Web Data Management -Programming Project Group 8

Georgios Dimitropoulos, 4727657
Manolis Manousogiannis, 4727517
Georgia Zarnomitrou, 4724089

1 INTRODUCTION

In this paper we present the inner workings of four scalable database systems (PostgreSQl, Apache Spark, Apache Lucene and Neo4j) with the help of a movie database created with information gathered from IMDB. Specifically, we display the strengths and weaknesses of each system and the lessons we have learned from our attempts to implement various queries in each system.

2 DATASET

In order to examine the performance of the four aforementioned systems we worked with a database regarding movies and tv series. The information in this database was gathered from IMDB and given to us by professor Lofi in the form of a backup file from a postgreSQL database. This database contains a variety of information like the titles of the movies or tv series, the names of the actors and the characters they portrayed, plot keywords, the name of the producing companies and other miscellaneous data. This database is composed of 18 tables where each table has hundreds of thousands or millions of rows and has a size of 6.3GB. The size of our database proved to be a problem during our implementations since we often needed to convert it online in other formats for use in different systems and some systems could not handle such large files.
Firstly, we attempted to merge this database with another one we found online[1] because this smaller dataset contained additional information regarding the success of some movies and the popularity of its actors and cast. However this proved an impossible task due to PostgreSQL's lack of sufficient built-in functions to handle strings, so we decided to add this additional information in the form of fictitious data. For this purpose we wrote our own scripts in PostgreSQL (these scripts along with our code for each system can be found here [2]) that created columns for the budget, gross and duration of the movies in the table movie_info. We also added a column in the table cast_info to present the likes the cast of a movie has in Facebook and one column in the table aka_name that represents the likes each actor has in Facebook.

3 QUERIES

In this section we present the various queries we want to implement in the four database systems we have chosen.

(1) Find the ratio of the facebook likes of a specific actor to those of the cast in all the movies he has participated in.
(2) Movie titles where the director is Abd Al-Hamid, Ja 'far and the actor is "Tom Hanks".
(3) Return the number of companies who have produced films with the word love in the title.
(4) The titles of the 1000 more successful movies from 1990 until 2000.
(5) The titles of the movies where an actor and the character he/she portrays have the same name.
(6) Find the title of each movie that contains the word "design" in the list of plot keywords.
(7) How many times does a word appear in the whole database.
(8) Return the sum of movie gross of the movies produced by "Amazon" for different production years.
(9) Return average of the movie budget spent by the company "Paramount Television" for different production years.
(10) How many times does the name "Harry" appear in all the TV series of the database.
(11) Find relation between tv series success and number of episodes.
(12) How could we be constantly aware of the number of rows of the table titles.
(13) Return the count of times the keyword "kill" appears in the titles of our movies.
(14) Return the ids of keyword table tuples where keyword is equal to "kill" or phonetic code is equal to "A2525".
(15) Return number of tuples in table keywords where keyword field starts with the word kill.

4 SYSTEM POSTGRESQL

In this section we present the system PostgreSQL and how it performed when we run the queries described in section 3. PostgreSQL is a powerful open source object-relational database that can be used to perform complex queries. The choice to pick this as one of our systems was an easy one since PostgreSQL is one of the most robust, reliable and widely known systems for handling relational databases.

In table 1 we display the execution time and planning time PostgreSQL required to perform the aforementioned queries. Planning time refers to PostgreSQL's functionality **query plan** where the system receives a query and searches for the optimal way to execute the user's request (this can be seen in figures 1 and 2 as PostgreSQL offers the possibility to view this information). In this table we can observe three empty rows, these represent queries we could not implement in PostgreSQL. Query 2 could not be implemented because of the way our database is structured, specifically the complexity of the database impedes the creation of a simple query that can go through all the necessary tables to answer the question. This shows a weakness of the Postgres system since queries depend on the way the database is structured, which is often flawed due to the human factor.

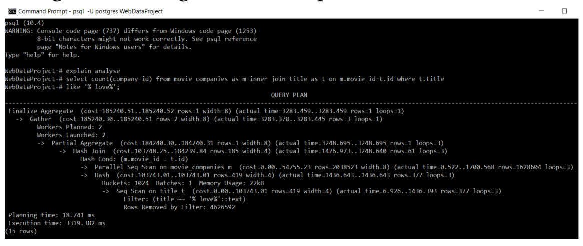
In order to resolve query 7 we would have to go through all the columns of all the tables of our database. This is an inefficient,

Queries	Execution Time (ms)	Planning Time(ms)
1	8315.539	0.101
2	Could not implement	Could not implement
3	335.684	1.929
4	500096.916	48.762
5	43327.073	47.007
6	2322.867	16.356
7	Could not implement	Could not implement
8	1041.994	21.230
9	5890.218	3.825
10	5867.972	18.546
11	Could not implement	Could not implement
12	2727.789	0.179
13	233.212	19.314
14	279.284	0.759
15	6.887	3.473

Table 1: shows the execution and planning time for our queries

difficult to implement, time consuming procedure that requires a lot of memory. We could not execute query 11 due to the size of our database and the limited resources of our computers (specifically we got an out of memory warning). However provided more resources we could get a response for this particular query. Before executing our queries we indexed the database at the appropriate columns (code for indexing is provided) which resulted in an improvement of our execution time (see figures 1 and 2).

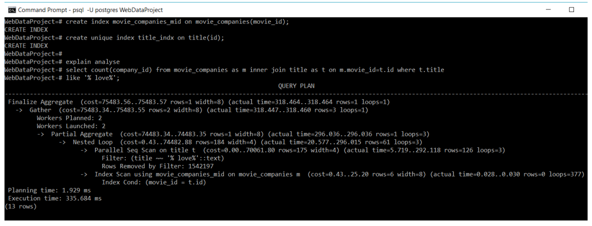
From table 1 we can observe that PostgreSQL performs well for most of the queries (execution time is under 5 seconds) but under performs for queries 4 and 5. Query 4 is executed in around 500 seconds because of the mathematical calculations and sorting required for it whereas query 5 takes longer to execute because of PostgreSQL's weakness handling string variables and more specifically in finding if two string values are equal.



```
Command Prompt - psql  -U postgres WebDataProject
psql (10.4)
WARNING: Console code page (737) differs from Windows code page (1253)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

WebDataProject=# explain analyse
WebDataProject-# select count(company_id) from movie_companies as m inner join title as t on m.movie_id=t.id where t.title
WebDataProject-# like '% love%';
                                                                         QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=185240.51..185240.52 rows=1 width=8) (actual time=3283.459..3283.459 rows=1 loops=1)
   ->  Gather  (cost=185240.30..185240.51 rows=2 width=8) (actual time=3283.378..3283.445 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=184240.30..184240.31 rows=1 width=8) (actual time=3248.695..3248.695 rows=1 loops=3)
               ->  Hash Join  (cost=103748.25..184239.84 rows=185 width=4) (actual time=1476.973..3248.640 rows=61 loops=3)
                     Hash Cond: (m.movie_id = t.id)
                     ->  Parallel Seq Scan on movie_companies m  (cost=0.00..54755.23 rows=2038523 width=8) (actual time=0.522..1700.568 rows=1628604 loops=3)
                     ->  Hash  (cost=103743.01..103743.01 rows=419 width=4) (actual time=1436.643..1436.643 rows=377 loops=3)
                           Buckets: 1024  Batches: 1  Memory Usage: 22kB
                           ->  Seq Scan on title t  (cost=0.00..103743.01 rows=419 width=4) (actual time=6.926..1436.393 rows=377 loops=3)
                                 Filter: (title ~~ '% love%'::text)
                                 Rows Removed by Filter: 4626592
 Planning time: 18.741 ms
 Execution time: 3319.382 ms
(15 rows)
```

Figure 1: Execution time of query 3 with no indexing



```
Command Prompt - psql  -U postgres WebDataProject
WebDataProject=# create index movie_companies_mid on movie_companies(movie_id);
CREATE INDEX
WebDataProject=# create unique index title_indx on title(id);
CREATE INDEX
WebDataProject=#
WebDataProject=# explain analyse
WebDataProject-# select count(company_id) from movie_companies as m inner join title as t on m.movie_id=t.id where t.title
WebDataProject-# like '% love%';
                                                                         QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=75483.56..75483.57 rows=1 width=8) (actual time=318.464..318.464 rows=1 loops=1)
   ->  Gather  (cost=75483.34..75483.55 rows=2 width=8) (actual time=318.447..318.460 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=74483.34..74483.35 rows=1 width=8) (actual time=296.036..296.036 rows=1 loops=3)
               ->  Nested Loop  (cost=0.43..74482.88 rows=184 width=4) (actual time=20.577..296.015 rows=61 loops=3)
                     ->  Parallel Seq Scan on title t  (cost=0.00..70061.80 rows=175 width=4) (actual time=5.719..292.118 rows=126 loops=3)
                           Filter: (title ~~ '% love%'::text)
                           Rows Removed by Filter: 1542197
                     ->  Index Scan using movie_companies_mid on movie_companies m  (cost=0.43..25.20 rows=6 width=8) (actual time=0.028..0.030 rows=0 loops=377)
                           Index Cond: (movie_id = t.id)
 Planning time: 1.929 ms
 Execution time: 335.684 ms
(13 rows)
```

Figure 2: Execution time of query 3 with indexing

5 SYSTEM APACHE SPARK

In this section we examine the Apache Spark system. It is a system mainly used for batch processing. Specifically, it can be used in order to process previously collected data in a long batch. Furthermore, it can also perform stream processing of real-time data or used in interactive processing. For all the aforementioned reasons, Apache Spark is a general purpose engine, which is able to deal with all of the aforementioned tasks.

More specifically, we used Spark SQL that is focused on the processing of structured data. Also it uses a query optimizer(Catalyst) which investigates data and queries to produce an efficient query plan for data locality and computation.

Based on our dataset, we found that this system was efficient in the query "Find the ratio of the facebook likes of a specific actor to those of the cast in all the movies he has participated in". For our implementation we used PySpark and for this reason we were able to exploit some libraries of Python such as Numpy and Pandas which facilitated some calculations that we performed. Furthermore, we took advantage of the Matplotlib library and we were able to perform some interesting and intuitive plots of some of our queries. More specifically, we were able to depict in Figure 3 the graphical representation of the results of the query "Return the sum of movie gross of the movies produced by Amazon for different production years."

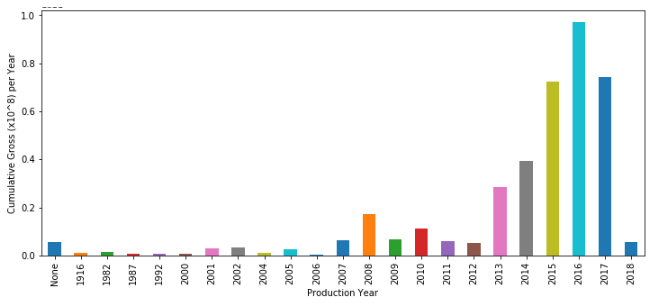


Figure 3: Cumulative Sum of movie gross of the movies produced by Amazon per Production Year

Moreover, in Figure 4 we can observe the graphical representation of the result of the query "Return average of the movie budget spent by the company "Paramount Television" for different production years". Finally, in Figure 5 we can observe the graphical representation of the result of the query "How many times does the name "Harry" appear in all the TV series of the database"

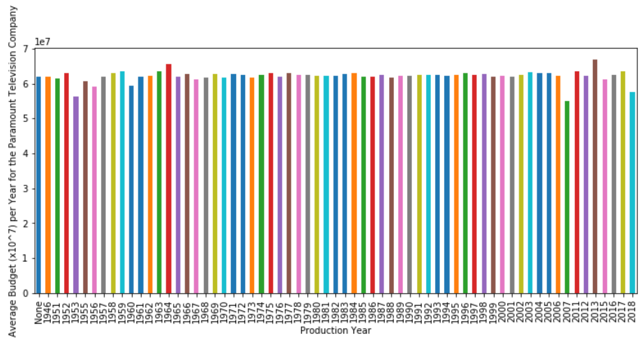


Figure 4: Average of movie budget spent by Paramount Television per Production Year

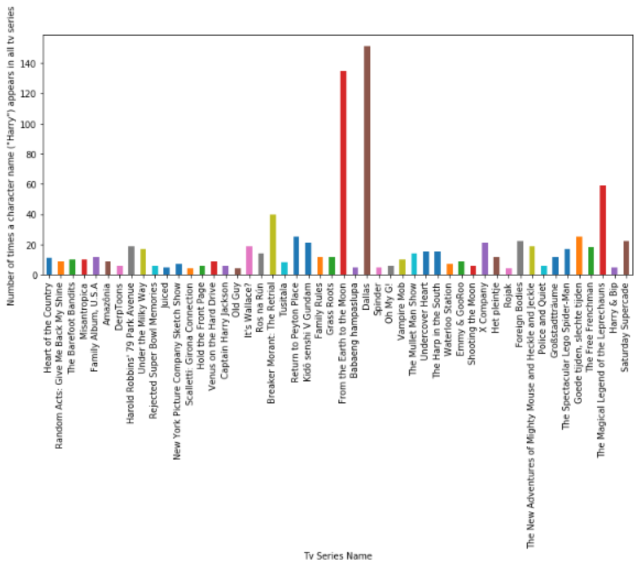


Figure 5: Number of times a character name (i.e. Harry) appears in all TV series

In Table 2 we can observe the execution time of our queries for the Apache Spark. As mentioned before, empty rows represent queries we could not implement. Query 2 was not implemented due to the structure of the database. Queries 5 and 11 were not able to be executed due to the size of our database and the limited resources of our computers (specifically we got an out of memory warning). Also, in order to execute query 7 we would have to go through all the columns of all the tables of our database and we found again that this was an inefficient, difficult and consuming procedure that requires again a lot of memory. Finally, we were not able to find an efficient way in order to implement query 12 that uses a trigger. However, if we were able to have more resources we could get a response for these queries.

6 SYSTEM APACHE LUCENE

In this section, we are going to present our third system,which is Apache Lucene. Lucene is an *inverted index* based text search engine library which is based entirely in Java. However, implementations in other languages are also available and we decided to implement our system in .NET.

Queries	Execution Time (seconds)
1	185
2	Could not implement
3	22.5
4	270
5	Could not implement
6	41.6
7	Could not implement
8	237
9	244
10	473
11	Could not implement
12	Could not implement
13	1.42
14	3.02
15	1.49

Table 2: Execution Time of the queries for the Apache Spark

The reason for selecting Lucene, is that it is considered to be a really fast search engine for text due to adding content to a full-text index, so we wanted to compare its performance in this task with PostgreSQL and Spark.
In order to explain the 'inverted index' term, we could say that it is equivalent to retrieving pages of a book with a specific keyword, by searching the index at the end of the book, rather than searching each page separately.
While implementing Lucene in .NET , we went through the whole procedure that takes place in this system. As the 'unit' of index and search in Lucene is a document, we had to process our dataset as a set of Documents which contain different fields. Every document was a row of our table and every feature (row) was a field in that document. A general overview of the process and indexing of our data is shown below.

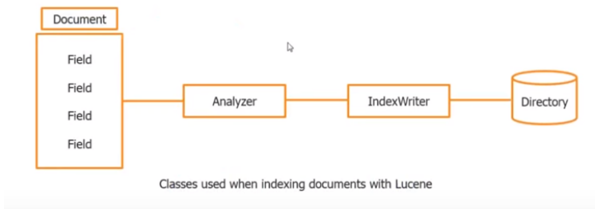


Figure 6: Process and index of data in Lucene

As Lucene is used for text search it was reasonable to test it using tables that contained text in it. From them, the most suitable was a table containing 236,000 different movie ids with their corresponding set of keywords and their phonetic code.So this means that we created 236,000 documents with 3 fields each.
After our indexing was complete we had to create a QueryParser object, which we used to pass our query to the Index searcher and retrieve our results.

Queries	Execution Time (miliseconds)
13	117
14	94
15	400

Table 3: Execution Time of the queries for Lucene

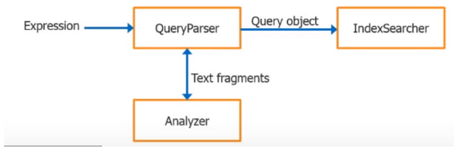


Figure 7: Text Search in Lucene

Due to the limitations we had, we did not perform all the queries mentioned above in Lucene systems. Some of them could not be implemented due to limitations of the system,such as comparing numerical values or finding mean numerical values, and some others were not implemented as our dataset was far too large to be loaded in our system, at least when using our local machines. In 3 we will present the execution time of each query implemented.
After performing those queries, we draw some useful results on Lucene system. One very interesting characteristic of the system, is that it is not case sensitive. Meaning that regardless of using capital or lower case letters for example, Lucene will be able to return all text related to what we requested. This is very important for a system used as a text search engine. A limitation of the system, is that it cannot under normal circumstances search for text in different document fields. Of course, when enabling the MultipleQueryParser in .NET this can be done easily,if we specify which fields of the document we want to search. However,what Lucene actually does in that case is that it performs multiple queries in each field separately, which of course makes it computationally expensive.

7 NEO4J

Neo4J is a graph processing system, which we could use to identify useful patterns in our dataset. Especially in the kind of dataset we were using, where directors, actors, different movies and many other entities were present, a graph processing system would be ideal to identify and visualize useful relationships between them. As an example, identifying the *oracle* of a specific actor or director, could give us the whole network that he or she has made with other people of the same industry.Tasks like that, would be impossible for all the other systems that we tested.
Unfortunately, due to time limitations we were not able to achieve a satisfying result with this system, as we had difficulties in creating the relationships between our nodes (entities) during the implementation phase. We managed to produce a graph which related every movie, with all the actors played in it , as well as its director, however this was not what we would like to achieve with this system. A figure of the visual relationship we mentioned above is attached below.

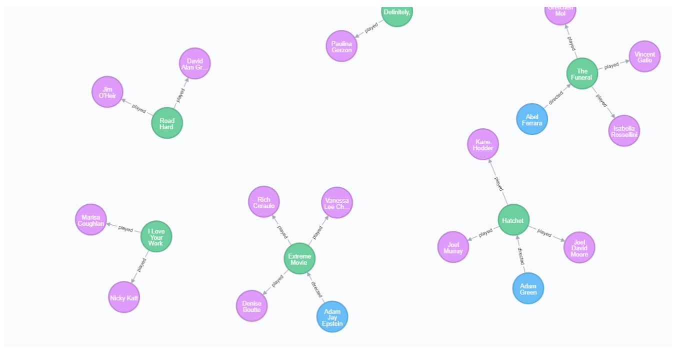


Figure 8: Realtionship between movies,actors and directors

8 COMPARISON

After using all four systems we have reached some conclusions regarding the strengths and weaknesses of each. Firstly, as mentioned above Apache Lucene is an ideal system for handling databases with a lot of text since in the majority of cases its execution time for this kind of queries is better than other systems. Another advantage of the Lucene system is that it is not case sensitive whereas both Apache Spark and PostgreSQL are, this means that additional code is needed in these systems to cover all possible combinations of upper and lower case letters (specifically we use the special command **ilike**). In addition, text based queries in both Postgres and Spark often require an exact knowledge of how the data is written in the database which is not always possible. Apache Lucene also offers the possibility of finding similarly written words like Bill and Hill, something that is not provided in the other systems.

On the other hand, the Lucene system cannot handle well numerical based queries unless we explicitly change its code since all the database information is treated as a string. In addition, even though Lucene can handle a query like the seventh one we proposed in section 3 this comes at a cost to the system's memory. Specifically such queries can be optimally handled by adding an extra column that contains all the information for that row but that would require binding a lot more memory.

As far as the Apache Spark system is concerned, it performed very well on queries that required calculations and a graphical representation of the result of the queries. The main reason for this advantage of this system is that we implemented it in PySpark which facilitates the usage of some Python Libraries. This is a possibility that is not provided by other systems. Through the figures 3-5 we can clearly depict the advantage of this system in comparison to the other systems in these kinds of queries. In addition we observed another beneficial effect of this system, specifically when executing the query "The titles of the 1000 more successful movies from 1990-2000", a sorting of the results was necessary. Hence, we observed that the Apache Spark system was more consistent in comparison to the other systems since we performed this identical query several times and the Apache Spark system was the only one that was able to preserve the same ordering of the results as a response to our query. Besides this, it was also able to achieve such a consistent performance in less time in comparison to the other systems (observing tables 1 and 2 we can see that Spark executed the query in half the time it took Postgres). Unfortunately we were not able to execute the queries of all systems in the same machine and this is something that justifies the slower performance of Apache Spark in some of the other queries.

In regards to the PostgreSQL system we observed that it was the only system able to handle a diverse type of queries with good performance (especially after indexing the database, the execution time was vastly improved). However it can only be used with bounded relational databases and this does not correspond well on the real world where data is generated every minute. Therefore we have reached the conclusion that all of these systems can become the ideal choice depending on the user's needs.

9 REFLECTIONS

In this section we will present the reflection we had regarding each of the systems as well as the project in general.
We have to mention that we spent a significant amount of time in trying to work with *Apache Flink*. We wanted to use this stream processing system, in order to compare it with Spark which performs better as a batch processing system. Of course the kind of data that we had did not help us in this direction as we had to split it into streams. Apart from that however, we generally believe that Flink is not easy to use if someone has no previous experience in this system. In order to use it, we had to create executable .Jar files using Java which was not easy at all. Even when more experienced with Java members of our team tried to figure it out, there was no result. We found out that there is an API() available in Python too, but the documentation and the material available online was very limited.
Regarding Spark, we did most of the work using Python, instead of Java which is mostly used, as we were more comfortable with it. However, we had many difficulties in the beginning in implementing most queries in Sql in a Python language environment.
Another major issue we had was related to our dataset. In *Lucene* for instance, that we implemented in .Net, we had to convert our .csv tables in .xml format in order to load them! Converting them was not a very big issue, but their size increased significantly, which made it impossible to use in our local machines. This could be resolved *if we used cloud services* provided from Amazon, however we did not predict this need. Using cloud services to test our platforms, would even make the comparison between the execution times of our systems more reliable as we would be able to run all systems and queries with the same computing power, rather than using different computers.
Finally, we should say that we really liked Neo4J graph processing system as it can create really useful patterns in a user friendly environment. However, we did not have the time to further discover its capabilities and create something more than those quite simple relationships we presented in the previous section.

REFERENCES

[1] https://data.world/data-society/imdb-5000-movie-dataset
[2] https://github.com/GZ-02/Web_Data_Management_Project