftware.com website. The time performance of running the query in MySQL wasn't that critical at 17 seconds. On the other hand, Neo4j was capable to execute this query in less than 1 second and even outputing several shortest paths while the Dijkstra algorithm implementation in MySQL only returns one shortest path.

However, when some aggregation was needed to filter only high frequency routes, Neo4j performance dropped considerably and MySQL was better to outperform it.

6. FUTURE WORK

Further works can be done to improve the comparison of the performances of the three databases analysed in this report. First of all, MongoDB strength comes also from its distributed capability, so it would be interesting to setup a cluster and see how the distribution of the data impacts on performances. Since MySQL is not natively a distributed database, to level the play field we could use ScaleBase, a distributed relational database platform that uses MySQL for data storage. Adding ScaleBase we would make MySQL a distributed database too and we would be able to compare performances fairlier.

As we can see in Appendix A, in Neo4j we tried two ways to perform the queries: (1) using the base DAILY_FLIGHTS relationship (with the daily frequency of a route) or (2) first aggregating in monthly and yearly frequencies and writing to the database new relationships called MONTHLY_FLIGHTS and YEARLY_FLIGHTS in order to use them in our queries when possible. Using this second approach we noticed an incredible gain in performance (some queries are more than 10 times faster using the aggregated fields) because the queries skip an aggregation level already done in advance. The problem in this second approach is that it requires a lot of time to do the first aggregation on all the dataset. Also if there are any changes in the data, the whole aggregation must be repeated. This is not our case because we are working with an historical database and the data does not change, but it might be the case in other scenarios. It would be interesting to use the second approach also with MySQL and MongoDB to see performance improvements.

One of the goal of the project was to get the data ready to be further analysed with some data mining techniques. An interesting analysis would be to run some clustering algorithm on the graph dataset (in Neo4j) to find communities of cities well connected and see how these communities changed through the years. To do so we think it will be possible to use the YEARLY_FLIGHTS relationship type (containing the yearly frequency of flights per route) and use it to build a weighted graph on where to run some clustering algorithm.

7. CONCLUSION

We found that the use of Spark was very beneficial to perform the manipulation and pre-aggregation for all of the databases, even for Neo4j which was not integrated with Spark.

For the Spark integration with MySQL, even if the writing performance was the worst we considered that the fact that MySQL support comes almost "out-of-the-box" with Spark is a very interesting factor that would allow traditional relational databases to take advantage of Spark's clustering power without much hassle.
As for the performance of the queries, we expected Neo4j to perform much better than MySQL and MongoDB in the shortest path query. What we didn't expect was that it will perform so badly when it did aggregation such as grouping and filtering when querying. However, once we realized this and did further pre-aggregation before querying, Neo4j showed better results. For this reason we would be very inclined to use this database with the help of Spark for a more thorough aggregation in future works.

We also expected to see a much better performance from MongoDB but we concluded that we needed data that required a more complex structuration in order to benefit from the flexibility brought by their document based records.

8. REFERENCES

[1] Transitioning from relational databases to MongoDB, source: http://blog.mongodb.org/post/72874267152/transitioning-from-relational-databases-to-mongodb

[2] SQL programming guide, source: http://spark.apache.org/docs/latest/sql-programming-guide.html

[3] MongoDB manual core, aggregation pipeline, source: https://docs.mongodb.org/manual/core/aggregation-pipeline/

[4] MongoDB manual reference, aggregation pipeline, source: https://docs.mongodb.org/manual/reference/operator/aggregation-pipeline/

[5] Neo4j and Apache Spark, source: http://neo4j.com/developer/apache-spark/

[6] Generating csv files to import into Neo4j, source: http://www.markhneedham.com/blog/2015/04/14/spark-generating-csv-files-to-import-into-neo4j/

[7] NoSQL explained, source: https://www.mongodb.com/nosql-explained

APPENDIX

A. IMPROVING IN PERFORMANCES WHEN FIRST AGGREGATING IN NEO4J

With Neo4j we decided to take a second approach to see if we had any improvement in performances. Before running all the reading queries we run two queries to aggregate by month and by year in order to find rispectively monthly and yearly frequency. For each aggregation we write a new relationship in the graph: MONTHLY_FLIGHTS and YEARLY_FLIGHTS, which contain the correlated frequencies. Having these new relationships allow us to use them in the queries in order to skip one passage of aggregation which, as we stated before, is the bottleneck in Neo4j.

We tested both approaches on a sample dataset containing only two years of flights (2011 and 2012). This is because we had problems running this second approach with the entire dataset since apparently it requires lot of memory to perform the aggregation, and we had not enough in the machine we were using. In Figure 9 we can see the results of the comparison between the two versions. We can see that even though the second version took extra time to aggregate the data, it is worth it since almost all the queries ran in less than 1 second and 10 – 20 times faster than the approach without aggregation. The time is displayed in minutes, seconds and centiseconds (m:s:cs).

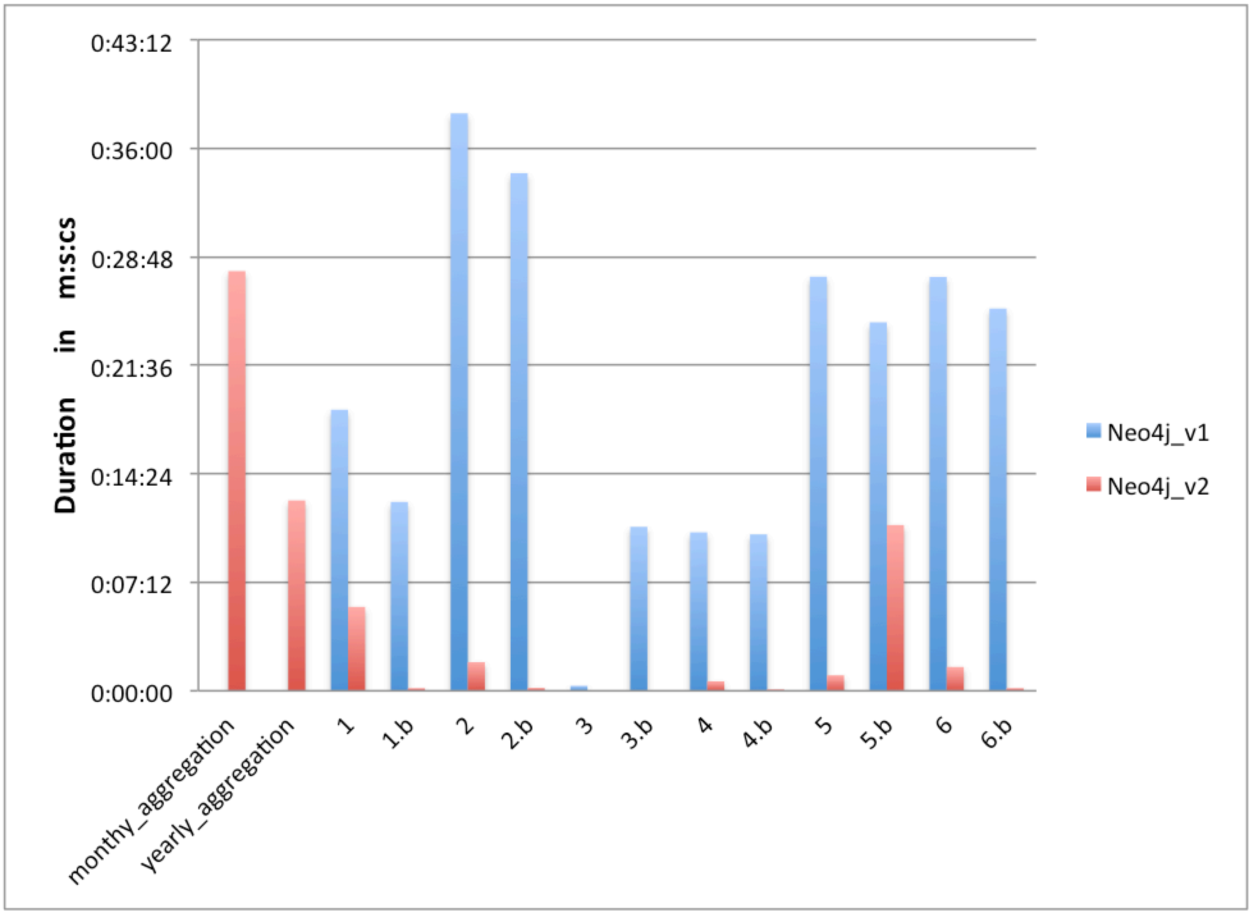


Figure 9. Queries execution time compared with different implementation on Neo4j.