Apache Spark Performance Compared to a Traditional Relational Database using Open Source Big Data Health Software

Joshua Powers (jpowers9)
powersj@gatech.edu
April 24, 2016

***Abstract*—The author outlines how big data software can be utilized to speed up health analytics software when faced with big data problems. Specific data analytics from the Observational Health Data Sciences and Informatics (OHDSI) Analytics tool's will be rewritten to demonstrate Apache Spark's ability to more quickly process data with Resilient Distributed Dataset (RDD) in comparison to the use of traditional relational databases such as PostgreSQL.**

***Index Terms*—Apache Spark, Big data, Health analytics, Scala, Python, R, PostgreSQL**

I. Introduction & Motivation

As technology continues to grow into every aspect of humans' lives more and more data are generated [1]. Traditional approaches to data storage heavily rely on relational databases, which can be slow due to either the reliance of a single system and/or disk based access, resulting in a key bottleneck to data processing. Therefore, utilizing modern big data software that are capable of speeding up the processing and analysis of data results in time savings and more time analyzing the data or running additional analyses [2].

II. Problem Formulation

A. Problem Statement

The key deliverable for this study is observing how big data software is well suited to tackling large data sets in a fast and efficient manner. This is demonstrated by taking an open source health analytics tool utilizing a traditional relational database and converting certain long running analytics to showcase the speed advantages of big data software.

B. Background

The Observational Health Data Sciences and Informatics (OHDSI) program is an international network of researchers working together to draw out the value in health data through large-scale analytics by developing various open source software [3]. One of those tools, the Automated Characterization of Health Information at Large-scale Longitudinal Evidence Systems (ACHILLES), provides descriptive statistics in health data. ACHILLES was released by OHDSI in 2014 and is written in the R language [4].

ACHILLES consists of two major components: first, an R package which generates summary statistics and second, a website that enables exploration and visualization of the generated summary statistics. This paper will focus on the former component.

To generate the summary statistics ACHILLES runs dozens of queries against a database in order to collect the various statistics across the OMOP data. The data sets are in the Observational Medical Outcomes Partnership common data model (OMOP CDM) v4 or v5 [5].

C. Data Science Problem

Apache Spark is an open source cluster computing technology that focuses on data parallelism and fault-tolerance. One of the key advantages of Spark is how it provides an in-memory data processing approach, which results in considerably faster execution when repeatedly accessing the data as compared to traditional disk-based access [6]. Given that dozens of queries run against the data by ACHILLES it makes sense to attempt to take advantage of Apache Spark to process the data. This results in considerably faster completion time.

D. Success Metric

The primary success metric in this project will be measuring the increase in speed of the data processing, namely the time it takes to do the analysis. Queries per second or overall latency would have also been interesting to analyze, however a clock time metric was simpler to implement and give a first overall indicator of performance. For the purpose of this project a sufficiently large big data set, GB in size with millions of rows, will be utilized and the data should be accessed repeatedly with ACHILLES [7].

E. Analytic Infrastructure

Two separate infrastructures were setup for this investigation:

The first, will include local baremetal hardware where both the PostgreSQL backed ACHILLES software and a single-node Spark instance were run. This system utilizes an Intel Core i5-3570K processor with 16 GB of memory, a 512 GB SSD, and runs Ubuntu 15.10.

The second, will be built using an Amazon Web Services (AWS) [8] Elastic MapReduce (EMR) cluster [9]. The cluster

Joshua Powers is with School of Computing Georgia Institute of Technology, Atlanta, GA, 30332 USA (http://powersj.github.io).

consists of four memory-optimized compute nodes (r3.xlarge) each with 4 vCPUs, 30 GB of memory, and an 80 GB SSD. The nodes run EMR version 4.5.0. This infrastructure will be used to run Spark in a scaled and distributed environment.

Both infrastructures used Apache Spark 1.6.1 and Apache Hadoop 2.7.2. The Scala-based Spark application will use OpenJDK 7, SBT 0.13.8, and Scala 2.10.6.

III. Approach & Implementation

The project setup and configuration consisted of four separate phases:

1) Generate the OMOP CDM v5 formatted data
2) Setup and configure the PostgreSQL database
3) Setup and run the ACHILLES software
4) Develop and run the Spark based queries

The following sections outline the approach and implementation of each phase.

A. Data Generation

The first phase was designed to obtain a large data set to be used by ACHILLES in the OMOP CMD v5 format. The OHDSI's ETL-CMS tool [10] obtained data from the Centers for Medicare and Medicaid Services (CMS) Research, Statistics, Data, and Systems public use files in the OMOP CDM v5 format [11].

The software came with a mechanism to download the raw data and transform it into basic CSV formatted files. After a brief setup and configuration of various system variables the script downloaded the DE_4 data set. The next step required the data to be processed against a particular vocabulary used by the data format to ensure consistency. The result was a raw data set ready for use with the following statistics:

Raw Data Set Statistics	
Num of Beneficiary Records	343,507
Num of Carrier Records	9,602,438
Num of In Patient Records	248,784
Num of Out Patient Records	1,695,249
Num of Prescription Records	11,214,419
Data Set Disk Size	3.0 GB

B. PostgreSQL

The second phase involved setting up and loading the generated data into a PostgreSQL database. While ACHILLES has support for Oracle and MS SQL database back ends, PostgreSQL, an open source, free, and easy to install and configure on the Ubuntu system, which was utilized for this project.

For this phase the OHDSI had another set of scripts available under the name, Common Data Model [12], that allowed loading and transformation of the CDM v5 format data into a database. Here are the following steps that the tools or author used during this phase:

1) Create an empty schema in the database (by hand)
2) Create tables and fields into the schema for the CDM (automated)
3) Load data into the schema (modified OHDSI script)
4) Add constraints including primary and foreign keys (automated)
5) Add a minimum set of indexes to the data (automated)

A major issue was discovered while loading the data into the database schema: a number of the index/key columns contained non-numeric values (e.g. 'val83'), however the database scheme expected only numeric values and not varchar values. Rather than modifying the data, the database scheme was altered to allow varchar data type instead of numeric. Removal of the non-numeric characters from these values should have been done so as to maintain the numeric type as this would later cause issues.

The last configuration step was to enable a mechanism for benchmarking the ACHILLES analytics when querying PostgreSQL. This was done by modifying the PostgreSQL configuration by setting the 'log_min_duration_statement' to be equal to zero. This would cause all queries' execution time to be logged in milliseconds.

Finally, to ensure that the data was sufficiently large some initial data statistics data were captured:

Database Statistics	
Table 'Condition Occurrence' Rows	19,166,896
Table 'Provider Rows'	10,652,520
Table 'Procedure Cost' Rows	9,378,683
Database Disk Size	9.3 GB

It was interesting to discover that the fully indexed database uses more than three times the disk space as the raw CSV files.

C. ACHILLES

Next, the third phase focused on getting ACHILLES software operational in order to benchmark the queries against the database; the second component, ACHILLES HEEL, was not run. The latest version of ACHILLES was obtained from the GitHub repo and R Studio and was used for development, as suggested by the developers of ACHILLES themselves. R Studio easily installed all the necessary dependencies required by ACHILLES.

ACHILLES runs over 150 various analytics divided into groups numbered by the hundreds (e.g. 100s, 200s, 300s, 400s, etc.). The author's first attempt at running ACHILLES resulted in numerous failures. These were a result of the changes that had been made earlier during data load into the database due to the non-numeric values for some indices. To overcome these issues groups 1300, 1500, and 1600 were removed. The author determined that it was preferable to focus on the Spark aspect of the project and not go back and fix the data issues. The ACHILLES software then began generating the statistics and completing the process.

The next section will describe how ACHILLES was benchmarked and results were obtained.

D. Spark

In the final phase of implementation, in order to demonstrate how a Spark backed system could outperform a database a small Scala program was developed. To begin, the data from each table of the PostgreSQL database was exported directly into CSV files to use as the raw data. Next, work began on the simple Scala program that would import the required data. SparkSQL [13], a module for working with structured data via SQL-like queries, allowed for easy load operations. The author did attempt to replicate ACHILLES queries using SparkSQL, but did not complete a full performance evaluation due to limited time. Some queries were far too complex to implement with somewhat limited functionality.

Next, the data is inserted into Resilient Distributed Datasets (RDD) [14] as the built-in RDD functions could then be used to manipulate the data. RDDs, the basic data objects in Spark, provide an immutable, partitioned collection of elements that can be operated on in parallel. With traditional data structures data is manipulated at each step, however with large data sets doing so could cause large amounts of thrashing. In order to avoid this, RDDs only complete transactions once an operation (e.g. count, collect, reduce, etc.) is called that generates a new value against a RDD. Other transformations (e.g. map, filter, sort, etc.) are lazy and held until an operation is called to generate a pipeline.

Finally, each identified query was put into its own function. The function takes the required data for the group analytic, captures the system time before and after any data transformations occur, and executes RDD operations to duplicate the ACHILLES SQL queries. Each query was generated to replicate the same function that the query was performing: usually a count of some set of values. It was found that there were some nontrivial queries that would be better suited for use by SparkSQL or Apache Hive [15].

The next section will describe how the Spark-based analytics were benchmarked and results were obtained.

IV. Experiment Design & Evaluation

This section presents the benchmarks produced by the PostgreSQL-backed ACHILLES, how the key ACHILLES analytics were identified for conversion to Spark, and the results and discussion of the Spark-based queries.

A. PostgreSQL Benchmarks

In order to determine how much faster Spark can run various analytics, the first step was to understand how fast or slow various analytics were taking in PostgreSQL.

As stated above, the timing for each ACHILLES analytic was determined by enabling the duration function in PostgreSQL. The first step was to capture the data for all the queries as a whole so as to get a big picture view of the query times. Figure 1 shows the query time for every Achilles analytic that was run. Keep in mind that three groups of queries (i.e. 1300, 1500, and 1600) had to be removed. The result was somewhat surprising: the vast majority of analytics are incredibly fast running taking less than 1 millisecond and the long running queries were coming from a clustered group of queries. The table below outlines the analytic breakdown by time:

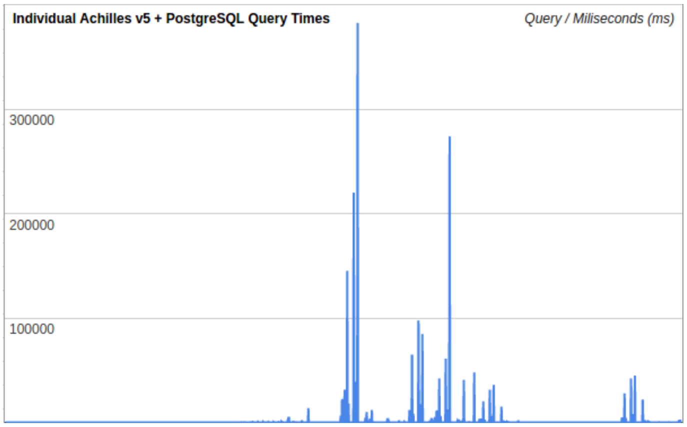

Fig. 1. Query time in milliseconds for every Achilles analytic using PostgreSQL

ACHILLES + PostgreSQL Analytic Runtime		
Execution Time Range	# of Analytics	% of Total
<1ms	1407	89.85%
1ms <100ms	45	2.87%
100ms <1s	44	2.81%
1s <10s	40	2.55%
>10s	30	1.92%

The next step was to identify which groups contained these very long running queries. In order to do this, ACHILLES was run again by group and the total time for every query captured. This provided insight into what groups should be targeted for conversion to Spark. Figure 2 shows the results by group. Here five of the groups were easily identifiable as containing very long running queries. Groups 400, 600, 700, 800, and 1800.

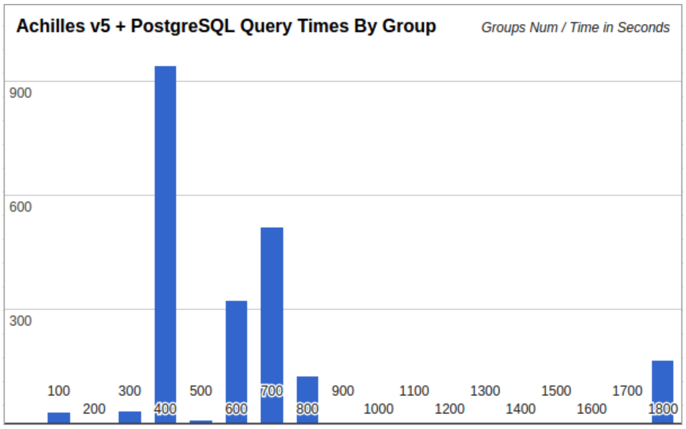

Fig. 2. Query time in milliseconds for every Achilles analytic by group using PostgreSQL

Group 400 is an obvious choice to target for conversion. As the longest running group it seems prime to show how Spark could be used, but it is also important because it utilizes a very large data set.

Group 300 however is also interesting because it has three fairly simple queries that take a significant amount of time per query, more so than some of the other groups.

In the next phase, when the targeted and specific queries in Groups 300 and 400 are run the three tables and data are used:

Spark Data Set Statistics		
Table Name	Size	Number of Rows
Condition Occurrence	1.4 GB	19,166,855
Provider	348 MB	10,652,423
Care Site	9.7 MB	231,665

The result of this investigation demonstrated two key factors: first, the overall timing of every analytic and second, a targeted set of groups attempted to migrate and reproduce in Spark. The overall timing helped prevent wasted time on analytics that are already very fast and the targeted set meant an opportunity to showcase how much faster Spark could possibly be.

B. Spark Benchmarks

This final section will present the overall results, discuss Spark load times, and explain the results from the Group 300 and Group 400 results.

First, as mentioned in the explanation of infrastructure, the Spark test would be run twice: first, on the same single node as the ACHILLES analytics and again on an AWS EMR cluster. Below are the results in terms of speedup as compared to the PostgreSQL results, where a value of one is the same speed as PostgreSQL, a number greater than one represents how many times faster Spark was, a number less than one is how many times slower:

Relative Spark Timing Results		
Analytic #	1-node Spark	4-node Spark
300	6.79	5.51
301	18.23	13.13
302	5.59	5.14
400	0.12	0.46
401	0.03	0.12
405	0.04	0.15
409	15.56	245.03
411	0.75	13.06
412	0.41	3.38
413	1.52	11.28

Beginning with the three group 300 analytics the performance increase was quite drastic. These queries were primarily conducting maps in addition to counting or distinct and then counts. The three queries averaged less than one second with an overall savings of 30 seconds! See Figure 3 as these analytics were a perfect example of how using Spark and in-memory computation can save considerable amounts of time. An interesting consequence of the 4-node cluster was a slight increase in overall time due to the distributed nature and smaller data set size. The overall result was still considerably faster in either case with Spark.

The first group of 400 analytics, 400, 401, and 405 produced results across the board that were found to be inferior to PostgreSQL, as shown in Figure 4. The author at first thought that something was terribly wrong with the data or query, however it was later discovered that these three analytics all had one thing in common: a group by in the statement. Group by is traditionally an expensive operation due to the need to combine like elements and requires all the data to be reshuffled across nodes. Comparing the 1-node to the 4-node case, the additional nodes helped process the data four times as much, however the PostgreSQL was still able to outperform Spark. The author wonders if using a Data Frame over an RDD would have helped or if indexing would provide any kind of optimization for PostgreSQL.

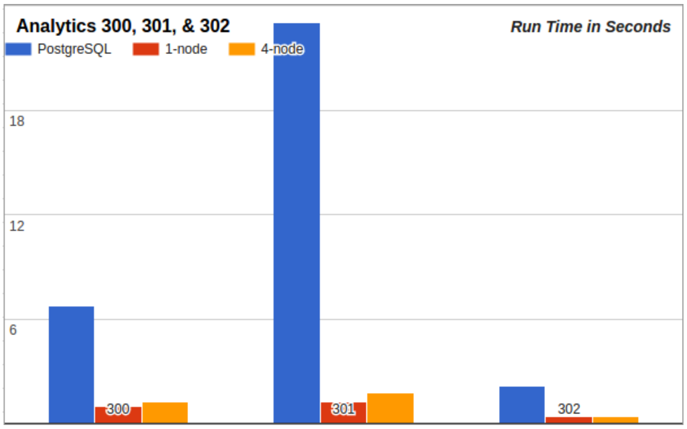

Fig. 3. Analytic 300, 301, & 302 Run Time in Seconds

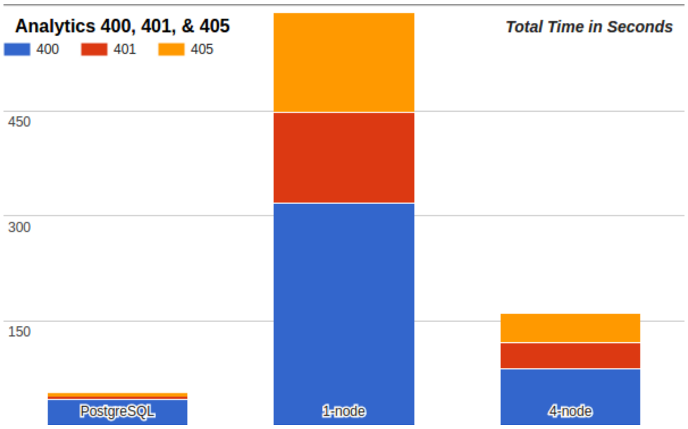

Fig. 4. Analytic 400, 401, & 405 Total Time in Seconds

Turning to the final four group 400 analytics, 409, 411, 412, and 413, these queries were principally taking specific data and counting the number of records. To put wall clock speeds on this set of results, see Figure 5. PostgreSQL ran for 50 seconds, the 1-node Spark for 10 seconds, and the 4-node Spark for only 1 second. The larger data set took advantage of being distributed and in-memory and therefore allowed for much faster processing time. The distributed result also showed how having a scaled infrastructure allowed for even better processing times of the data by a factor of 10 compared to the single node.

Finally, when describing Spark a discussion about load times also needs to be made. Due to the fact that Spark will only complete actions on an RDD after an operation is called the author forced a cache plus collect to occur before running

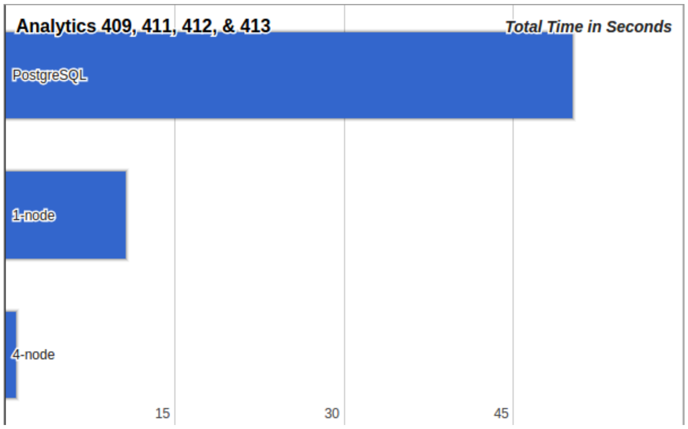

Fig. 5. Analytic 409, 411, 412, & 413 Total Time in Seconds

any of the queries. This can be a time consuming task as show in the table below:

Spark Caching Time in Seconds		
Analytic Group	1-node Spark	4-node Spark
300	77.7	74.8
400	231.6	163.0

However, it needs to be established that Spark does require additional time to initially get the data in place during the first operation, but then all operations after that would demonstrate the same speedup as shown in the results. The intent behind this project was to try and compare query run times between PostgreSQL's pre-loaded and indexed data and Spark, not including the necessary data load time.

V. Conclusion

Ultimately Spark showed how processing in-memory can lead to the faster processing of data. However, in order to get these advantages the data must be loaded first and the queries optimized in terms of data access. The results also demonstrated how a fully indexed database when only a single node is available, can perform very well over large quantities of data.

VI. Supplemental Material

Links to the code for this project and a video presentation about the topic are both available below:

1) Code: https://github.com/powersj/spark4achilles
2) Video: https://www.youtube.com/watch?v=k5bl7VhgEmQ

VII. Acknowledgments

The author wishes to thank Dr. Jimeng Sun, The SunLab, the CSE8803 Big Data Analytics for Health Care TAs for their instruction and guidance, the OHDSI for the open source software and tools, Dr. Walter and Marjie Powers, Olga Martyusheva, and Alex Balderson for their feedback, evaluation, and support.

References

[1] Stephen Kaisler, Frank Armour, Juan Antonio Espinosa, and William Money. Big data: Issues and challenges moving forward. In *System Sciences (HICSS), 2013 46th Hawaii International Conference on*, pages 995–1004. IEEE, 2013.

[2] Rubén Casado and Muhammad Younas. Emerging trends and technologies in big data processing. *Concurrency and Computation: Practice and Experience*, 27(8):2078–2091, 2015.

[3] Observational health data sciences and informatics, 2016.

[4] Achilles for data characterization, 2016.

[5] G Hripcsak, JD Duke, NH Shah, CG Reich, V Huser, MJ Schuemie, MA Suchard, RW Park, ICK Wong, PR Rijnbeek, et al. Observational health data sciences and informatics (ohdsi): opportunities for observational researchers. *MEDINFO*, 15, 2015.

[6] Ravi Narasimhan and T Bhuvaneshwari. Big dataa brief study. *Int. J. Sci. Eng. Res*, 5:1–4, 2014.

[7] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[8] Amazon web services (aws) - cloud computing services, 2016.

[9] Elastic mapreduce (emr) - amazon web services, 2016.

[10] Etl-cms, 2016.

[11] Medicare claims synthetic public use files (synpufs), 2016.

[12] Ohdsi common data model, 2016.

[13] Spark sql, 2016.

[14] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.

[15] Apache hive, 2016.