ILLINOIS DATA SCIENCE INITIATIVE

TECHNICAL REPORT

Performances Under Reads and Writes

Author:
Richa Misra and Alex O'Kennard

May 5, 2017

Database Performances Under Reads and Writes

ALEX O'KENNARD AND RICHA MISRA[1] AND PROFESSOR ROBERT J. BRUNNER[2]

[1] *National Center For Supercomputing Applications (NCSA)*
[2] *Laboratory for Computation, Data, and Machine Learning*

Compiled May 5, 2017

https://github.com/lcdm-uiuc

1. ABSTRACT

This paper aims to outline the difference between two databases, SparkSQL and PostgreSQL through performance. By taking a look at time logs of each database, the performance of each database can be mapped out and compared to. This paper looks into the properties of the databases performance through its reads and writes.

2. INTRODUCTION

A solid and reliable method of measuring performance of databases is how long the system takes to analyze an input. The responsiveness of a database can become a critical selling point to companies that wish to store their data. We believe that a time-based approach, via time-logs, is the most efficient because one can easily compare time in various databases despite the different sizes of datasets.

The performance of each database can be measured by how fast it processes individual reads and writes. When a database reads information, it analyzes the given data and that data is transformed from hard-disk to memory. Writing is similar in the sense that information is modified and transferred from memory to hard-disk. A database's performance and its efficiency can be based upon its reading and writing speed aside from other contingent factors such as storage, security, and reliability–factors that should be taken into account to for a bigger picture but are not the vital focus of this paper.

Different type of databases have different average speed of reading and writing. In this report, we are primarily comparing a distributed database (SparkSQL) to a more general, traditional database (PostgreSQL). A distributed database is a database that is not attached to the common processor. For example, it may be used/stored in multiple computers or may be 'distributed' over a network of interconnected computers. Traditional databases in that sense only are involved with one computer as they are stored in that computer. For our purposes, we used the NCSA's Nebula cluster in order to test reads and writes.

Overall, the objective of this paper is on performance through measuring the time it takes for a database to read and write. By comparing different types of databases and inputting different sizes of data sizes, we can more accurately research databases that are not only efficient (via their performance) but also its stability in handling various datasets.

3. BACKGROUND

As mentioned earlier, aside from measuring reads and writes, we looked into various factors such as the sizes of datasets used with the two different types of database– distributed and traditional– and chose to use SparkSQL and PostgreSQL.

A. About SparkSQL

SparkSQL is a part of the Apache Spark data framework. Spark, as a cluster computing platform, implements Hadoop and its famous MapReduce model to support a wide range of workloads and computations such as interactive queries and iterative algorithms. Spark's notable feature is its ability to combine different processing types in the same engine while being efficient in maintaining separate tools at the same time

B. About PostgreSQL and Database Management System

PostgreSQL is categorized specifically as Database Management System. A Database Management System (DBMS) is an intricate set of collected software programs that allows users to create, retrieve, update, and manage data in a systematic and orderly fashion. Basically, a DBMS acts as a middleman, safeguarding that data is structured and accessible, between a database and user or even an application that is trying to access data in the dataset. It is important to note the role of DBMS in such PostgreSQL because its database performance is based on how fast the DBMS can extract information–through its queries– and supply it to the user.

C. About Datasets

The sizes of different data sets are also an important factor to the project. The research was conducted on a small set (800 MB) and a relatively large set (31 GB). We went with Amazon and a financial dataset (respectively) as we constructed queries to run on both databases. For data sets such as the financial one, in order to be able to run it on certain databases, it needs to be broken down into little segments and analyzed piece by piece.

4. RUNNING DATASETS ON SPARKSQL

We collected our data with a few python scripts that created tables and then performed repeated queries. In order to get meaningful data, we performed 15,308 total queries on SparkSQL so we can get an idea of the performance. Overall, we did

multiple writes of the same table, selects on columns, selects based on values greater than certain amounts, takes based on selects, and counts based on selects. We chose these methods as they are fairly standard usages of databases, so queries could easily be replicated to whichever database we chose and be standardized. In order to collect data from these queries, first we needed a script to do operations on SparkSQL. Our method was to create log files of terminal output by piping output of a Python script which runs queries to the 'tee' command, which is used to feed terminal output into a file. The script with queries simply performs various queries in a loop to ensure consistency.

```
 for i in range(NUM_QUERIES):
        sqlContext.sql("SELECT Text FROM amazon WHERE
        Id=22010").collect()
stdFinish(0,1)
 for i in range(NUM_QUERIES):
        sqlContext.sql("SELECT COUNT (Score) FROM
        amazon WHERE ProductId='B000E5C1YE' AND
        Score =5").collect()
```

These are example queries that have been looped "NUM QUERIES" times, which for our tests were four times each. Also, we generated ten separate log files of the same types of queries to get a wider range of data points, due to the possibility of the cluster performing inconsistently.

```
  def stdFinish(donewith,nextone):
        print("Just finished "+str(NUM_QUERIES)+"
        of type " + QUERY_LIST[donewith] + ", now
        working on: " + QUERY_LIST[nextone] + "..."
        + ID)
```

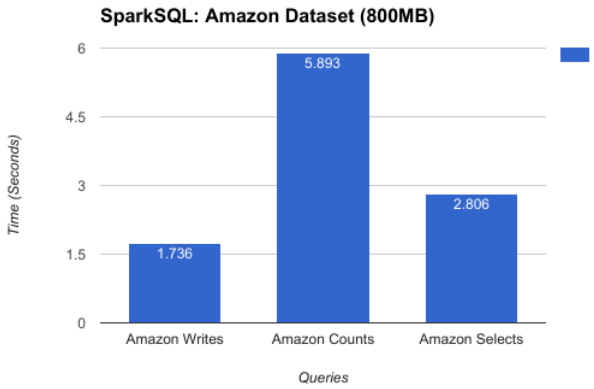
This is the basic formatting function we used to split our chunks of queries up for later gathering. It takes in two integers which are the indices of a "QUERY LIST" that represents shorthand names for the queries we had to perform, such as "Amazon Counts". It prints out a string identifying what queries the script finished, how many, and what queries are next. At the end of the line, there is an 'ID' string ("sparkquery") that is used to identify lines which delimit each block of queries when gathering statistics later. Once we had our generated log file, we called another python script which would clean our log file. Since Spark generates some extraneous data and we were reading in all terminal output, we needed to make sure our logs only contained relevant times.

```
def filterfile(filename):
  with open('output-'+filename,'w') as fout:
     with open(filename,'r') as fin:
       for line in fin:
          if 'DAGScheduler' and '
          finished:' in line or ID in line:
                fout.write(line + "\n")
     print("Filtered file generated..")
filterfile(sys.argv[-1])
```

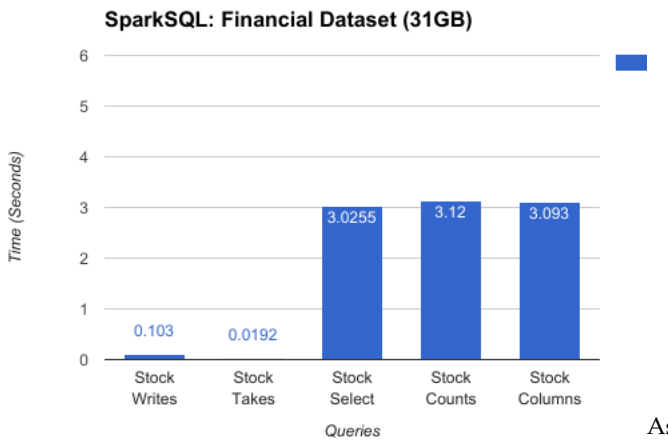
This was our filtering function that removed any data which didn't contain certain keywords which indicated if the output line contained relevant timing information. After getting filtered data into log files which looked like the below:

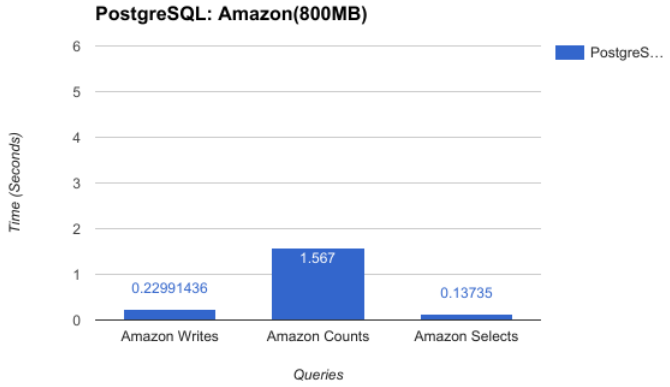
```
17/04/09 21:57:24 INFO DAGScheduler: Job 0 finished:
runJob at PythonRDD.scala:393, took 2.109470 s

17/04/09 21:57:26 INFO DAGScheduler: Job 1 finished:
runJob at PythonRDD.scala:393, took 1.641506 s

17/04/09 21:57:28 INFO DAGScheduler: Job 2 finished:
runJob at PythonRDD.scala:393, took 1.600349 s

17/04/09 21:57:30 INFO DAGScheduler: Job 3 finished:
runJob at PythonRDD.scala:393, took 1.595986 s

Just finished 4 of type Writes basic, now working on:
Basic select ... sparkquery
```

we needed to aggregate the data into something we could easily compare. We had one final script that took in files with a 'log' extension from a directory, converted each of them to an array, and gave some statistics on each array in the form of a dictionary with a query type as a key and performance in seconds/query as a value. Finally, we mapped the dictionary creation to all log files in a folder with logs, merged the dictionaries, and mapped statistics gathering functions so we could have concrete data to compare to other databases. The function 'parse file' takes in a filename and then does the mapping for one file, and then the combination of the shown functions applies it to the directory and reduces it to one dictionary, which is represented in our current graphs. In our current graphs, we can see the average time it took to run reads and writes on Spark SQL. For the Amazon dataset, the total size of Amazon was 800 MB. From our current results, you can see that "Amazon Counts" took a longer time (at approximately six seconds) compared to the two and three seconds for 'Amazon Writes' and 'Amazon Selects'. We can attribute this to the fact that the queries for counting and selecting required more searching than simply writing a table to the database.



For the financial dataset, the average times for 'Stock Writes' and 'Stock Takes' were significantly lower compared to other methods. Because each file was only a few hundred megabytes at the most, writing was fairly fast. Taking is a very simple operation which doesn't have any requirements to select data, so that was also very fast. For 'Stock Select', 'Stock Counts', and 'Stock Columns', the times were each around three seconds, which is still pretty fast considering there was a significant amount of data overall that needed to be selected for each query.





5. RUNNING DATASETS ON POSTGRESQL

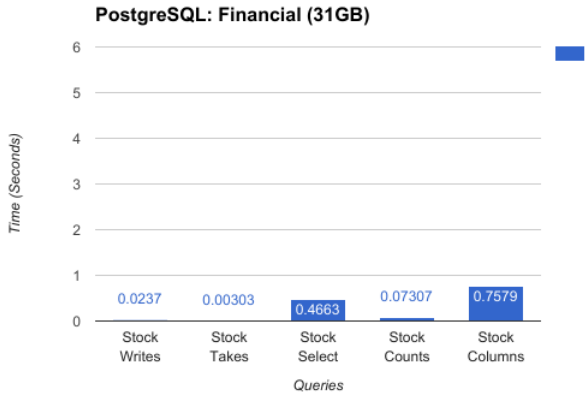
Alike the collection methods used for reading and writing with SparkSQL, we used various python scripts to time and then clean the data. Due to the lack of log output with PostgreSQL, we created our own simple logging function to record the time for each query. On PostgreSQL, we also performed 15,308 total queries, identical in their query structure, on our two datasets to make our comparison to SparkSQL more straightforward. For data collection, we created a new script that has several functions which are simply modifications of the original SparkSQL queries, modified for Psycopg, a Python library which aids in interactions with a PostgreSQL database. As a result of Psycopg's usage in our script, we no longer needed an additional script to clean our data, as our custom logs had no extraneous data like SparkSQL's did. Psycopg's lack of simple table insertion made testing writes more difficult, but the rest of the functions for performing queries were rather straightforward. Below is an example of one function used to collect data from our Amazon dataset queries, where 'time query' is a function we used to time an individual query:

```
def amazon_count(conn):
    for i in range(NUM_QUERIES):
        time_query(conn,"SELECT COUNT (Score)
        FROM amazon WHERE ProductId='B000E5C1YE'
        AND Score =5;")
        time_query(conn,"SELECT COUNT(DISTINCT
        UserId) FROM amazon;")
```

As our methods for data collection were rather similar to what we did for SparkSQL, we used the same logging format and delimiting function, just with 'psqlquery' instead of 'sparkquery'. On the other hand, our data wasn't spread across multiple large log files like SparkSQL was, so we had to slightly modify our statistics gathering functions to also take into account larger logs as well as multiple ones. For the datasets, we used the same sized Amazon and financial stock datasets.

From our current results, you can see that across 1249 total tables, it took about 1.5 seconds for 'Amazon Counts' while 'Amazon Writes' and 'Amazon Selects' were fairly low (0.2 and 0.1 seconds respectively). Not only can this be due to the size of the dataset (which is 800 MB) but also because of PostgreSQL being one of the most optimized databases.

As for the financial dataset, everything ran under one second per query, causing 'Stock Select' and 'Stock Columns' to be the outliers as they run at more that 0.1 seconds each while 'Stock Writes' and 'Stock Takes' and 'Stock Counts' all took similarly fast time. Compared to the speed in the Amazon dataset (slowest being about 1.5 seconds and fastest being 0.137 seconds), the financial dataset, though significantly larger in size, was faster. This could be in part due to Amazon having one 800 MB table compared to the smaller partitioned tables of the financial dataset, resulting in longer times for Amazon's dataset.



6. STATISTICAL ANALYSIS FOR SPARKSQL:

	SparkSQL DEV	Spark Variance	Spark Min
UNITS	secs	secs	secs
Amazon Writes	2.07	4.286	1.59
Amazon Counts	2.855	8.152	0.0152
Amazon Selects	0.8369	0.7004	0.392
Stock Writes	0.121	0.0147	0.006846
Stock Takes	0.0382	0.001	0.006718
Stock Select	2.123	4.509	0.403
Stock Counts	3.927	15.423	0.411
Stock Columns	2.152	4.631	0.427

	Spark Max	Spark(+2 σ)	Spark(-2 σ)
UNITS	secs	secs	secs
Amazon Writes	8.7815	5.875	-2.404
Amazon Counts	17.602	11.603	0.183
Amazon Selects	12.619	4.4798	1.1322
Stock Writes	0.486	0.345	-0.139
Stock Takes	1.126	0.0956	-0.0572
Stock Select	9.144	7.2715	-1.2205
Stock Counts	137.00132	10.974	-4.734
Stock Columns	9.588	7.397	-1.211

7. STATISTICAL ANALYSIS FOR POSTGRESQL:

	Postgres DEV	Postgres Var	Postgres Min
UNITS	secs/query	secs/query	secs
Amazon Writes	0.086	0.0073	0.2197
Amazon Counts	1.43	2.049	0.129
Amazon Selects	0.0223	0.0004987	0.1223
Stock Writes	0.01625	0.000264	0.000394
Stock Takes	0.402	0.1623	0.000091
Stock Select	0.5618	0.3157	0.000087
Stock Counts	0.8705	0.7577	0.000095
Stock Columns	0.962	0.926	0.000076

	Postgres Max	Posgres(+2 σ)	Posgres(-2 σ)
UNITS	secs	secs	secs
Amazon Writes	0.2438	0.4019	0.0579
Amazon Counts	3.015	4.427	-1.293
Amazon Selects	0.1954	0.1819	0.0927
Stock Writes	0.0758	0.0562	-0.0088
Stock Takes	0.0599	0.80703	-0.80097
Stock Select	2.409	1.5899	-0.6573
Stock Counts	0.2799	1.814	-1.667
Stock Columns	2.093	2.6819	-1.16

The statistics on PostgreSQL indicate that aside from logs displaying its superior performance, PostgreSQL is also more reliable and consistent. The statistics for PostgreSQL reveal that if these datasets were to be run again in the same conditions and parameters, the new data points would line up fairly similar to the current ones. Whereas SparkSQL's statistics compared to PostgreSQL indicate that if it were to be run again, the data will be more varied than previous trials, thus more inaccurate, contributing to its inconsistency.

PostgreSQL's standard deviation is not only smaller than SparkSQL's but its even more impressive since PostgreSQL's average time statistics were faster than those of SparkSQL. For example, the standard deviation for PostgreSQL's 'Amazon Counts' at 1.43 seconds and its average time is at 1.567 seconds compared to SparkSQL's 'Amazon Counts' at 2.885 seconds with a mean of 5.893 denote SparkSQL's irregularity as well as its generally worse performance. This further exhibits that PostgreSQL is less likely to have any outlier-like data points since the spread of the data on the bell curve is fairly small.

Lastly, the two standard deviations (2σ) help us understand the extensiveness of the standard deviation data. For example, if we zoom in on SparkSQL's 'Stock Counts' and 'Stock Columns' which has time averages at 3.12 and 3.093 second, their standard deviations at 3.927 and 2.152 seconds indicates a fair spread of data. However, when focusing on the two standard deviations, the difference between the two categories is distinct. This means that when comparing the spread of data, by only looking at one standard deviation, of 'Stock Counts' and 'Stock Columns' at first seems normal however by taking into account two standard deviations, we are able to see exactly how different the ranges of these two categories are compared to each other with 'Stock Counts' having a bigger range of possible data points thus less accuracy.

8. CONCLUSION

From what we found, it appears that PostgreSQL performed better than SparkSQL in all cases. We attribute this to the superior optimizations of PostgreSQL and the possibility that SparkSQL didn't gain an advantage from being a distributed database due to multiple users operating on the cluster. Regardless of circumstantial issues, based on our data we have determined that PostgreSQL is a better choice of database when it comes to reads and writes.

9. SOURCES

System Properties Comparison PostgreSQL vs. Spark SQL. (n.d.). Retrieved May 05, 2017, from https://db-engines.com/en/system/PostgreSQL

"Hadoop vs. Traditional Database: Choose a Big Data Database." Qubole. N.p., 01 Aug. 2016. Web. 14 Apr. 2017.

Minewiskan. "Grant read definition permissions on object metadata (Analysis Services)." Microsoft Docs. N.p., 4 Mar. 2017. Web. 14 Apr. 2017.