DATA 516 Final Project – Part II
Apache Spark Timing Exercises on AWS EMR

Rex Thompson
rext@uw.edu

DATA 516, Autumn 2017
University of Washington
12/12/2017

Abstract – **We perform experiments in Apache Spark on Amazon Web Service's (AWS) Elastic Map Reduce (EMR) to time how long it takes to fit a Gaussian Mixture Model (GMM) and to perform a simple join of two dataframes. Specifically, we explore how cluster size and data partition count interact to affect the time it takes to fit a GMM. We show that for some clusters, increasing node count has no effect on performance, and we find that performance improves with increasing data partition count until a certain point at which additional partitions no longer affect execution time. We also explore how data partitioning and data broadcasting interact to affect the time it takes to perform a simple join. We show that for some clusters, performance is not affected by partition count of either of the two joined dataframes, nor is it affected by the broadcast status of one of the two dataframes. We show that partition count may affect the time it takes to operate on the join's resulting relation; however, our results are inconsistent and therefore we cannot definitively confirm to what extent this has an effect, if at all.**

1. INTRODUCTION

Apache Spark is a distributed computing platform built on top of the Hadoop Map-Reduce framework. The main abstraction in Spark is that of the Resilient Distributed Datasets (RDDS), which is a read-only and fault-tolerant abstraction for in-memory cluster computing. Spark offers a variety of libraries for big data processing including: Spark SQL for relational databases; Spark Streaming for online transaction processing (OLTP) and streaming services; ml and MLlib libraries for machine learning; and GraphX for graph processing applications. [1, 2]

In a prior work, we explored speed-up and scale-up in Apache Spark by fitting a 7-component Gaussian Mixture Model (GMM) to datasets of various sizes and on clusters of various sizes.

We explored speed-up by fitting 7-component GMMs on a dataset with 1.9 million rows and four columns, using 2-, 4- and 8-node m3.2xlarge clusters. The system automatically split the data into four partitions. We were expecting that increasing cluster size would result in faster execution times; however, we were surprised to observe only a slight improvement in run times for each cluster size doubling. The m3.2xlarge clusters have 16 node slices per cluster, so we concluded that the process was simply being overwhelmed by cross-communication, leading to nearly consistent run times across the clusters of varying sizes.

We explored scale-up on a 2-node m3.2xlarge cluster. We used a larger version of the dataset from before (with 15.6 million rows) and took random samples of 1, 5, 10, 25, 50 and 75% of the data. We fit 7-component GMMs on the random subsets as well as on the full large dataset, each of which was automatically split into 8 partitions. As expected, we observed near-linear scale-up: the execution time increased roughly linearly with the proportion of the full dataset used.

In this paper, we explore different components of Apache Spark's performance by answering the two research questions below which were inspired by our prior work:

RQ1. **GMM Cluster Size and Partition Count:** How do cluster size and partition count interact to affect the time it takes to fit a GMM?

RQ2. **Join Partitions and Broadcast State:** How do partition count and data broadcasting interact to affect the time it takes to perform a join?

In Section 2 we describe the ecosystem and data used in this study. We summarize our experiment setup and findings for each of the two research questions presented above in Sections 3 and 4, respectively. Finally, we summarize our conclusions in Section 5.

2. EXPERIMENT ECOSYSTEM AND DATA

We used Amazon Elastic Map Reduce (EMR) clusters of type m2.2xlarge and of various sizes as described in the relevant sections below. Each cluster had 4 vCPU (i.e. four slices per node), 34.2 GiB of memory, and 850 SSD GB storage. We accessed the clusters in Python (Version 2.7.12) through a Jupyter Notebook which we bootstrapped at cluster setup. We used the PySpark Dataframe API (version 2.2.0) with the 'pyspark.ml' and 'pyspark.sql' modules for the GMM and join exercises. We used the 'time' module to record how long it took to execute select code chunks.

The aforementioned tools were all accessed with a 2015 MacBook Pro running MacOS High Sierra (Version 10.13.1). However, note that this machine did not perform any of the timed exercises itself; instead, all computations and timing exercises were performed on the Amazon EMR clusters.

For the GMM exercises we used sample data from the Sloan Digital Sky Survey. The data was 98.3 MB in size with 1,946,979 rows and four columns (excluding row IDs). The data was stored on Amazon's Simple Storage Service (S3).

For the join exercises, we used paid on-street parking data provided by the City of Seattle. We stored this data on S3 as well. The data consists of two related tables:

- **Transaction data:** parking transactions from Seattle pay stations. Includes date/time, duration, payment amount and method (e.g. card, cash, mobile app), meter ID, blockface ID, etc. The data is 5.32 GB in size with 62,327,970 rows (excluding header) and 10 columns.
- **Blockface data:** Seattle block-level parking-related attributes. Includes paid parking times and rates, Peak Hour parking restrictions, street capacity, neighborhood, record effective dates, etc. The data is 2.7 MB in size with 13,706 rows (excluding header) and 39 columns.

Both datasets were made publicly available and accessible by the City of Seattle. The terms of use require the inclusion of the following disclaimers:

> *The data made available here has been modified for use from its original source, which is the City of Seattle. Neither the City of Seattle nor the Office of the Chief Technology Officer (OCTO) makes any claims as to the completeness, timeliness, accuracy or content of any data contained in this application; makes any representation of any kind, including, but not limited to, warranty of the accuracy or fitness for a particular use; nor are any such warranties to be implied or inferred with respect to the information or data furnished herein. The data is subject to change as modifications and updates are complete. It is understood that the information contained in the web feed is being used at one's own risk.*

3. RQ1: GMM Cluster Size and Partition Count

I. Background

In our previous work, we explored Spark's speed-up and scale-up by fitting GMMs on datasets and clusters of various sizes. In each case we had used the default number of data partitions as decided by the YARN resource manager. While the near-linear scale-up had matched our expectations, we were surprised at the lack of speed-up as we saw only very mild gains in performance with each doubling of cluster size.

We suspected that the large node slice count in our prior work (i.e. 16/node) might have overwhelmed the execution, resulting in the non-linear scale-up we observed. We had also read that partitioning the data may increase performance when using the 'pyspark.ml' library. This led us to our research question: **How do cluster size and partition count interact to affect the time it takes to fit a GMM?**

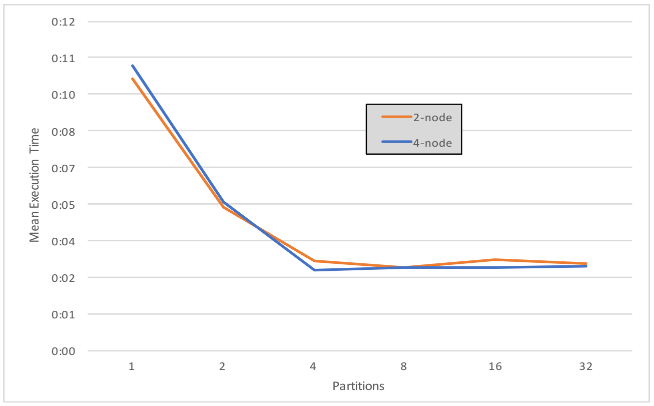

Figure 1: GMM fitting time by data partition count and cluster size

II. Experiment Setup

We used a 2-node m2.2xlarge cluster to fit 7-component GMMs on the small Sloan Digital Sky Survey dataset from our prior work. With only 4 slices per node, the 2-node m2.2xlarge cluster had a total of only 8 node slices. This is only one quarter of even the smallest of the clusters we had used in our prior work, in which we used 2-, 4- and 8-node m3.2xlarge clusters with 16 slices per node for a total of 32, 64, and 128 node slices. We specifically chose this smaller cluster configuration in response to our assumption that the massive node slice count had resulted in excessive cross-communication which overwhelmed the processing time in our prior work.

We used the 'df.rdd.repartition()' command to split the data into 1, 2, 4, 8, 16 and 32 partitions. We timed how long it took to fit the GMM for each partition configuration, and we repeated each experiment six times to yield an average execution time. We then repeated the exercise on a 4-node cluster of the same type.

III. Results

Our results are presented in Figure 1 above. As shown, for both cluster sizes the execution times roughly halved for each doubling of partition count from 1 to 2 to 4, but then leveled off for partition counts of 8 and above. This confirms that repartitioning the data can in fact improve performance; however, it appears that this improvement only scales to a certain point. Additional partitions beyond 4 did not appear to slow the process down; however, they did not speed it up either.

Surprisingly, we observed essentially no improvement on the 4-node cluster compared to the 2-node cluster. In fact, the average times were almost identical for both cluster sizes at each of the partition configurations. This was a surprising observation since we expected that the lower node slice count would result in an improvement in performance when changing cluster size, seeing as in our previous work we acknowledged that we most likely overwhelmed the process with too many node slices from the beginning.

4. RQ2: Join Partitions and Broadcast State

I. *Background*

We had performed extensive experiments fitting GMMs using Spark via PySpark in our previous work. However, with our surprise at what little effect cluster size had on the times to fit the GMMs in our previous exercise, we were curious to see how Spark would handle other common types of operations.

We decided to test how Spark handled a simple data join in different configurations. Specifically, we hoped to answer the following question: **How do partition count and data broadcasting interact to affect the time it takes to perform a join?**

II. *Experiment Setup*

We used a 2-node m2.2xlarge cluster (with 4 slices/node, for a total of 8 node slices) to join Transaction and Blockface data from the City of Seattle paid on-street parking dataset on ElementKey. The Blockface data was small, at only 2.7 MB. However, the Transaction data was quite large, at over 5 GB. We used the following join command in 'pyspark.sql':

```
df = spark.sql("SELECT * \
      FROM transactions t LEFT JOIN blockface b \
      ON t.ElementKey = b.ElementKey \
      WHERE (t.TransactionDateTime < b.EffectiveEndDate \
             OR b.EffectiveEndDate IS NULL) \
      AND t.TransactionDateTime >= b.EffectiveStartDate")
```

This join combined the dataframes on the ElementKey column, and then filtered the results based on the transaction data's TransactinDateTime column values compared to the EffectiveStart/EndDate records in the Blockface data.

First, we split the Transaction data into partition counts of 1, 2, 4, 8, 16, 32, 64 and 128, and we split the Blockface data into partition counts of 1, 2 and 4. For each configuration permutation we performed the merge shown above six times. We assumed that splitting the large transaction dataset into several partitions would speed up the execution time.

We also repeated the experiment above, but this time we broadcast the Blockface data and used fewer partition options than the previous experiment, this time only splitting the transaction data into 1, 4, 16 and 64 partitions. We again split the Blockface data into 1, 2 and 4 partitions, and again repeated the timing for each configuration permutation six times. We had read about broadcasting, and we assumed that sending the smaller Blockface data to all nodes would speed up the processing time by cutting down on the amount of cross-node communication needed for this join.

We repeated the experiment once more, but this time we used the default number of partitions for both datasets (as decided by YARN), but we set the number of partitions of the join's resulting relation to 1, 2, 4, 8, and 16. We performed this experiment mostly to sanity check the validity of our prior results, in which we had not explicitly defined the number of partitions for the join's resulting relation. Specifically, we wanted to ensure that lazy loading was not invalidating our results.

Finally, we repeated the experiment once more, this time explicitly partitioning the Transaction data on ElementKey, whereas we had previously not specified any columns in the repartition command. We tested partition counts of 1, 4, 16 and 64, and we performed the join six times for each configuration permutation. A colleague had recommended this step as a potential way to see improvements in the times to process the join, since we were confused by the consistency of join times observed in our prior experiments.

III. *Results*

In all but the last of the trails above we saw surprisingly consistent execution times across all configurations and permutations. In the first test, we saw a mean execution time of 5:18.1, with a standard deviation of 2.3 seconds and a minimum and maximum of 5:04.3 and 5:29.4, respectively, across all Transaction and Blockface partitions counts. Broadcasting the Blockface data did not help, as we saw a mean execution time of 5:41.5 with a standard deviation of 2.5 seconds and a minimum and maximum of 5:38.9 and 5:45.4, respectively, across all Transaction and Blockface partition counts in this test. Partitioning the joined relation did not speed things up either, as we saw a mean execution time of 5:46.2 with a standard deviation of 3.5 seconds and a minimum and maximum of 5:42.0 and 5:51.0, respectively. The timings for all of these experiments were shockingly consistent, especially given the amount of variability in the configurations of each and our assumption that partitions would greatly improve performance.

The only experiment for which we saw slightly different timings was the one in which we partitioned the Transaction data on a specific column (i.e. ElementKey), but even in this case the results were not consistent. For Transaction partition counts of 4, 16 and 64 we saw timings very similar to those reported above; however, when using a single partition "partitioned on" Blockface, some of the trails were completed under the 5-minute mark, while the rest were completed in roughly the same amount of time as the prior experiments.

In this final experiment – with a single partition on the Blockface data, "partitioned on" Element Key – we observed that the execution time increased from approximately three minutes in the first trial to approximately 5.5 minutes (consistent with the other experiments) in the latter trials. Thinking that the speed increase may be related to the number of partitions of the joined relation itself, we printed the number of partitions for the joined relation and observed that in some of the trails this relation was split into 33 partitions, while in others it was split into 40. However, these splits did not seem to correspond to the execution times: the execution time had increased for each subsequent trial; but we observed that the 2nd, 3rd and 4th trials were split into 33 partitions, with the 1st, 5th and 6th split into 40 partitions.

Thinking these results looked odd, we performed this experiment again and were surprised to get similar – but not identical – results. On this second attempt, we again saw faster execution times for the first few trails, and slower times (closer to those observed in prior experiments) in the later

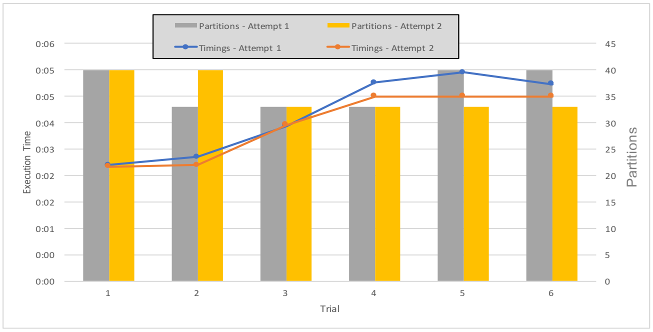

Figure 2: Execution Time and Join Relation Partitions

trials. The number of partitions for the joined relation again was observed to be 33 or 40. However, the different partition counts were in a different order than the prior experiment, and again did not appear to correlate with the differences in the execution times we observed.

The results of these two single-partition “partitioned on Blockface” experiments are shown in Figure 2 above. The orange and blue lines represent the timings for performing the join for each trial; the axis is on the left. The grey and yellow bars in the background represent the number of partitions reported for the joined relation for each trial; the axis is on the right.

As shown, both attempts exhibited similar timings; however, these timings do not appear to correspond directly to the number of partitions assigned to the resulting joined relation. In the second attempt, the faster speeds were observed on the trails in which the resulting join relation was partitioned into 33 pieces, rather than 40. However, this pattern was not observed on the first attempt, in which partitions of 40 corresponded to both runtime extremes.

5. CONCLUSIONS

We explored the speed-up and scale-up of Apache Spark using PySpark’s Dataframe API and its GMM function from the ‘pyspark.ml’ module. We observed near-linear scale-up when testing GMM fit times on databases of different sizes; however, we saw surprisingly little speed-up when we used larger clusters to perform the same task.

In Research Question 1, we explored how cluster size and data partition count interact to affect the time it takes to fit a GMM. We showed that at least for m2.2xlarge clusters, increasing node count has no effect on performance, and we found that performance improves with increasing data partition count, but at a certain point more data partitions fail to further improve execution time.

In Research Question 2 we explored how data partitioning and data broadcasting interact to affect the time it takes to perform a simple join. We found that at least for m2.2xlarge clusters, performance is not affected by partition count of either of the two joined dataframes, nor is it affected by the broadcast status of one of the two dataframes. We found that partition count does appear to affect the time it takes to operate on the join’s resulting relation; however, our results were inconsistent and therefore we could not definitively confirm to what extent this had an effect, if any.

6. ACKNOWLEDGMENTS

We thank Parmita Mehta for providing the Sloan Digital Sky Survey data used in the prior work and Research Question 1.

7. REFERENCES

[1] M. Zaharia et al. *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing.* In NSDI, 2012.

[2] *Spark SQL: Relational Data Processing in Spark*. Michael Armbrust, Reynold Xin, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael Franklin, Ali Ghodsi, Matei Zaharia. ACM SIGMOD Conference 2015, May. 2015.