Data 516 Project Final Report

By: Marques Chacon and Ramiro Steinmann Petrasso

Introduction

For our project, we decided to benchmark two systems running the same queries on the same dataset, to compare their performances and see whether one of the systems was consistently faster than the other, or if the two systems each had conditions under which they were faster than the other.

Our motivation for this project was to learn more about a system with which we were unfamiliar, by comparing it to one that we had extensive previous experience with. For this reason, we decided to pick Spark and a standard SQL Server database. In previous data management classes, such as DATA 514, we'd worked a lot with regular relational databases from a variety of providers, such as local sqlLite databases, Amazon RDS MySQL databases, and AzureSQL SQL Server databases. Meanwhile, both of us had zero prior experience working with Spark. We had little idea what to expect when comparing it with the standard database setup we were familiar with, and theorized that working with the project would help us build on top of what we were going to learn about it in this course.

When deciding how to set up both systems, we decided to frame the project from the perspective of personal and low-budget projects, and thus set both systems up with the least expensive specs possible. We did this because we wanted to minimize our own expenses for this project (to zero, if possible), but also because if we ourselves were picking a data system in the future, it would likely be for our own personal projects, because when working in a professional capacity it was likely that we would simply be told what system our projects would be run on.

We picked a dataset to benchmark our systems using a similar thought process. Because we were framing our project from the perspective of smaller users and personal projects, we decided to use a relatively large dataset that we would scrape ourselves, since datasets for personal projects that called for the use of data systems would likely be structured similarly. Using nba_api, a package for scraping data from the official NBA website, we scraped a list of all shots from all regular season games they had data for up until the 2022 season. We kept the dataset unnormalized, as it looked in the individual API requests, intentionally, because we figured it was likely that most developers would not follow normal form rules while working on their personal projects.

Evaluated Systems

For this project, we worked on two different systems: Spark and a standard SQL Server Database. We aimed to configure both on the same platform, if possible, and to set both up on the minimum price point they offered, in order to evaluate which of the two systems runs faster for smaller projects that may have a limited budget. We picked Azure as the platform we would use because of past familiarity with it, as well as the fact that it had offerings for both Spark and standard SQL databases.

For running Spark, we set up a cluster on Azure Databricks. Our minimally spec'd cluster was set up with Databricks runtime version 13.3 LTS, running on a single node. Our node had a node type of Standard_DS3_v2, which has a 14 GB memory capacity and 4 cores.

For running a SQL database, we used AzureSQL to set up a SQL Server Database. We set up our database server with 1 vcore and a 32 GB memory capacity.

After setting up the database and the Databricks cluster, we uploaded the exact same dataset to both of them. Our dataset was a list of every shot attempted at a regular season NBA game, from the season starting in 1996 to the season ending in 2022. The idea behind using this dataset was that it would look similar to other datasets generated for small personal projects, which would likely use minimum specs for any data system they find necessary to use in order to minimize expenses. We scraped the dataset using a Python script and the nba_api package and stored it in CSV format for importing into both systems.

The dataset is almost entirely contained in a single CSV/table, called "ShotData". This table contains one row/tuple for each individual shot and contains most relevant information to each shot, such as the player who attempted it was, when the shot was made, what team it was made for, who the home and away team was, which court region and which position the shot was made from, etc. The CSV for this table is approximately 1.01 GB in size and has a cardinality of 4,966,816. The only other table in the dataset is a small auxiliary table containing all different names each team has gone by, as well as their corresponding numerical ID and three letter abbreviation. This is a very small table, with a cardinality of 39.

Problem Statement and Methods

We wanted to answer the question, "How does the performance of Azure with Spark compare to a regular Azure SQL database?" To answer this question, we created different types of SQL queries, as well as statistically analyzed various other queries to see how their performances differed. We also took a look at how different aspects of a query affected the runtime, including "simple" queries and "complex queries", and the number of joins within a query.

We first constructed seven different queries that were transferable to both Spark SQL and regular Azure SQL. These queries answered different questions about our dataset, including the following:

1. Have popular "regions" to attempt shots in an NBA court shifted over time?
2. What is the average field goal percentage for home teams vs. away teams, and how has this changed over time?
3. What are the top 10 players who have attempted the most 3 point shots, and what percentage of the time do each of those players prefer Layup Shots, Jump Shots, Slam Dunk Shots, etc.?
4. What section of the court does each player shoot the most from? Which section does each player have the highest field goal percentage?
5. What is the average shot distance per period?
6. What team has the highest field goal percentage for any team in any year? Which has the lowest?
7. What is the average points per attempt for each player with a minimum of 300 attempts? Who has the highest during the last 5 minutes of a game?

For each query, we ran them once in both Azure Databricks (which runs Spark) and Azure Data Studio (which holds a regular SQL database). By getting data on each of these 7 queries, we were not only able to compare how a query performed on Spark vs SQL, but also see if there were any differences in savings for different queries. For instance, if a query was innately more complex, one may assume that the savings would be bigger than if the query was very simple. Our specific queries for these questions can be found in the azure_queries folder.

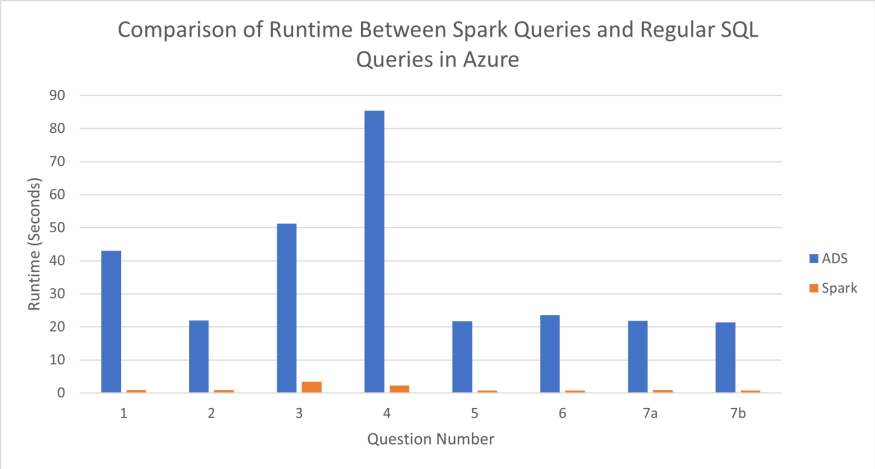
While we compared the runtimes for each of these queries, we also took a more statistical approach to answering our research question, by generating 30 different runs under different conditions. For instance, we did 30 runs of a simple Spark query and 30 runs of the same query in Azure Data Studio. Then we compared that with a

complicated query, and generated 30 runs in both Spark and ADS. From there, we constructed a two-sample t test to compare the mean runtimes for each condition. For instance, we would compare the runtimes of running a simple query in spark, and see if it was significantly different than running it in ADS. Similarly we looked at any differences for a more complicated query, creating a similar test for that scenario. Then, we compared a simple and complex query in the same environment (Spark vs No Spark) to see if complexity truly affected the runtime.

After analyzing the differences in runtime between queries run with Spark and those without, we considered only queries run in Spark, and then compared how the number of joins would affect the runtime. We only considered Spark queries because running even just the query with no joins in Azure Data Studio took 35 minutes, and the queries with one or more joins timed out.

Results

After running the seven different queries that answered each question, we entered the runtimes into an Excel spreadsheet and then plotted the results. The paired bar graph below shows our results:

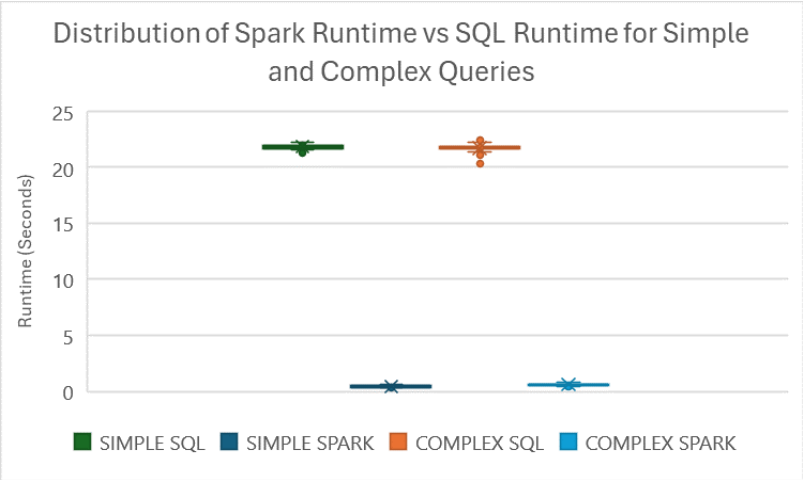


From the plot, you can clearly see that the queries running on a regular SQL database take much longer, with the least complex query running for 20 seconds while the most complex running for 85 seconds. Alternatively, the queries running on Spark are quick, with the most complex queries taking around 4-5 seconds to complete, and most others taking under a second to run. This graph seems to show the obvious benefit of using Spark, regardless of the type of query that we are running, as it outclasses regular SQL in every way. However, one thing that struck us as peculiar was that the query for question 3 took longer to execute in Spark than the query for question 4, while in the SQL database, question 4 took much longer. We figure that it could be due to different execution plans between each system, but this warrants a deeper dive in future research.

The next part of our analysis focused on statistically analyzing the differences in runtime by looking specifically at two-sample t tests that compare the average runtime from a set of 30 runs in both a Spark and SQL environment. Furthermore, we subdivided these conditions into “simple” and “complex” to denote the complexity of the query. Our simple query gathered all of the players in the dataset who have missed a shot, and sorted the result in descending order alphabetically. This can be represented as follows:

- SELECT DISTINCT PLAYER_NAME FROM shotdata WHERE SHOT_MADE_FLAG = 0 ORDER BY PLAYER_NAME DESC

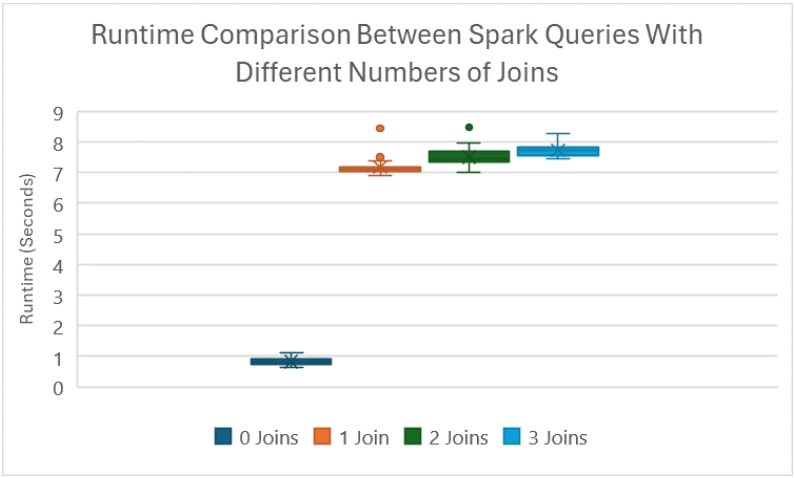
For our complex query, we reused the query that answered the question of players who had the highest points per attempt in the last 5 minutes of a game. Due to the length of this query, we will not show it in this report, but it can be found in the azure_queries folder. Here are our results for the different queries:



As we noted before, the Spark queries outclassed the regular SQL queries, with all of them taking under a second in both the simple and complex case, while most of the regular SQL queries took between 21-22 seconds. We then compared the means for each of these groups to see if there were any distinct differences, and through several different two-sample t tests, we found the following:

- Comparing Simple SQL to Simple Spark and Complex SQL to Complex Spark yielded very low p-values of $2.2 * 10^{-16}$ each, indicating a significant difference in means when comparing across systems
- Comparing Simple and Complex queries in Spark resulted in a meaningful difference, with a p-value of $8.46 * 10^{-9}$, but Simple and Complex queries in SQL did not result in meaningful differences, with a p-value of 0.159.

Finally, we wanted to see how different numbers of joins would affect the runtime in Spark. As mentioned earlier, queries with at least 1 join timed out in our regular SQL Server database, so these results will summarize only Spark. We used a simple "SELECT * FROM shotdata" query and then joined on GAME_ID once, twice, and thrice to get the results for one, two, and three joins respectively.



From this graph we can see that the more joins there are in the query, the longer it takes to complete the query. Interestingly, most of the increase in execution time came from 0 joins to 1 join, but only marginally did it increase from 1 to 2 joins and 2 to 3 joins. This could be due to Spark caching the result of each joined table so the additional time to join others is negligible. To assess these differences statistically, we looked at 3 different two sample t-tests and each one was significant:

- Comparing 0 joins to 1 join resulted in a p-value of $2.2 * 10^{-16}$
- Comparing 1 join to 2 joins resulted in a p-value of $4.3 * 10^{-5}$
- Comparing 2 joins to 3 joins results in a p-value of 0.001

Conclusion

While we originally thought Spark would outperform the regular SQL Server database, the extent to which it did so greatly surprised us. Spark completely outclassed the SQL Server database by every metric we threw at it, with the queries run on the Databricks cluster regularly running around 20 times faster than on the database.

While Spark outperformed the SQL Server consistently, it's worth noting that the difference in runtime greatly varied from query to query, likely due to differences in the execution plans chosen by each system. For example our queries for question 3 and 4 were both slower when run on the database compared to on the Databricks cluster; however, question 3 ran faster than question 4 on the database, while running slower than question 4 on the Databricks cluster. This logically means that the difference in runtimes was significantly greater for question 4 than question 3.

While this project saw Spark completely outclass the regular SQL database it was compared to, we see several future avenues of research that can further develop the comparison between the two systems. One potential future avenue is comparing the two on datasets of different sizes. One of the readings we covered in this course pointed out that, due to overhead costs, big data systems are often overused. While our dataset was not overly large, it is possible that benchmarking the systems on smaller ones could see the differences in runtime drop. Another potential avenue of future research is comparing Spark/Databricks to a system run on a local machine rather than another system running on the cloud, since locally-run systems are common in personal projects.

Overall, this benchmarking project left us with a positive impression of working with Spark, both in terms of performance and ease of use, and we recommend other smaller developers to consider it for their own projects as well.